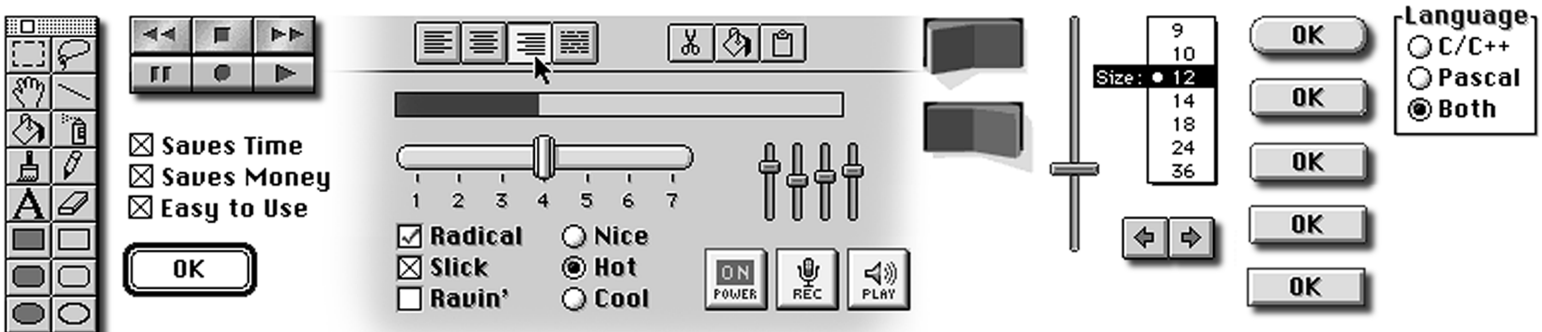
Tools Plus

PROFESSIONAL

LIBRARIES + FRAMEWORK



C / C++ and Pascal

System 6, 7 and Mac OS 8

User Manual

Tools Plus

PROFESSIONAL

LIBRARIES + FRAMEWORK

Version 6

Water's Edge Software
2441 Lakeshore Road West
Box 70022
Oakville, Ontario
Canada, L6L 6M9

Important Information for Evaluation Kit Registrants

A special edition of Tools Plus is distributed as an Evaluation Kit that can be obtained, free of charge, from user groups and various electronic bulletin boards and the Internet. Users of the Tools Plus Evaluation Kit are bound by restrictive terms and conditions that do not apply to *registered* Tools Plus developers who have purchased a license.

If you have obtained a Tools Plus Development Kit as a result of registering an Evaluation Kit, discontinue using the evaluation kit and take advantage of the latest Tools Plus features. You must recompile your applications using the licensed libraries that come with the Development Kit. Do not revert to using editions of Tools Plus that are distributed as evaluation software.

Free Updates and Software Upgrades

Please see the Technical Support chapter at the end of this manual for important information about receiving free software updates and free upgrades.

Copyright ©1989-2001 Water's Edge Software

Tools Plus™, Tools Plus Professional™, Tools Plus Pro™, Tools Plus Academic™, and Tools Plus Lite™ are trademarks of Water's Edge Software.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of Water's Edge Software.

4th Dimension® is a registered trademarks of ACIUS
Adobe®, Acrobat® and Photoshop® are trademarks of Adobe Systems Incorporated. Acrobat Reader copyright © 1987-1997 Adobe Systems Incorporated.
Apple® and Lisa® and are registered trademarks of Apple Computer, Inc.
Finder™, Macintosh®, MultiFinder™, QuickTime™ and ResEdit™ are trademarks of Apple Computer, Inc. Power Macintosh™ is a trademark of Apple Computer, Inc. used under license
MacPaint® and MacDraw® are registered trademark of Claris Corporation
Infinity Windoid™ is a trademark of Infinity Systems
PowerPC™ is a trademark of International Business Machines Corporation, used under license therefrom
Resorcerer® is a registered trademark of Mathemæsthetics Inc.
CodeWarrior™ is a trademark or Metrowerks Inc.
Microsoft® and Word® are registered trademark of Microsoft Corporation
Symantec C™, Symantec C++™, THINK C™, THINK Pascal™ and THINK Reference™ are trademarks of Symantec Corporation
Eudora™ is a trademark of the University of Illinois Board of Trustees, licensed to Quallcomm Incorporated

Published in Canada.

Tools Plus libraries, framework, and user manual were designed and created by Steve Makohin and Steven Waters.

We express gratitude to Marcel Achim of Metrowerks for his work on the CodeWarrior compiler. Without his assistance, dedication and responsiveness, Tools Plus for CodeWarrior would not be here today.

Thanks to Marlene Atcheson, Phil Calippe, Greg Galanos, Intellisoft Development Inc., “Kevin” at Symantec, Trent McLeod, Tony Minichillo, Herb Payerl, Diane Postill, Karen Postill, Rick Ruse, and Stan Witkowski. Thank you to Ken Bereskin of Apple for his technical expertise in Macintosh programming in the early days, and to Les Titze for his technical prowess and encouragement when Tools Plus was in its infancy.

A special thank you goes to Greg Kowal for his direction, insight and mentoring, and to Eugene Roman for his business sense and his chutzpah. And of course, thank you to all Tools Plus developers and beta testers for making Tools Plus a success!

This document was created on a Macintosh computer using Word 5.1a, MacPaint, MacDraw and Photoshop applications. Tools Plus and its user manual were designed and created entirely on Macintosh computers. Information in this manual is subject to change without notice.

Tools Plus™ and SuperCDEFs™
Software License and Support Agreement (SLSA), and Limited Warranty

This legal document is an agreement between Water's Edge Software of 2441 Lakeshore Road West, #70022, Oakville, Ontario, Canada, L6L 6M9, and you, the licensee, (herein referred to as "**LICENSEE**"). This legally binding agreement takes effect when signed by Water's Edge Software and the LICENSEE, or when you open the CD wrapping, which ever comes first, and is in effect for the duration, and under the conditions stated herein. All parts of this agreement, those being [i] Software License, [ii] Support Agreement, [iii] Limited Warranty, and [iv] Acknowledgment, are collectively referred to herein as the "**SLSA**."

BY SIGNING THIS SLSA and/or OPENING THE CD PACKAGE, YOU ARE AGREEING TO BECOME BOUND BY THE TERMS OF THIS SLSA, WHICH INCLUDES THE SOFTWARE LICENSE, SUPPORT AGREEMENT, LIMITED WARRANTY, and ACKNOWLEDGMENT.

In order to preserve and protect its rights under applicable law, Water's Edge Software does not **sell** any rights in its SOFTWARE. Rather, Water's Edge Software grants the right to **use** its SOFTWARE by means of an SLSA. Water's Edge Software specifically **retains title** to all Water's Edge Software computer software.

SOFTWARE LICENSE

1. SCOPE OF LICENSE: This Software License's scope pertains to the following products and items that are contained on the Tools Plus Professional 6 CD-ROM:

- [i] Tools Plus libraries, in compiled form
- [ii] Tools Plus libraries, in source code form
- [iii] Tools Plus interface files for Pascal, and header files for C/C++
- [iv] Tools Plus framework source code
- [v] SuperCDEFs control definition ('CDEF') resources
- [vi] SuperCDEFs source code
- [vii] SuperCDEFs source code interface files
- [viii] Tools Plus User Manual
- [ix] SuperCDEFs User Manual
- [x] Any documentation otherwise enclosed on the CD

2. TERMINOLOGY: A common terminology is used throughout this agreement as follows:

- [i] Tools Plus and its related files, regardless of their form, and SuperCDEFs and its related files, regardless of their form, are referred to herein collectively as "**SOFTWARE**". This includes, but is not limited to the following items from section 1: i, ii, iii, iv, v, vi, vii. This also includes all variations of Tools Plus libraries, Tools Plus source code, SuperCDEFs 'CDEF' resources, and SuperCDEFs source code, and the interface and/or header files that are related to Tools Plus and/or SuperCDEFs, including variants of these items that are modified by you, the LICENSEE, or by other LICENSEES as part of the Water's Edge Software Open Source Program, described herein.
- [ii] A subset of the SOFTWARE is "**SOURCE CODE**" that is comprised of all variations of Tools Plus source code, SuperCDEFs source code, and the interface and/or header files that are related to Tools Plus and/or SuperCDEFs, including variants of these items that are modified by you, the LICENSEE, or by other LICENSEES as part of the Water's Edge Software Open Source Program, described herein. This includes, but is not limited to the following items from section 1: ii, iii, iv, vi, vii.
- [iii] The Tools Plus User Manual and the SuperCDEFs user manual, regardless of their form, that being electronic, printed, or otherwise, are referred to herein collectively as "**DOCUMENTATION**". This includes, but is not limited to the following items from section 1: viii, ix, x.
- [iv] The term "**distribute**" is used to denote when an entity is exposed, or allowed to be exposed to any person or entity other than the LICENSEE.

3. GRANT OF LICENSE. In consideration of payment of the License fee, which is a part of the price you paid for Tools Plus, and your agreement to abide by the terms and conditions of this SLSA, Water's Edge Software, as Licensor, grants you, the LICENSEE, a non-exclusive right to use and display a copy of the SOFTWARE on a single COMPUTER (i.e., a single-user CPU) at a single location, so long as you comply with the terms of this SLSA. The LICENSEE is also granted a non-exclusive right to use and display a copy of the DOCUMENTATION. Water's Edge Software reserves all rights not expressly granted to the LICENSEE.

4. OWNERSHIP OF SOFTWARE. As the LICENSEE, you own the magnetic disk, CD-ROM, or other physical media on which the SOFTWARE is originally or subsequently recorded or fixed, but an express condition of this License is that Water's Edge Software retains title and ownership of the SOFTWARE and DOCUMENTATION, regardless of the form or media in or on which the original and other copies may exist. This SLSA is not a sale of the original SOFTWARE or DOCUMENTATION or any copy or of any variant of the SOFTWARE.

5. MODIFICATION RESTRICTIONS. You, the LICENSEE, may modify the SOURCE CODE providing that your modifications are confined to changes that are compiled into the Tools Plus libraries, and/or are compiled into a SuperCDEF 'CDEF' resource or a functional equivalent thereof in newer versions of the Macintosh Operating System. You may also modify the Tools Plus header files (C/C++) and interface files (Pascal) to reflect changes you make to the SOURCE CODE. Only original Tools Plus source code, or variants thereof, may be compiled into Tools Plus libraries. You may add new routines to Tools Plus libraries. You may modify the Tools Plus framework.

6. COPY RESTRICTIONS. The SOFTWARE and DOCUMENTATION are copyrighted, and are protected by Canadian and United States copyright laws, and international treaty provisions. Water's Edge Software retains these copyrights, including copyrights to modified SOURCE CODE. The LICENSEE agrees to treat modified SOURCE CODE in the same fashion as original SOURCE CODE from Water's Edge Software.

(6.1) Unauthorized copying of SOFTWARE or DOCUMENTATION, including SOFTWARE that has been modified, merged, or included with other software, is expressly forbidden unless otherwise stated in this SLSA. You may be held legally responsible for any copyright infringement that is caused or encouraged by your failure to abide by the terms of this SLSA.

(6.2) Water's Edge Software grants you, the LICENSEE, the right to integrate compiled Tools Plus libraries with source code produced by you in your development of executable applications and "plug-ins" using the SOFTWARE. Water's Edge Software also grants you the right to compile the SOFTWARE and to imbed the resulting object code into executable applications and "plug-in" products that you have developed, and the right to distribute such products with such imbedded object code, without royalty to Water's Edge Software, PROVIDED that you: (a) imbed the object code in such a manner as to prevent its extraction from your products, or access from your products in a form that would allow it to be imbedded in another executable application, or accessed by another application or "plug-in"; (b) agree to indemnify, hold harmless, and defend Water's Edge Software from any claims or lawsuits, including attorney's fees, that may arise from the use or distribution of your products containing such imbedded object code. Whether the imbedding of such object code complies with this SLSA shall be subject solely to the reasonable determination of Water's Edge Software. For the purpose of making any such determination, you agree to provide Water's Edge Software, at its request, and at not cost to Water's Edge Software, a copy of any executable application or "plug-in" you have developed that contains object code compiled from this SOFTWARE.

(6.3) You may distribute SuperCDEFs 'CDEF' resources, or derivatives thereof, only as part of an executable application or "plug-in" that you create.

(6.4) This SLSA expressly forbids the distribution of the SOFTWARE in a form that other developers may access, which includes, but is not limited to libraries and/or 'CDEF' resources that are based on the SOFTWARE. You may, however, create an executable application or "plug-in" that is used by developers, such as an application generator or source code generator, providing that the resulting application, source code, or other produced item are not dependent upon the SOFTWARE in order to perform its function.

(6.5) You, the LICENSEE, may print a single copy of the electronic user manual for your own use.

Except as specifically provided above, you shall not copy, modify, transfer, license, sublicense, rent, lease, sell, convey, translate, convert to any programming language or format or decompile or disassemble the SOFTWARE, or any portion of the SOFTWARE, **nor assign or transfer the license or any interest herein**.

7. USE RESTRICTIONS. As the LICENSEE, you may physically transfer the SOFTWARE from one computer to another, provided that the SOFTWARE is used on only one computer at a time. You may not electronically transfer the SOFTWARE from one computer to another over a network. You may not distribute, or allow to be distributed, copies of the SOFTWARE or DOCUMENTATION to others. You may not modify, adapt, translate, reverse engineer, decompile, disassemble, or create derivative works based on the SOFTWARE unless specifically provided in the SLSA.

8. TRANSFER RESTRICTIONS. This SOFTWARE is licensed only to you, the LICENSEE, and may not be transferred to anyone without the prior written consent of Water's Edge Software. Any authorized transferee of the SOFTWARE shall be bound by the terms and conditions of this SLSA. In no event may you transfer, assign, rent, lease, sell or otherwise dispose of the SOFTWARE, on a temporary or permanent basis, except as expressly provided herein.

9. COPYRIGHT NOTICE. Applications and "plug-ins" created with the SOFTWARE must prominently and legibly display a copyright notice in their startup window and/or "About…" box using a font that is no smaller than 9 points shown in high-contrast colors. Any documentation relating to applications and/or "plug-ins" that are dependent upon Tools Plus libraries, regardless of its form, must also display a Water's Edge Software copyright notice. One of the following notices must be used, or the LICENSEE must obtain permission in writing from Water's Edge Software to use an alternative notice.

(a) Tools Plus™ libraries copyright © 1989-2001 Water's Edge Software
(b) Created with Tools Plus™ © 1989-2001 Water's Edge Software
(c) Portions of this application © 1989-2001 Water's Edge Software. All rights reserved.

In the case of SuperCDEFs, one of the following notices must be used or the LICENSEE must obtain permission in writing from Water's Edge Software to use an alternative notice

(a) SuperCDEFs™ copyright © 1996-2001 Water's Edge Software
(b) Custom controls copyright © 1996-2001 Water's Edge Software
(c) Portions of this application © 1996-2001 Water's Edge Software. All rights reserved.

You many not modify the embedded copyright notice in SuperCDEFs unless you change the look or feel of the control. In such cases, you **must** replace the embedded copyright notice from Water's Edge Software with one of your own.

10. OPEN SOURCE PROGRAM: You, the LICENSEE, may work collaboratively with other developers who are licensed under this SLSA, and whose standing is in good order. This collaborative work may include, but not be limited to the exchange of source code, documentation, know-how, and other proprietary information. You may only provide, exchange or solicit such information through channels that are officially sanctioned by Water's Edge Software.

11. NON-DISCLOSURE: The SOFTWARE and DOCUMENTATION include proprietary information and trade secrets. You, the LICENSEE, agree to keep this information confidential, and to defend it from being distributed, or from becoming known to anyone, with the exception of your participation in the Open Source Program as defined herein. You agree to use whatever means are necessary, within reason, to ensure that this.

12. TERMINATION. This SLSA takes effect when it is signed by both the LICENSEE and by Water's Edge Software or when you open the CD wrapping, which ever comes first, and it remains in effect until it is terminated. This SLSA will terminate automatically without notice from Water's Edge Software if you, the LICENSEE, fail to comply with any provision of the SLSA. Upon termination you shall destroy all copies of the SOFTWARE and DOCUMENTATION, including modified copies and/or copies that are imbedded in other products, if any, or return them, postage prepaid, to Water's Edge Software.

(12.1) You, the LICENSEE, may voluntarily stop using the SOFTWARE and DOCUMENTATION on a permanent basis providing that you destroy the SOFTWARE and DOCUMENTATION, including modified copies and/or copies that are imbedded in other products, if any, or return them, postage prepaid, to Water's Edge Software.

(12.2) In the event that you, the LICENSEE, voluntarily stop using the SOFTWARE and DOCUMENTATION on a permanent basis, and/or if the SLSA is terminated, you **must continue** to indemnify, hold harmless, and defend Water's Edge Software from any claims or lawsuits, including attorney's fees, that may arise from the use or distribution of your products containing the SOFTWARE, even if those products are distributed outside of the terms of this SLSA.

(12.3) You agree that violation of this SLSA constitutes great and irreparable damage to Water's Edge Software, and that you may be held liable for such damages, including punitive damages, and be prosecuted to the fullest extent of the law.

SUPPORT AGREEMENT

In recognition that the LICENSEE has access to Tools Plus source code and SuperCDEFs source code, and that the LICENSEE has the ability to modify and append the SOURCE CODE, and that interdependencies may exist between portions of original, unmodified Water's Edge Software SOURCE CODE and the LICENSEE's modifications and/or additions, and with the understanding that the LICENSEE's efforts may negatively influence the SOFTWARE or cause it to fail, Water's Edge Software does not offer technical support for the SOFTWARE.

LIMITED WARRANTY

THE SOFTWARE AND DOCUMENTATION (INCLUDING INSTRUCTIONS FOR USE) ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. FURTHER, WATER'S EDGE SOFTWARE DOES NOT WARRANTY, GUARANTEE OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF USE, OF THE SOFTWARE OR DOCUMENTATION IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU, THE LICENSEE. IF THE SOFTWARE OR DOCUMENTATION ARE DEFECTIVE, YOU (AND NOT WATER'S EDGE SOFTWARE OR ITS DEALERS, DISTRIBUTORS, AGENTS, OR EMPLOYEES), ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION.

Water's Edge Software warrants to the original LICENSEE that the CD-ROM on which the SOFTWARE is recorded is free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of shipment. Water's Edge Software's entire liability and your exclusive remedy as to the CD-ROM shall be, at Water's Edge Software' option, either (a) return the purchase price or (b) replacement of the disk that does not meet Water's Edge Software Limited Warranty and which is returned to Water's Edge Software postage prepaid. If failure of this CD-ROM has resulted from accident, abuse, or misapplication, Water's Edge Software shall have no

responsibility to replace the CD-ROM or provide a refund. In the event of replacement of the CD-ROM, the replacement CD-ROM will be warranted for the remainder of the warranty period or thirty (30) days, whichever is the longer.

The above is the only warranty of any kind, either expressed or implied, statutory or otherwise, including but not limited to the implied warranties of merchantability and fitness for a particular purpose that is made by Water's Edge Software on this product.

No oral or written information or advice given by Water's Edge Software, its dealers, distributors, agents or employees shall create a warranty or in any way increase the scope of this warranty, and you may not rely on any such information or advice.

Neither Water's Edge Software, nor anyone else who has been involved in the creation, production, or delivery of the SOFTWARE or DOCUMENTATION shall be liable for any direct, indirect, consequential or incidental damages (including damages for the loss of business profits, business interruption, loss of business information, and the like) arising out of the use or inability to use such product even if Water's Edge Software has been advised of the possibility of such damages.

THIS WARRANTY GIVES YOU SPECIFIC RIGHTS. YOU MAY HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE (AND PROVINCE TO PROVINCE) AND CERTAIN LIMITATIONS CONTAINED IN THIS WARRANTY MAY NOT APPLY TO YOU. Water's Edge Software's liability to you for actual damages from any cause whatsoever, and regardless of the form (whether in contract, tort (including negligence), product liability or otherwise), will be limited to $50.

ACKNOWLEDGMENT

You acknowledge that you have read the SLSA, understand it, and agree to be bound by its terms and conditions. You also agree that the exclusive statement of agreement between the parties and supersede all proposals or prior oral or written agreements, or any other communications between the parties relating to the subject matter of the SLSA.

Contents

1 Introduction to Tools Plus

Tools Plus Overview 27
- Differences Between Tools Plus Packages 27
- The (Optional) Tools Plus Framework 28
- Visual Design Environment 28
- Tools Plus User Manual Formats 28
- Printing the User Manual 29

Registered Developer Benefits Period 29
- An Ordinary Application’s Architecture (without Tools Plus) ... 30
- A Tools Plus Application’s Architecture 32
- Powerful Features Using Simpler, Higher-Level Coding 33
- Event Processing is Virtually Eliminated 34
- Apple Event Support is Built into Tools Plus 34
- Supports Resource-Based and Dynamic Interface Design 35
- Accessing Tools Plus Libraries 35
- Creating New Applications With Tools Plus 36
- Using Tools Plus in an Existing Application 36

Range Checking 37
The Tools Plus Advantage 37
Who can benefit from Tools Plus 39
What kind of applications can be written with Tools Plus 39
What is Tools Plus not suitable for 40
System Requirements 40
Tools Plus Performance 41
Special Considerations with Mac OS Versions 42
Assumptions made when this manual was written 43
Conventions used throughout this manual 43
Software Updates 44
Evaluation Kit Registrants 44
For your information (recommended reading) 44
How to Get Started with Tools Plus 45
Stress Testing Applications 45
- Spotlight and other Testing Tools 46

Creating and Editing Resources 46
The List Manager, List Boxes, Tables and Beyond 47
Tools Plus Features 48

2 Installing Tools Plus

Installing Tools Plus in CodeWarrior C (68K) 59
- Adding Tools Plus to a CodeWarrior C (68K) Project 60
- Adding Tools Plus to a CodeWarrior C (68K) Plug-In 60

Installing Tools Plus in CodeWarrior Pascal (68K) 62
- Adding Tools Plus to a CodeWarrior Pascal (68K) Project 63
- Adding Tools Plus to a CodeWarrior Pascal (68K) Plug-In 63

Installing Tools Plus in CodeWarrior C (PPC) 65
- Adding Tools Plus to a CodeWarrior C (PPC) Project 66
- Adding Tools Plus to a CodeWarrior C (PPC) Plug-In 66

Installing Tools Plus in CodeWarrior Pascal (PPC) 67
- Adding Tools Plus to a CodeWarrior Pascal (PPC) Project 68
- Adding Tools Plus to a CodeWarrior Pascal (PPC) Plug-In 68

Installing Tools Plus in THINK C/C++ (68K) 5, 6 and 7 69
- Adding Tools Plus to a THINK C (68K) Project 69

Installing Tools Plus in Symantec C/C++ (68K) 8.0.5 or later 70
Adding Tools Plus to an SPM C/C++ (68K) Project 70
Installing Tools Plus in Symantec C/C++ (PPC) 8.6 or later 71
Adding Tools Plus to an SPM C/C++ (PPC) Project 71
Installing Tools Plus in THINK Pascal (68K) 72
Adding Tools Plus to a THINK Pascal (68K) Project 72
After Compiling 73
Compiling the CodeWarrior C (68K) Demo Application 74
Compiling the CodeWarrior Pascal (68K) Demo Application 75
Compiling the CodeWarrior C (PPC) Demo Application 76
Compiling the CodeWarrior Pascal (PPC) Demo Application 77
Compiling the THINK C (68K) 5, 6 or 7 Demo Application 79
Compiling the SPM C/C++ (68K) 8 Demo Application 80
Compiling the SPM C/C++ (PPC) 8 Demo Application 81
Compiling the THINK Pascal (68K) Demo Application 82

3 Designing Your Application

Overview 83
High Level Structure of a Tools Plus Application 84
A Macintosh Event, in Brief 84
Macintosh Event Queue 84
Key Up Events 85
Tools Plus Events, and the Event Loop versus an Event Handler 85
The Event Handler Routine 86
Recursion in the Event Handler Routine 87
System 5 and 6’s Finder/MultiFinder, and System 7 and higher 88
Finder 88
MultiFinder 88
System 7 and higher 89
The C Header file (ToolsPlus.h) 89
Pascal Strings versus C Strings 89
Using C and/or Pascal strings in Tools Plus parameters 89
Setting your prefixes 90
Appearance Manager 91
Multi-system compatibility with custom window & controls 91
Using the Appearance Manager 91
Embedding Controls 92
Dialogs and the Dialog Manager 92
Power Macintosh Performance 92
Off-screen GrafPorts and GWorlds 93
Writing Plug-Ins or External Code Modules 93
What to read next 96

4 Initialization

Initializing Tools Plus 97
Stack and heap 98
Other application initializing activities 99
Initialization Failure 104
Other Initialization 104
The Cursor 104
Deinitializing Tools Plus 105
Set maximum stack size in a 680x0 application 105
Change an application’s maximum stack size 106
Setting the parameter range error action routine 106

5 Windows

Overview 109
Resource-Based Programming 109
Window Types 111
Title Bar, Close box, and Zoom box 111
Size Box 112
Color Backdrops and Background Themes 112
Maximum Number of Open Windows 112
Tool Bar and Floating Palettes 113
Standard Windows 113
Active Window 113
Work Window 114
Current Window 114
Editing Field Window 114
Modal Windows 115
Window Layers 115
Global and Local Co-ordinates 116
Objects in Windows 116
The ‘dftb’ Resource - Font and Color Settings 116
Substituting Window ProcIDs 118
Live Window Dragging and Resizing 118
Special Considerations 118
Handling Windows 118
Opening a window 119
Appearance and Behavior Specification 120
Floating Palette, Custom WDEFs and Appearance Manager 122
Opening a dialog 127
Attach a dialog list to a window 131
Appending a dialog list to a window 131
Setting defaults for fields created from edit text items 132
Setting defaults for fields created from static text items 132
Setting defaults for editing fields created by ‘CNTL’ resources 132
Setting defaults for static text fields created by ‘CNTL’ resources 133
Setting defaults for list boxes created by ‘CNTL’ resources 133
Setting defaults for pop-up menus created by ‘CNTL’ resources 133
Opening a tool bar 134
Tool bar inside a window 134
Getting the first unused window number 135
Setting the backdrop color for new windows 135
Clearing the backdrop color for new windows 135
Setting the backdrop color for an open window 136
Setting the background theme for an open window 136
Setting the background theme for the next new window 137
Closing a window, tool bar, or palette 137
Changing a window’s size 138
Moving a window 138
Hiding/showing a window 139
Activating a window 140
Remove keyboard focus from a window 140
Making a window current without activating it 141
Making the active window current 141
Changing a window’s title 141
Setting a window’s size limits 142
Setting the “standard” and “user” co-ordinates for zooming 142
Getting the “standard” and “user” co-ordinates for zooming 143
Setting a dialog item’s display rectangle 143
Getting a dialog item’s display rectangle 144
Setting font settings for new dialogs as they are created 144

Getting font settings used by new dialogs as they are created 144
Getting a window's status information 145
Manually refreshing user interface elements 146
Determine the Nth window from the front...................................... 147
Getting your app's active window number 148
Getting your app's current window number 148
Getting your app's frontmost window number 148
Getting your app's tool bar number 149
Getting your app's frontmost floating palette's number 149
Getting your app's frontmost standard window number.................. 149
Getting your app's work window number 150
Getting the window number of your app's active edit field............. 150
Getting the window number containing the keyboard focus 150
Determining if a window is open 151
Determining if a window is visible 151
Determining if a window is active 151
Determining a window's type 152
Determining which object has the keyboard focus 152
Getting a window's pointer 153
Auto-resizing subsequently created objects 153
Hiding/showing the Finder and other applications 153
Turn the live window dragging/resizing option on or off 154
Replace a window type throughout the application 154
The Infinity Windoid 155

6 Buttons

Overview 157
Button Types 157
Button States 157
Button Titles.. 158
Fonts... 158
Colors 158
Default Button 158
Selecting Buttons and Command Keys.................................... 158
Substituting Button ProcIDs 158
Handling Buttons 159
Appearance Manager Controls 159
Push Button (CDEF 23) 160
Check Box (CDEF 23)... 160
Radio Button (CDEF 23) 160
Bevel Button (CDEF 2) 160
Tabs (CDEF 8).. 161
Disclosure Triangles (CDEF 4) 162
Clock (CDEF 15) 162
Group Box (CDEF 10).. 162
Chasing Arrows (CDEF 7).. 163
Little Arrows (CDEF 6) 163
Static Text (CDEF 18) 163
Placard (CDEF 14).. 163
Visual Separator (CDEF 9)... 164
Image Well (CDEF 11)... 164
Pop-Up Arrow (CDEF 12).. 164
Picture Control (CDEF 19) 164
Icon Control (CDEF 20) 165
Window Header (CDEF 21) 165
User Pane (CDEF 16) 165
Appearance Manager and Keyboard Focus 166
Creating a new button 166

Appearance and Behavior Specification 167
Custom Control Definitions (CDEFs)...................................... 168
Automatically embedding controls 171
Embedding a button into a button 172
Embedding a button into a scroll bar.. 172
Getting the first unused button number 173
Setting colors for new buttons as they are created 173
Resetting the colors for new buttons to the default 173
Deleting a button 174
Hiding/showing a button 174
Determining if a button is visible 175
Activating a button (giving it the keyboard focus) 175
Getting a button’s co-ordinates 176
Enabling/Disabling a button 176
Determining if a button is enabled 177
Selecting/Deselecting a button 177
Determining if a button is selected... 177
Getting a button’s minimum value limit 178
Setting a button’s minimum value limit 178
Getting a button’s maximum value limit.. 178
Setting a button’s maximum value limit 179
Getting a button’s current value 179
Setting a button’s current value 179
Changing a button’s title 180
Flashing a button (simulating selection) 180
Moving a button 180
Changing co-ordinates without moving the image 181
Changing a button’s size 181
Changing a button’s co-ordinates.. 181
Specifying how a button is automatically moved/resized 182
Setting a button’s font, size and style settings................................ 183
Getting a button’s font, size and style settings 183
Setting a button’s colors 184
Getting a button’s colors 184
Setting a default button 185
Removing the “default button” status from a window 185
Getting a button’s control handle 185
Replace a button type throughout the application 186

7 Picture Buttons

Overview 187
Button Types 187
Button Behavior 187
Selection Effects.. 188
Disabling Effects 188
Button’s Value and Stages 188
Handling Picture Buttons 189
Creating a new picture button 189
Resource IDs 190
Icon Resource IDs 190
3D SICN Buttons 191
PICT Resource IDs 191
Behavior and Appearance Specification 192
Rate of Repeating Events 197
Picture Buttons on Color Backgrounds..................................... 197
Getting the first unused picture button number 198
Deleting a picture button 199
Hiding/showing a picture button 199

Determining if a picture button is visible 200
Getting a picture button’s co-ordinates 200
Enabling/Disabling a picture button.. 201
Determining if a picture button is enabled 201
Selecting/Deselecting a picture button 201
Determining if a picture button is selected...................................... 202
Getting a picture button’s minimum value limit 202
Setting a picture button’s minimum value limit.............................. 202
Getting a picture button’s maximum value limit 203
Setting a picture button’s maximum value limit 203
Getting a picture button’s current value 203
Setting a picture button’s current value 204
Setting a picture button’s value and selection state 204
Setting a picture button’s value change rate..................................... 205
Setting a picture button’s value change speed 205
Flashing a picture button (simulating selection) 206
Moving a picture button 206
Changing co-ordinates without moving the image 206
Specifying how a picture button is automatically moved 207

8 Scroll Bars

Overview 209
Scroll Bar States.. 210
Colors 210
Text 210
Scroll Bar Speed 210
Substituting Scroll Bar ProcIDs... 211
Handling Scroll Bars 211
Processing doScrollBar Events... 211
Action routine 211
Appearance Manager Controls 212
Scroll Bar (CDEF 24) 213
Slider (CDEF 3) 213
Progress Indicator or “Thermometer” (CDEF 5)............... 213
Little Arrows (CDEF 6) 213
Appearance Manager and Keyboard Focus 214
Creating a new scroll bar 214
Appearance and Behavior Specification 215
Custom Control Definitions (CDEFs)...................................... 216
Embedding a scroll bar into a button 218
Embedding a scroll bar into a scroll bar... 218
Getting the first unused scroll bar number 219
Setting colors for new scroll bars as they are created 219
Resetting the colors for new scroll bars to the default 220
Deleting a scroll bar 220
Hiding/showing a scroll bar 221
Determining if a scroll bar is visible 221
Activating a scroll bar (giving it the keyboard focus)...................... 222
Getting a scroll bar’s co-ordinates 222
Enabling/Disabling a scroll bar... 223
Determining if a scroll bar is enabled 223
Getting a scroll bar’s minimum value limit 223
Setting a scroll bar’s minimum value limit..................................... 224
Getting a scroll bar’s maximum value limit.................................... 224
Setting a scroll bar’s maximum value limit 224
Getting a scroll bar’s current value 224
Setting a scroll bar’s current value 225
Moving a scroll bar 225

Changing co-ordinates without moving the image 225
Changing a scroll bar’s size 226
Changing a scroll bar’s co-ordinates 226
Specifying how a scroll bar is automatically moved/resized 227
Setting a scroll bar’s font, size and style settings 228
Getting a scroll bar’s font, size and style settings 228
Setting a scroll bar’s colors 229
Getting a scroll bar’s colors 229
Setting the line scrolling speed for new scroll bars 230
Setting the page scrolling speed for new scroll bars 230
Setting a scroll bar’s line scrolling speed 231
Setting a scroll bar’s page scrolling speed 231
Setting a scroll bar’s action routine 231
Getting info in a scroll bar’s action routine 233
Getting a scroll bar’s control handle 233

9 Editing Fields

Overview 235
The Field’s String 235
Dynamic String Handles 235
The Active Field 236
Editing Field Window 236
Activating a Field and Editing Text 236
Length Limited Fields 237
Clicking and Tabbing 237
Keyboard Focus on Tool Bars and Floating Palettes 238
Alignment of Text in a Field 239
Fonts 239
Colors 239
Disabled Fields 240
Filtering Characters 240
Word Wrap 240
User Interaction with Fields 241
Mac 512KE and Mac Plus keyboard with numeric pad 243
The Edit Menu 243
Large Fields and Buffers 244
Fields with Scroll Bars 244
Memory Management 245
Desk Scrap 245
TextEdit Scrap 246
Scrap “Undo” Text 246
Field’s String 246
Field’s Edited Text 246
Edited “Undo” Text 247
”Low Memory” Protection 247
Tips for Conserving Memory 247
Handling Fields 247
Special Handling of Fields 248
Appearance Manager and Keyboard Focus 248
Appearance Manager Controls 248
Edit Text (CDEF 17) 249
Static Text (CDEF 18) 249
Creating a Field Using a ‘CNTL’ Resource 249
Allocating memory for a field’s string 250
Creating a new field 250
Appearance and Behavior Specification 251
Single Line Fields 253
Embedding a field into a button 257

Embedding a field into a scroll bar 257
Getting the first unused field number 258
Deleting a field 258
Hiding/showing a field 259
Determining if a field is visible 259
Getting a field's co-ordinates 260
Setting a field's font, size and style settings 260
Getting a field's font, size and style settings 261
Setting a field's colors 261
Getting a field's colors 262
Activating a field (giving it the keyboard focus) 262
Getting a field's selection range 263
Setting a field's selection range 263
Deactivating a field 263
Enabling/Disabling a field 264
Determining if a field is enabled 264
Clicking in an inactive field or keyboard focus item 264
Detecting a Tab in an active field or keyboard focus item 265
Tabbing to the next/previous field or keyboard focus item 266
Getting the active field's edited text 267
Getting a handle to the active field's edited text 267
Getting the active field's edited text length 267
Getting a field's string 268
Getting a handle a field's string 268
Getting a field's string length 269
Determining if a field is empty 269
Saving the active field's edited text as the field's string 269
Getting the window number of your app's active edit field 270
Getting the active field's number 270
Turning field length limiting on/off 270
Set field length limiting for an existing field 271
Turning string handle resizing on/off 271
Set appearance and behavior for disabled fields 271
Set appearance and behavior for a disabled field 273
Pasting into a field under your application's control 274
Moving a field 276
Changing co-ordinates without moving the image 276
    Scrolling fields 276
Changing a field's size 277
Changing a field's co-ordinates 277
Specifying how a field is automatically moved/resized 278
Scrolling a field to its default position 279
Creating a new field filter 279
Apply a filter to subsequently created editing fields 280
Specify minimum free memory required after "undo" is set up 281
Specify minimum free memory for editing text 281
Specify "low memory while typing" threshold 282
Getting a field's TextEdit handle 282

10 List Boxes
Overview 283
    Auto-Positioning Options 284
    Fonts 284
    Colors 284
    Appearance Manager Controls 284
        List Box (CDEF 22) 285
    Creating a List Box Using a 'CNTL' Resource 285
    Appearance Manager and Keyboard Focus 285

Special Considerations ... 285
Handling List Boxes ... 286
Creating a new list box ... 286
Appearance and Behavior Specification ... 287
Embedding a list box into a button ... 290
Embedding a list box into a scroll bar ... 290
Getting the first unused list box number ... 291
Deleting a list box ... 291
Hiding/showing a list box ... 292
Determining if a list box is visible ... 292
Activating a list box (giving it the keyboard focus) ... 293
Getting a list box's co-ordinates ... 293
Adding a new line / replacing an existing line in a list box ... 294
Inserting resource names into a list box ... 295
Copy a set of strings to a list box ... 295
Getting a line's text ... 296
Searching lines for specific text (alphabetic order) ... 296
Selecting/Deselecting a line ... 297
Determine if a line is selected ... 297
Determine the next selected line number ... 298
Inserting a blank line into a list box ... 298
Deleting a line ... 299
Determining if a list box is enabled ... 299
Setting a list box's font, size and style settings ... 300
Getting a list box's font, size and style settings ... 300
Setting a list box's colors ... 301
Getting a list box's colors ... 301
Determining the number of lines in a list box ... 301
Turning a list box's drawing on/off ... 302
Moving a list box ... 302
Changing co-ordinates without moving the image ... 303
Changing a list box's size ... 303
Changing a list box's co-ordinates ... 303
Specifying how a list box is automatically moved/resized ... 304
Getting a list box's list handle ... 305

11 Pop-Up Menus

Overview ... 307
Fonts ... 308
Colors ... 308
Command Keys & Hierarchical Pop-Up Menus ... 309
Creating a Pop-Up Menu Using a 'CNTL' Resource ... 309
Pure System Pop-Up Menu ... 309
Tools Plus Pop-Up Menu (CDEF 63) ... 309
Bevel Button Pop-Up Menu (CDEF 2) ... 310
Handling Pop-Up Menus ... 310
Creating a new pop-up menu ... 311
Appearance and Behavior ... 312
Pop-Up Menus on Color Backgrounds ... 314
Embedding a pop-up menu into a button ... 316
Embedding a pop-up menu into a scroll bar ... 317
Getting the first unused pop-up menu number ... 317
Attaching or detaching a hierarchical menu to a pop-up menu ... 318
Setting colors for new pop-up menus as they are created ... 318
Resetting the colors for new pop-up menus to the default ... 319
Adding, changing or renaming a pop-up menu item ... 319
Metacharacters ... 319
Inserting a pop-up menu item ... 321

Inserting resource names into a pop-up menu 321
Deleting a pop-up menu or pop-up menu item 322
Getting a pop-up menu's co-ordinates 323
Hiding/showing a pop-up menu 323
Determining if a pop-up menu is visible 324
Getting a pop-up menu item's text 324
Renaming a pop-up menu item 325
Enabling or disabling a pop-up menu or pop-up menu item 325
Determining if a pop-up menu is enabled 326
Displaying or hiding the Check mark 326
Displaying or clearing special marks 326
Getting a pop-up menu item's special mark 327
Setting a pop-up menu item's icon 327
Getting a pop-up menu item's icon 328
Changing a pop-up menu item's style 328
Determining the number of items in a pop-up menu 328
Determining the selected item in a pop-up menu 329
Moving a pop-up menu 329
Changing co-ordinates without moving the image 329
Changing a pop-up menu's size 330
Changing a pop-up menu's co-ordinates 330
Specifying how a pop-up menu is automatically moved/resized 331
Setting a pop-up menu's font, size and style settings 331
Getting a pop-up menu's font, size and style settings 332
Setting a pop-up menu's colors 332
Getting a pop-up menu's colors 333
Setting a pop-up menu item's colors 333
Getting a pop-up menu item's colors 334
Getting a pop-up menu's control or menu handle 334

12 Panels

Overview 335
Color Tables 336
Creating a new panel 336
Appearance and Behavior Specification 337
Getting the first unused panel number 341
Setting the standard color table's colors 342
Getting the standard color table's colors 342
Setting the custom color table's colors 343
Getting the custom color table's colors 344
Deleting a panel 344
Hiding/showing a panel 345
Determining if a panel is visible 345
Getting a panel's co-ordinates 346
Moving a panel 346
Changing co-ordinates without moving the image 347
Changing a panel's size 347
Changing a panel's co-ordinates 347
Specifying how a panel is automatically moved/resized 348
Setting a panel's font, size and style settings 349
Getting a panel's font, size and style settings 349
Setting a panel's colors 350
Getting a panel's colors 350

13 Menus

Overview 353
- Menus in Plug-Ins 354
- Colors 354
- Menus Accessed by MultiFinder and System 7 or higher 354
- Edit Menu 355
- Menus and Editing Fields 356
- Apple Menu and Desk Accessories 356
- Menus and Desk Accessories 357
- Help Menu and Applications Menu 357
- Command Key Equivalents 358
- Planning for Balloon Help 358
- Handling Menus 358

Creating the Apple menu () 359
Creating and renaming a menu or menu item 359
- Metacharacters 360

Creating a menu using a ‘MENU’ resource 361
Creating a set of menus using an ‘MBAR’ resource 362
Identify the Select All edit menu item 363
Getting the first unused menu number 363
Getting the first unused hierarchical menu number 363
Attaching or detaching a hierarchical menu 364
Inserting a menu item 364
Inserting resource names into a menu 365
Deleting a menu or menu item 366
Updating the menu bar (redrawing it) 367
Hiding/showing a menu bar 367
Getting default menu colors for your application 368
Setting default menu colors for your application 368
Getting a menu’s colors 369
Setting a menu’s colors 369
Getting a menu item’s colors 370
Setting a menu item’s colors 370
Getting a menu item’s text 371
Renaming a menu item 371
Enabling or disabling a menu or menu item 372
Displaying or hiding the Check mark 372
Displaying or hiding special marks 373
Getting a menu item’s special mark 373
Setting a menu item’s Command-key equivalent 374
Getting a menu item’s Command-key equivalent 374
Setting a menu item’s icon 374
Getting a menu item’s icon 375
Changing a menu item’s style 375
Determining the number of items in a menu 376
Determining a menu’s parent menu 376
Determining a menu item’s submenu 377
Highlight or unhighlight a menu 377
Getting a menu’s handle 377

14 Cursors

Overview 379
- Color Cursors 379
- Automatic Cursor Changes 379
- The Watch Cursor 380
- Starting your application 380
- The Cursor Table 381
- Advanced Features 381

Cursor Animation 382
Handling Cursors 383
Changing the cursor's shape 383
Resetting cursor shape according to window orientation 384
Setting the cursor animation sequence 384
Keeping cursor animation running 384
Creating a new cursor table 385
Getting the first unused cursor table number 385
Deleting a cursor table 385
Creating/replacing a cursor zone (using a rectangle) 386
Creating/replacing a cursor zone (using a region) 386
Getting the first unused cursor zone number 387
Deleting a cursor zone 387
Changing the cursor for a cursor table or zone 387
Getting a cursor zone's bounding rectangle 388
Getting a cursor zone's region 388
Indicate that cursor zone regions have been altered 388
Making a window use a cursor table (or stop using one) 389
Determining which cursor zone contains a specified point 389
Determine which cursor zone contains the cursor 389
Enabling/disabling button clicks during a watch cursor 390

15 Balloon Help

Overview 391
Help Inheritance 392
Balloon Help for the Finder ('hfdr' resource) 392
Balloon Help for Menus ('hmnu' resource) 392
Balloon Help for Objects in Windows 392
Using 'hdlg' and/or 'hrct' Resources in Dialogs or Dialog Lists 393
Manually Assigning Help Resources Data to a User Interface Element 394
Manually Assigning Help Data Without Using Resources 395
hdlg' and 'hmnu' Resource Settings 397
Efficiently Storing Numerous Help Messages 397
Balloon Help Performance Issues 397
Issues with THINK Pascal 398
Setting Help for a Button 399
Setting Help for a Picture Button 401
Setting Help for a Scroll Bar 401
Setting Help for a Field or Static Text 402
Setting Help for a List Box 402
Setting Help for a Pop-Up Menu 403
Setting Help for a Panel 404
Setting Help for a Cursor Table 404
Setting Help for a Cursor Zone 405
Setting Help for a non-Tools Plus Control 406
Deleting a non-Tools Plus control 406
Forcing Recalculation of Balloon Help 407

16 Event Management

Overview 409
Polling versus Dispatching 409
Task Switching 410
Macintosh Events 411
The Event Queue 411
Watch Cursor -- a busy system 412

Tools Plus Event Record 412
Event Record Fields 415
Event Modifiers 416
    Event Modifiers Using C 417
    Event Modifiers Using Pascal 418
Background Processing 418
The Event Handler Routine 419
The Window Event Handler Routine 420
Modal Event Handling 421
Filtering Events (the Event Filter Routine) 421
Serial Events 422
Tools Plus Event Codes 423
Translating Toolbox events to Tools Plus events 424
Automatic Apple Event Support 426
Simulated Apple Event Support 427
Routines for Handling and Processing Events 429
    Setting an event handler routine for a window 429
    Setting an event handler routine for a window 429
    Process events continuously 430
    Process a single event while the application is busy 430
    Process a single toolbox event 431
    Set an Apple Event error 431
    Determine number of files to be opened or printed 432
    Retrieve file info for a file that needs to be opened or printed 432
    Stop processing events, return control to application 433
    Determine if Tools Plus is set to stop processing events 434
    Scheduling background processing 434
    Determining if “scheduling processing” is supported 435
    Wait for subsequent clicks 435
    Discontinuing multiple clicks or drags 435
    Ignoring the first click of a multiple click sequence 436
    Determining if your application is suspended 436
    Determine if Tools Plus is processing a series of events 437
    Stop Tools Plus processing a series of events 437
Timers and Timer Events 438
    How Tools Plus Generates Timer Events 439
    Timer Accuracy 439
    Timer Resolution 440
    Timers and doNothing (null) Events 441
    The Possibility of a Timer Overflow 441
    Creating a new Timer 442
Deleting a timer 445
Responding to Events 446
    doActivate 446
    doAutoKey 447
    doButton 448
    doChgInField 449
    doChgMonitorSettings 450
    doChgWindow 450
    doClick 451
    doClickControl 454
    doClickDesk 454
    doClickToFocus 454
    doDeactivate 455
    doGoAway 456
    doGrowWindow 457
    doKeyDown 457
    doKeyInControl 459

doKeyUp 459
doListBox 460
doManualEvent 461
doMenu 462
doMoveCursor 463
doMoveWindow 463
doNothing 463
doOpenApplication 464
doOpenDocuments 465
doPictButton 466
doPopUpMenu 467
doPreRefresh 467
doPrintDocuments 469
doQuitApplication 470
doRefresh 471
doResume 471
doScrollBar 472
doSuspend 473
doTimer 473
doZoomWindow 473
"Field To Event" Cross reference 475

17 Color Drawing & Multiple Monitors

Overview 477
Using One Monitor 477
Using Multiple Monitors 477
Physical Monitors 479
Detecting Monitor and Screen Changes 479
Changing Screen Settings 479
Determining if Color QuickDraw is used 479
Determining the number of logical screens 480
Beginning color-dependent drawing on a window 480
Ending color-dependent drawing on a window 481
Determining the number of colors on a screen 481
Determining if the screen is set to draw in color 482
Test for changes in monitor settings 482
Determining the number of physical monitors 482
Determining the number of colors on a monitor 483
Determining if the monitor is set to draw in color 483
Get a handle to the monitor's Graphics Device 484
Determine the main monitor number 484
Determining if a rectangle's area is visible in a window 484
Determining if a region is visible in a window 485
Getting a window's foreground color 485
Getting a window's background color 486
Setting a window's foreground color 486
Setting a window's background color 486
Storing a color's components in an RGB Color record 487
Erasing an area on a window 487
Erasing an area on a window 487
Calculating a disabled color 488
Resetting the current window's pen to default values 489
Using the system's highlight color 489
Drawing text on the highlight color 489
Highlighting a rectangle and preparing to draw text 490
Highlighting a region and preparing to draw text 491
Getting the current window's pen settings 491
Setting the current window's pen settings 492

18 User Notification

Overview 493
Notifying the User 493
Define settings for notifying the user 494
Notifying the user that your application needs attention 495

19 Dynamic Alerts

Overview 497
Multitasking in Dynamic Alerts 497
Icons 498
Text 498
Buttons 498
Routine’s Value 498
Automatic User Notification 498
Appearance Manager 499
Alert Samples 499
Displaying a dynamic alert 501
Custom Button Combinations 501
Advanced Techniques 502
Changing button titles on dynamic alerts 503
Getting preferences for dynamic alerts 503
Setting preferences for dynamic alerts 504
Allow/disallow doNothing events during alerts 505
Determine number of open dynamic alerts 505

20 Miscellaneous Routines

Overview 507
Drawing strings 507
Drawing text 509
Drawing a picture 509
PICT Resource IDs 510
Appearance and Behavior 510
Drawing a picture offset in its frame 513
Drawing an icon 514
Intelligent Icon Drawing 514
Icon Family 515
Icon Selection 515
Drawing the Icon, Selecting, Disabling, and Masking 516
Creating Your Own Icons 516
Set the default appearance for disabled icons 518
Maintaining Indexed String (‘STR#’) Structures 519
Creating an indexed string structure 519
Counting the number of strings 520
Getting a string 520
Setting a string 520
Inserting or appending a new string 521
Deleting a string 521
BitMaps and PixMaps 522
Creating a bitmap 522
Drawing to a bitmap 522
Copying to a bitmap or to a window 523
Creating a BitMap or PixMap 523
Destroying a BitMap or PixMap 524
Converting a BitMap or PixMap to a region 525
Determining the System version 525
Determining the Tools Plus version 526
Play the System Error sound 527

Wait for a specified time 527
Synchronizing to Vertical Retrace 528
Drawing “Zoom Lines” 528
Drawing a standard Macintosh progress thermometer 530
Determining if Appearance Manager is available 530
Determining if Appearance Manager routines are available 531
Determining if Appearance Manager is running 531
Set all bytes in a record to zero 532
Determine if two records are equal 532
Determining the minimum value of two numbers 533
Determining the maximum value of two numbers 533

21 Multiple Languages
Overview 535
Where do those words appear? 535
Changing the words 535
The STR# Resource 535
Changing the language 537

22 Other Macintosh Features
Overview 539
Alerts 539
Dialogs 539
Custom Controls 540
Lists 540

23 Memory
Overview 541
Testing Memory Requirements 541
Testing for Memory Availability 542
Editing Fields 543
Handle Blocks 543
The Style Table 544
Good memory habits 544

24 Font Heights
Font heights table 545

25 Special Routines
Use these routines with caution, or don’t use them! 547

26 Completing Your Application
Overview 553
Application’s Icons 554
Icon Family 554
File Types, Creators, and the Application Signature 555
Signature (the Creator code) 555
Bundle 556
Version 556
mstr Resources 557
SIZE’ Resource 558
Cloned SIZE resources 559

27 Technical Support

What does Technical Support do? 561
What doesn't Technical Support do? 561
Electronic Mail (Email) and Web Support 561
Mail Support 562
Fax Support 562
Telephone Support 562
Notification by Email 562
Updates and Upgrades by Email 562
Updates by the web 563
Mail updates 563
Tools Plus Developer Forum 563
Known Bug List 563
Bug Alert Service 563
Registered Developer Benefits Period 563
How to Submit Queries or Problem Reports 564

Index 567

1 Introduction to Tools Plus

Tools Plus Overview

The Tools Plus Libraries + framework lets Macintosh developers *easily* create professional looking applications using Metrowerks CodeWarrior, Symantec (THINK) C/C++, or THINK Pascal compilers. Additionally, CodeWarrior users can create plug-ins. Virtually any user interface element is created with a single line of simple code. Once created, elements work with each other without the need for additional support code, thereby letting you eliminate thousands of lines of source code. Over 80% of the effort is gone! Tools Plus can also bring Macintosh resources to life in ways that the Macintosh's toolbox can't.

User interface elements, everything from a simple button to a sophisticated dialog, come to life with a single line of code. Windows drag, zoom and resize. Buttons click. Pop-up menus pop. The Edit menu edits. Scroll bars scroll. In spite of Tools Plus's power and robust features, it is easy to learn and easy to use making it ideal for novices, intermediate developers, and experts alike.

Tools Plus supports and automates all standard user interface elements and seamlessly integrates support for popular extras like floating palettes, a tool bar, the best picture buttons in the industry, tabs, sliders, a complete "3D look" (with or without the Appearance Manager) and much more.

Using Tools Plus simplifies your programming and accelerates development. Less than 200 core Tools Plus routines replace the need for many hundreds of Mac toolbox routines, and tens of thousands of lines of source code. You'll create applications in less time, with much less source code, with far fewer bugs, and with more features than if you had used ordinary C/C++ or Pascal. The resulting executables are compact, lightning quick, and efficient.

Tools Plus libraries can be compiled into applications or plug-ins that run on any Macintosh (512KE or higher), Power Macintosh or Mac OS compatible running on System 6 using Finder or MultiFinder, System 7, or Mac OS 8 and higher. Tools Plus is royalty free. A single license lets you create an unlimited number of applications, and sell an unlimited number of copies.

Differences Between Tools Plus Packages

Tools Plus is available in a number of packages, each being tailored to a specific audience:

- **Tools Plus Pro**: This is the complete Tools Plus developer kit with libraries for CodeWarrior C/C++ and Pascal, Symantec C/C++, THINK C/C++, and THINK Pascal. 680x0 and PowerPC-native libraries are included. The user manual is in electronic format only (PDF, also known as Adobe Acrobat format which produces the best visual results and is suitable for printing, and eDoc format which requires the least resources and runs quickest). Also included as an added bonus are SuperCDEFs world class-controls, custom color window WDEFs, and other additional color resources. You must purchase this Software Development Kit in order to use it.
- **Tools Plus Lite**: Similar to Tools Plus Pro, except it has only CodeWarrior 680x0 libraries. Tools Plus Lite is ideal for developers who want the power of Tools Plus with a minimal investment. Tools Plus Lite is available through select channels for a limited time only. You can easily and economically upgrade to Tools Plus Pro.
- **Tools Plus Academic**: Similar to Tools Plus Pro, except that it does not include SuperCDEFs controls, and it bears "personal use" licensing restrictions. Tools Plus Academic is available only to students and members of faculty of accredited academic institutions. You can easily and economically upgrade to Tools Plus Pro.
- **Tools Plus Evaluation Kit**: This is the only Tools Plus kit that is available free of charge. You can get it from the Water's Edge Software web site, the Internet, electronic bulletin boards, and other sources. It is designed to give a developer the opportunity to try Tools Plus before buying it. It contains almost all of Tools Plus Pro's routines, and an electronic user manual in eDoc format only. SuperCDEFs are not included. The Evaluation Kit's Software License Agreement lets a developer try Tools Plus for thirty days, after which time he must either purchase Tools Plus (Pro, Lite or Academic), or stop using the Evaluation Kit.

The (Optional) Tools Plus Framework

The Tools Plus framework is included with Tools Plus libraries, but you don't have to use the framework because the power of Tools Plus is in the libraries. They replace the need for complex toolbox coding. The Tools Plus development kit also includes a framework that is essentially a "skeleton" for a fully functioning application. Realize that Tools Plus libraries are fully enabled and complete without using our framework. You can easily design your own framework as though you were doing "macro coding" because Tools Plus libraries do all the dirty work for you. Our framework is just one of many approaches you can use to write an application when you have Tools Plus libraries at hand. This is why we emphasize that using the Tools Plus framework is *optional*.

When you explore the Tools Plus framework, the surprising part will be how easily you can add "flesh" to the framework, that is to add the features, look and feel that make your application unique. That's because Tools Plus libraries have literally thousands of features and options that you can add to your application, simply by adding one line of code. Our framework is an excellent starting point for most kinds of developers:

Novice programmers are provided with an example of a full application showing them how to structure their project and how to create an application that not only has all basic functionality, but easily supports all Tools Plus features.

Programmers who are new to Macintosh can ease their transition from a UNIX or PC environment, and readily apply their existing skills to the Macintosh with the assistance of Tools Plus libraries. For their convenience, we provide the sample framework just so they don't have to start with a "blank sheet of paper."

New Tools Plus developers who already know how to program a Macintosh now have a jump start to help them get going right away. Our framework makes it easy to harness the power of Tools Plus libraries and get results fast.

See the "Framework Example" folder for complete details about the sample framework.

Visual Design Environment

There are a number of rapid development tools that are made up of a "visual designer" paired with a code generator. The appeal of these tools is that you can design your interface visually. Their drawback may have as significant an impact on you because these code generators do just that: they generate generous volumes of complex toolbox code that you will have to learn and maintain. Worse still, the code they generate often depends on an even larger, multi-megabyte C++ class library, or on a rigidly structured set of source code modules.

Tools Plus takes a very different approach. We removed the burden from the "visual designer" to generate perfectly executing, elegant, and maintainable source code, and put the emphasis on our libraries to "do the right thing" and to "do them right" with minimal instructions from the developer. All that is required from you is a small amount of truly simple source code.

But the appeal of a visual designer is not lost on Tools Plus. Tools Plus libraries and its optional framework leverage the power of your existing tools, specifically your resource editor, to enable you to visually design your application's user interface. You can always upgrade or replace your user interface design environment (your resource editor) and Tools Plus will make the best use of your design, thus giving you the best of both worlds. If you have a powerful resource editor like Resorcerer from Mathemæsthetics (see "Creating and Editing Resources" later in this chapter), you can attain near-WYSIWYG (What You See Is What You Get) visual designing capabilities.

Imagine being able to create a fully operation window and all its user interface elements with one line of code (LoadDialog(3, DialogID)), and being able to access any user interface element with an equally simple line of code (SelectButton(12,on)). Remember: as soon as you create it, it works! That's why Tools Plus libraries and *optional* framework, teamed with a resource editor, gives you an exceptional rapid-development arsenal.

Tools Plus User Manual Formats

The Tools Plus Pro, Tools Plus Academic and Tools Plus Lite software development kits (SDK) ship with two user manuals, both of which are identical in content, but different in format only:

- An electronic user manual in eDoc format
- An electronic user manual in PDF, or Portable Document Format (Adobe Acrobat format)

The Tools Plus Evaluation Kit, a free kit that lets you try Tools Plus before buying it, includes only an eDoc manual.

Unlike a printed user manual whose page numbers start at 1 at the first chapter, the page numbers in the electronic user manuals are numbered sequentially starting from the title page, which is identified as page one. This is done so that you can reference the table of contents and index, and simply “go to” a required page number.

eDoc for Simplicity and Speed

The eDoc electronic user manual is designed for instant gratification, ease of use, and speed. The advantages that this format presents are:

- You can view the manual by double clicking it. There is nothing to install first.
- You need much less disk space than with PDF (under 2 MB)
- Text searches are much faster than with PDF
- Displaying anything is faster than with PDF

With all these advantages, eDoc has a few disadvantages as well:

- If you print an eDoc manual, it does not look as good as PDF
- If a font that is required by the eDoc manual does not already exist on your system, a substitute font is used. This may deliver suboptimal viewing and printing results.
- Some parts of the eDoc manual have been optimized for viewing on a monitor as opposed to printing. Their lower resolution will not look as detailed when printed.
- eDocs can only be viewed on a Macintosh

PDF for Perfect Printing

The PDF electronic user manual is designed for “perfect” viewing and printing. The advantages that this format presents are:

- The document contains all required fonts, high resolution images, and line art for the best possible printing results.
- When viewed on a monitor, PDF files show all text as anti-aliased. Many people prefer this to standard text.
- PDF files can be viewed on Macintosh, Windows™, OS/2™ and Unix™ computers.
- All viewing detail is preserved, no matter how much you magnify the image.
- The PDF viewer has more sophisticated viewing and navigation services than eDoc.

PDF has several disadvantages as well:

- You must install viewing software on your hard disk before you can view or print a PDF document. All Tools Plus SDKs include Adobe Acrobat Reader which consumes several megabytes of disk space.
- Viewing and searching PDF files is much slower than eDoc
- Some people don’t like the anti-aliased text, preferring the crisper, standard Macintosh fonts.

All Tools Plus SDK CDs include Adobe Acrobat Reader, an application that lets you view and print PDF files. See the “Acrobat Reader” folder on your CD that includes instructions on how to install Acrobat Reader in a “Read Me” file. We recommend that you install the “Reader+Search” variant of Acrobat for the greatest versatility. If you already have an Acrobat Reader installed that is the same version or newer than the one on the Tools Plus CD, you do not need to install Acrobat Reader.

Printing the User Manual

If you purchase Tools Plus, you may print one (1) copy of the user manual for your own use. You may not print multiple copies, or allow multiple copies to be printed. Use the PDF version of the user manual for printing because it delivers the best visual results, and it does not require the installation of any special fonts.

Registered Developer Benefits Period

As part of your Tools Plus licensing fee, Water’s Edge Software provides the following products and services to you for one full year starting from your initial purchase, at no additional cost:

- Prompt, world-class technical support with no limit to the number of emails/calls
- Software updates (bug fixes and minor revisions)
- Software upgrades (major releases containing considerable new functionality and/or improvements to existing services and features)

- Access to the electronic Tools Plus Developer Forum where you can meet other Tools Plus developers and leverage their expertise and experiences.
- Access to the online Tools Plus Known Bug List (a detailed list of all known bugs confirmed to date, their status, work arounds, and what we are doing about them)
- Subscription to the Tools Plus Bug Alert Service. This service sends you an email as soon as a new bug is discovered and confirmed in Tools Plus libraries + framework. The email details the impact of the bug, work arounds, and what we are doing about it. This service is highly recommended for all developers!
- Subscription to Water's Edge Software's press releases, as well as internal communiqués that are intended *only* for Tools Plus licensees. This service keeps you informed about what we are doing and the projects that are being planned.

Your benefit period starts with your initial Tools Plus purchase, and continues for one full year. Software updates and upgrades include delivery to you at no additional cost. Our goal is to have at least two substantial releases per year. We automatically send you a reminder when it is time to renew your benefits period for an additional year. The reminder includes complete details about your renewal.

An Ordinary Application's Architecture (without Tools Plus)

Ordinary applications, those written without Tools Plus, have an architecture that can be represented by the model shown in figure 1 on the following page. All of Macintosh's capabilities can be accessed through the routines in the Macintosh's built-in *toolbox* and through the data structures that are created and maintained by those routines. An ordinary application creates its interface and makes it work by using the toolbox's routines.

Numerous complexities arise because the application must also continuously manage the relationships between user interface elements, and between elements and their environment, again by using toolbox routines. The Event Manager, for example, can effect TextEdit, the Menu Manager, Window Manager, Dialog Manager, Control Manager, List Manager, and others.

To further complicate matters, parts of the toolbox are available only on certain Macintosh models such as Color QuickDraw that is available only on the Macintosh II series and newer computers. Similarly, parts of the toolbox are available only on certain system *versions*, such as System 7's pop-up menu CDEF (Control DEFinition) and System 7.5's floating palette WDEF (Window DEFinition). Even identical Macintosh models running the same system version can be quite different to the application's world just by attaching a second monitor to one of the Macs. All these variations mean the programmer must do things differently depending on the Macintosh model, the system version and computer configuration that is running his application, or he must account for all the possibilities if the application might run on a variety of Macintosh models, system versions or configurations.

The final aspect of an ordinary application, although this is not readily apparent in the model in figure 1, is that the entire Macintosh toolbox gives the application access to the Macintosh's capabilities on a *low level*, meaning that even simple things like creating a color button can take dozens of lines of code, and making it work can take dozens or hundreds of lines more. Complex tasks like make a floating palette work and function properly with all other user interface elements can require over a thousand lines of source code.

Many features found in today's popular Macintosh applications don't even exist in the toolbox. Although System 7.5 was the first to include a floating palette WDEF (Window DEFinition resource that gives your application the *look* of a floating palette), there is nothing in the System 7.5 toolbox that makes the window *behave* like a floating palette. In an ordinary application, the programmer must accomplish this himself using toolbox routines. The same applies to a tool bar, picture buttons, and an editing field with scroll bars attached to it. When working with the toolbox, all work involves low level coding, and low level coding means the developer must assemble a number of toolbox routines to create just about anything that is useful.

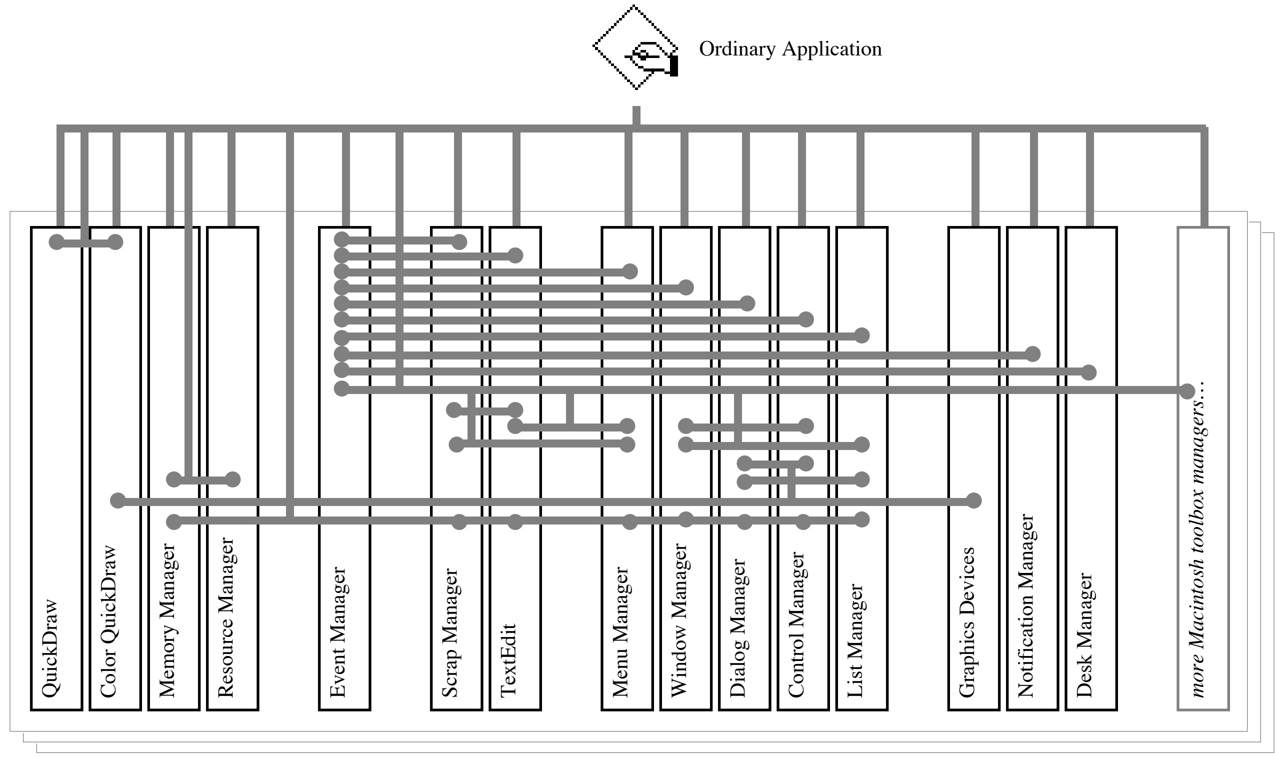


Figure 1 -- An ordinary application's architecture without Tools Plus. The application must interact correctly with the necessary toolbox managers to create the application's user interface and to make it work. The application must also manage the relationships *between* the toolbox's various managers.

To summarize some characteristics that are common in an ordinary application, that being one written without Tools Plus:

- The developer uses the toolbox's routines to create his application's interface.
- The developer typically writes multiple lines of source code that access numerous toolbox routines to accomplish even simple tasks, like creating a color button (i.e., create something usable by assembling a number of toolbox routines and writing the necessary support code to make them work right).
- The developer uses the toolbox's routines to respond to low level Event Manager events, and writes source code to decode the event data, determine its meaning, and apply events to the various parts of the interface, again by using more toolbox routines.
- The developer writes code to maintain the relationships between various user interface elements and to account for all user interaction and machine conditions.
- The developer accounts for variations between Macintosh models, variations between system versions, and variations in Macintosh configurations within his application's code and his selective use of toolbox routines.

Various developer tools are available to address some of these chores, and we'll compare them to Tools Plus later in this section.

A Tools Plus Application's Architecture

Applications written with Tools Plus can be represented by the model shown below in figure 2. The shaded area at the bottom of the model represents the Macintosh toolbox, just like in applications that don't use Tools Plus (as seen in figure 1). The striking difference is that your application no longer has to deal with the toolbox directly. Instead, Tools Plus provides a relatively small number of routines (just a few hundred) that are logically organized and designed to do immediately usable things. As a result, you can create *working* user interface elements with a single line of code, and those user elements automatically behave like they should regardless of the Macintosh model, system version, or machine configuration that runs your application.

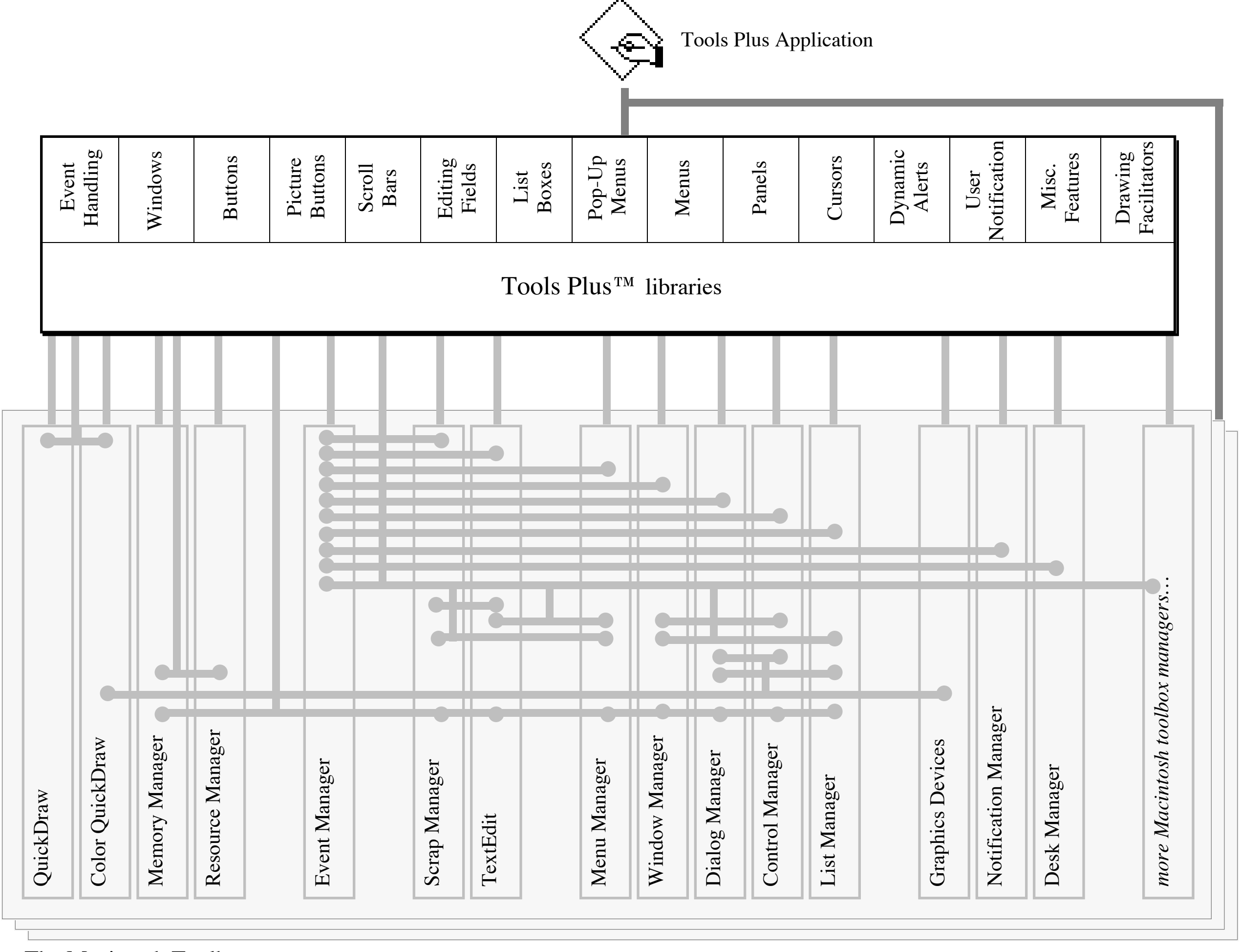


Figure 2 -- An application's architecture when using Tools Plus. You create ready-to-use user interface elements and Tools Plus makes them work. This results in a dramatic reduction in effort, source code volume, and code complexity.

To summarize some characteristics of an application that is written with Tools Plus:

- The developer uses a single Tools Plus routine to create virtually any element of his application's interface.
- Tools Plus makes user interface elements work, and automatically manages relationships between elements.
- Tools Plus virtually eliminates event management by applying events correctly to the various elements.
- Tools Plus accounts for variances in Macintosh models, system versions, and computer configurations.
- The developer experiences an approximate 80% to 95% reduction in development effort and source code volume.
- Applications run quickly and requires little memory or disk space.

A developer who uses Tools Plus libraries while writing an application can focus his attention on *creating* his application instead of tending to the mechanics involved in creating elements, making them work, establishing relationships between elements, complex event processing, and variations between Macintosh models, system versions and configurations. The result is a much more effective developer or development team.

The model in figure 2 also illustrates that Tools Plus does not prevent you from using the Macintosh's toolbox if you choose to do so. You may decide that you want to add a new feature that does not exist in Tools Plus, or you may need to access low level information. In these cases, you can use the Macintosh's toolbox routines as you normally do. When using Tools Plus libraries, you can write your application using either Object Oriented Programming (OOP) or procedural (traditional) coding techniques.

Powerful Features Using Simpler, Higher-Level Coding

Tools Plus doesn't just make programming easier by making a developer more efficient. It simplifies the Macintosh programming experience by letting you program at a consistently higher level throughout your development, testing and support cycle. Programming is substantialy simpler because Tools Plus's routines are ready to use, unlike the routines found in the Macintosh toolbox that need to be organized into usable tasks or operations. All you need is a single Tools Plus routine to create a user interface element, establish its relationships with other user elements, and make it work.

A simple example of this is the Apple menu. With Tools Plus, you can give your application full reign over desk accessories just by creating the Apple menu with the AppleMenu routine (one line of code). Tools Plus automatically takes care of all activities pertaining to desk accessories, such as when the user selects a desk accessory from the Apple menu, repositions the accessory by dragging it, clicks buttons or types in the accessory, copies and pastes text in the desk accessory, and eventually closes it. All this is handled automatically by Tools Plus meaning you don't have to write any code for it. Tools Plus also accounts for your application running on System 6's Finder and MultiFinder as well as the full-time MultiFinder that runs on System 7.0 and later. All you need to know is that when you create the Apple menu, it will work perfectly in all situations.

Another example of an immediately usable operation is Tools Plus's LoadMenuBar routine: it loads an 'MBAR' resource, reads, loads and installs all the pull-down menus itemized in the 'MBAR' resource including an Apple menu, a fully functioning Edit menu, and all other pull-down menus. It also loads, installs and attaches all hierarchical menus that are attached to the pull-down menus. All this is accomplished with a single line of code. The best part is that you don't need to write any code to make the menus work, including the Edit menu's Undo/Redo, Cut, Copy, Paste, Clear and Select All items. They all work correctly as soon as they are created.

Tools Plus's features integrate with each other according to the User Interface guidelines found in Inside Macintosh. An active editing field, for example, automatically intercepts and processes key-strokes from the keyboard. An editing field's text can also be affected by the Edit menu, which lets the user Cut, Copy, Paste, and Clear the text, as well as transfer text between your application and other applications and desk accessories via the clipboard. The Edit menu is automatically updated by the user's actions in the active editing field: when an insertion point is in the field, the Edit menu's "Cut" and "Copy" commands are disabled, since no text is selected for cutting or copying. These relationships between the Edit menu, and editing field and the user are all automatic in Tools Plus.

Tools Plus supports and automates all standard user interface elements and seamlessly integrates support for popular extras like floating palettes, a tool bar, picture buttons, tabs, sliders, a complete "3D look" and much more. It also includes a wealth of features that are sought after by developers, such as cursor animation, zoom lines like those found in the Finder, hiding the menu bar and/or the Finder and other applications, and automatic reconciliation with Macs that lack Color QuickDraw. In almost all cases, a Tools Plus feature is implemented using one line of simple code.

Tools Plus lets you say "run" using one word instead of painfully describing *how* to run using several carefully worded and precise paragraphs. You can see how using Tools Plus frees you, the programmer, to do more important things like writing applications instead of trying to make all the pieces work!

Event Processing is Virtually Eliminated

Tools Plus practically eliminates the need for event processing code in your application. An entire chapter called "Event Management" is dedicated to this subject in this user manual, so only a brief overview is presented here. Unlike traditional Macintosh applications that use the toolbox's GetNextEvent or WaitNextEvent routines to get a low level event which then must be manually processed, a Tools Plus simply calls an "event handler" routine (which you write) whenever an event is available. Events are generated by user activity such as typing or mouse-clicks, and by system activity like refreshing a window, or inserting a floppy disk. You can write a single event handler routine for your entire application, and optionally write event handler routines that handle events for specific windows.

The big difference between using Tools Plus and the toolbox is that Tools Plus does everything it possibly can before informing your application of an event. Many events are processed internally and are never reported to your application, such as when the user types in an editing field. This is because the field automatically processes the typing and there is no need to tell your application about it. Other events *are* reported to your application, such as when the user selects a button or clicks a window's close box.

Tools Plus also translates the Macintosh's events into something your application can use right away, so instead of getting a generic, low level "mouse down" event, your application gets a highly informative and very specific Tools Plus event such as: the "Save" button was selected in the "Add Customer" window (or button 4 was selected in window 15).

The following example illustrates the difference between an ordinary C or Pascal program, and a program that is written with Tools Plus. The left column represents the steps a traditional program has to take to detect and process a very simple event, whereas the right column has the benefit of Tools Plus. Note that the left column is *highly* simplified!

Ordinary C or Pascal Programming		Programming with Tools Plus	
1	Get an event	1	Tools Plus calls your event handler routine
2	Determine the type of event (a "mouse-down")	=	Button was selected (button 3 in window 15)
3	Determine its location (a window's content region)		
4	Determine where in the window (a "control")		
5	Track the mouse in and out of the control		
6	If the mouse button was released inside the control's region, report that the control was selected. Otherwise ignore the entire event.		
=	Mouse-down event in a control (a control *handle* is known) in a window (a window *pointer* is known)		

Of course, additional steps could be taken in traditional C, C++ or Pascal to obtain a window number and button number, but this is possible at the expense of more programming and added complexity in your source code.

Tools Plus gets an event from the toolbox's event manager, automatically applies it correctly to your application's user interface, and reports a highly informative and specific Tools Plus event to your application only if your application needs to be informed of something. For developers with advanced event processing needs, Tools Plus optionally lets your application get an event directly from the toolbox's Event Manager, inspect the event record, possibly alter it or discard it, and even sythesize an event (i.e., a fake "mouse down"), then pass that event to Tools Plus for processing. This gives you everything from fully automatic event processing to manual event processing capabilities.

Apple Event Support is Built into Tools Plus

An application that is written with Tools Plus does not need to be Apple Event aware, but we strongly recommend that you make it so, especially if your application will run on Mac OS 8.5 or later. Tools Plus automatically supports all four required Apple Events: "open application", "open documents", "print documents", and "quit application". See the Event Management chapter's section named "Automatic Apple Event Support" for details.

Supports Resource-Based and Dynamic Interface Design

Tools Plus lets you define your application's user interface in several ways. In all cases, Tools Plus dramatically facilitates development by making user interface elements work as soon as they are created, and by providing powerful routines that make it easy for your application to interact with those elements.

Dynamic Interface: As the name suggests, your application can create user interface elements completely under its own control without you having to define resources. Virtually any element of your application's user interface can be created with a single line of code. Creating your application's interface dynamically has the following advantages:

- The application's interface can be created based on external data. Your could, for example, write a system that lets you store user interface definitions in a database then create those interface elements as required.
- New Macintosh developers may find it daunting to learn a resource editor and to create an application along a dual path by defining resources and writing code to call those resources.
- Your application's source code has a complete definition of its user interface, so you don't have to refer to a collection of resources for this information.
- Your application has an additional degree of security to protect it from power users who are familiar with a resource editor. By having the user interface defined in your application's code instead of resources, power users are less capable of altering the application.
- Reduced interaction with slow media like a floppy disk or a CD-ROM may increase performance.
- Some developers may prefer this approach over resource-based development.

Resource-Based Interface: You define elements of your user application's interface by creating resources using a resource editor such as Apple's ResEdit. These resources can include menus ('MENU' resource), a menu bar ('MBAR' resource), dialogs ('DLOG' and 'DITL' resource defining a window and its user interface elements), a single control ('CNTL' resource), and other elements. Resource-based interface design has the following advantages:

- The user interface can be designed visually using an inexpensive resource editor such as Apple's ResEdit
- The user interface definition can be separated from your application's source code to allow the interface to be changed without having to recompile your application
- It facilitates localization
- You can apply custom colors to windows and controls without writing any code
- Typically reduces the amount of source code you need to accomplish a task
- It can save memory

Tools Plus routines completely replace the need to use the toolbox's Dialog Manager thereby letting you avoid the numerous complexities and short-comings that are typically encountered when trying to make your application's user interface work and behave like a Macintosh should.

Combination: Sometimes it may be appropriate to create parts of your user interface dynamically while defining other parts with resources. Tools Plus makes this easy because you use the same routines to interact with user interface elements that are created dynamically as those that are created using resources. This differs significantly from other systems in general, and from the Macintosh toolbox specifically, both of which require that you program in a different manner depending on whether you define your interface by using resources or not.

Accessing Tools Plus Libraries

Tools Plus arrives as a set of compiled 680x0 libraries or as a single compiled Power Macintosh library. It also includes a C/C++ header file or a Pascal interface file. You compile the Tools Plus libraries into your application, a process that takes just seconds, then you access to the routines in the libraries by using the C/C++ header file or Pascal interface file. As far as your application is concerned, Tools Plus routines can be seen as a replacement for many, many toolbox routines.

Creating New Applications With Tools Plus

If you are starting a new application using Tools Plus and you are already familiar with the Macintosh toolbox, then using Tools Plus will feel very familiar. However, you will immediately notice that very little code is required to create your user interface and make it work, and that your event management code almost disappears.

As with all projects, alway devote considerable energy to designing your system before you even think of coding it. Include functional definitions and detailed designs on paper for all windows and menus. Clearly define how all user interface elements work and how they relate to your system's functionality. This up-front detailed design work will save countless hours in modifications and corrections later on in your development cycle.

If it is practical for you to do so, define all your menus and dialogs as resources using a resource editor. A powerful resource editor like Resorcerer lets you visually define complex dialogs with settings for font, font size, style and item color information. When creating dialogs, define "non-standard" user interface elements, ones that the ordinary Dialog Manager does not know about like Tools Plus's picture buttons and panels, as "user items." Later, when your application opens the dialog and creates its contents using a single line of code, your application can read those user item co-ordinates and create the user interface elements using Tools Plus routines. Tools Plus lets you bring those resources to life with a single line of code.

If it is not practical to define windows, dialogs and menus using resources, Tools Plus also provides powerful routines to let you easily accomplish the same things completely under your application's control. In the case of dialogs, which are simply windows containing user interface elements, you will need to use a system that lets you draw the interface to scale and obtain object co-ordinates. Here are some popular solutions:

1 Use a resource editor to design your dialogs. When you select an item, the resource editor can provide the object's co-ordinates.
2 Draw your interface using a drawing application like MacDraw. Don't use a painting package like MacPaint unless it lets you keeps objects as separate elements, even when they overlap. Drawing applications typically have an option that lets you display object co-ordinates in pixels.
3 The least expensive solution, although also the least versitile, is to draw your interface on graph paper.

To translate your interface design to a working interface, use Tools Plus routines to create each element in the interface. These routines require co-ordinates for each element, and you can provide those co-ordinates from your design. Run your application and take a screen capture (⌘-Shift-3) of your dialog and print it. If you used a drawing application to design your interface, display the original drawing on your screen and get a screen capture and print it. Take the two screen captures, your original design drawn to scale and your actual application, overlay them, and put them up to a window. You will immediately see where any elements are out by a pixel or two. Write down all the corrections (i.e., shift this edge down two pixels) on your printed copy, then go back to your source code and alter your co-ordinates by that amount.

The tutorials and demo application that are included with Tools Plus provide examples for creating and using just about any user interface element.

Using Tools Plus in an Existing Application

Some developers may be put off by the prospect of modifying their existing applications to take advantage of Tools Plus's wealth of features, professional appearance, and boost in developer effectiveness. Developers who have updated fairly large applications to use Tools Plus were able to realize these advantages in only a few days, and in doing so, they removed hundreds or thousands of lines of low-level code and they simplified their applications.

Tools Plus can use your existing resources, thus retaining your original interface design. The following steps update an ordinary application to incorporate Tools Plus:

1 Use Tools Plus routines to create your menus and/or to use your menu resources (likely one line of code). See the Menu chapter for details.
2 Use Tools Plus routines to create your windows and/or to use your dialog resources (likely one line of code per window or dialog). See the Windows chapter for details.
3 Remove code that gets an event from the toolbox (GetNextEvent or WaitNextEvent routines).
4 Alter your event loop such that it is enclosed within a routine, thus becoming your event handler routine. See the Event Management chapter for details on how to write an event handler routine. It's nearly identical to traditional event loop, just simpler.

5 Update your event handler routine to respond to Tools Plus's highly informative and specific events instead of low-level events from the Event Manager. The Event Management chapter details this.
6 Use powerful Tools Plus routines to interact with your user interface elements instead of using toolbox routines.

In each step you will remove reams of complex and obsolete source code that used to interact with the toolbox and its variety of managers. You will replace that code with very few lines using Tools Plus routines. In most cases, many lines of conventional code will be replaced with a single Tools Plus routine.

Resist the temptation to add new features as you initially update your application to use Tools Plus. Your first objective is to make your application work as it has before, but with the advantages of simpler and less voluminous source code. This is a typical trait of applications that use Tools Plus. When your application is working using Tools Plus routines, you can then easily start adding Tools Plus features to your application to give it new capabilities and a completely professional appearance.

Range Checking

Tools Plus has built-in parameter range checking that is always active, even when you turn off your compiler's range checking option. This feature is available to both C and Pascal programmers. Although this feature adds about 1K of code and requires a *very slight* amount of processing time, the benefits are well worth it when you consider the time you will save during application development.

If your application passes a parameter to a Tools Plus routine with a value that is out of the required range, an alert is displayed stating:

Error: Parameter passed to a Tools Plus routine is not within the legal range of values.

When this occurs, the offended Tools Plus routine is not executed. Instead, an alert is displayed with the above message and a "Continue" button. Your application resumes executing, bypassing the offended routine, when the user clicks the Continue button.

To facilitate debugging, you can install your own action routine that will be called instead of displaying the parameter range alert. If you have a stop point in this routine, you can step out of the routine line by line and eventually return to the offended Tools Plus routine. See the SetParamRangeErrProc routine for details on how to install your own action routine.

The Tools Plus Advantage

There are many advantages to using Tools Plus when you are writing a Macintosh application:

Easy to learn…
You'll be creating professional quality applications soon after you open Tools Plus. With other tools, you may well end up spending months learning complex class libraries or how to use a development environment that is less than intuitive.

Flexible…
Tools Plus does not demand that you adhere to a rigid framework or to cumbersome design constraints. It fits in with your programming style, whether your code is procedural or object-oriented. Code generators and other application frameworks impose their style and inherent restrictions upon you, forcing you to learn a new way of programming and to adopt someone else's style and preferences.

Simpler…
Programming with Tools Plus is a simplified experience throughout your application's development, testing and maintenance cycle. Application generators are great for quickly creating the "first cut" of your application. After that's done, they make you resort to low level coding throughout the remainder of your development, testing and maintenance cycle. Your initial gains can easily be negated over the remainder of your project. Tools Plus virtually eliminates low-level coding drudgery.

Reduces code…
A single Tools Plus routine is often equivalent to hundreds of lines of conventional code. Tools Plus lets you eliminate (or never create) thousands of lines of source. Less code results in fewer bugs and much simpler, maintainable source code. Class libraries can easily add 50,000 lines of code to your application.

Expandable…
You can easily add new elements to your user interface after you create your program. Typically, all you need is one line of code to create the item and a case for it in your event handler routine. Tools Plus takes care of making the new element work correctly and manages its relationship with other user interface elements.

Non-obstructive…
Tools Plus is designed to handle the vast majority of your application's user interface and event-related work. You can easily add functionality that is beyond the scope of Tools Plus libraries, such as support for QuickTime and other technologies that have not been invented yet.

Compiles quickly…
Tools Plus is made up of a small number of libraries and files and takes just seconds to compile into your application. Compare that with code generators that create dozens of source files that are dependent on dozens (sometimes hundreds) of other files. It takes just a few seconds to compile Tools Plus libraries into your application instead of ten minutes or more with large class libraries or a complex framework.

Compact and fast running…
Tools Plus libraries require little memory or disk space. This helps you create applications that are compact and efficient. You also get lightning performance that you would expect from hand-optimized code. These benefits are passed onto your projects to help you produce commercial quality applications.

System Independent…
Tools Plus routines work seamlessly with System 5 and System 6 (when running under Finder and MultiFinder), and System 7 or higher. They also run on both 680x0 series and PowerPC processors in emulation and native mode. They sense the presence or absence of Color QuickDraw and automatically account for the differences. They require no special consideration for math co-processors. It is easy to write applications that are backward compatible with older systems while giving them powerful features that are typically available only in newer ones.

Consistent…
Tools Plus for PowerPC is identical to Tools Plus for 680x0 processors, thereby easing your transition to Power Macintosh. The C/C++ header and Pascal interface are nearly identical helping you make the transition from Pascal to C/C++ if you should choose to do so.

Portable and reusable…
You can move code between applications more easily because Tools Plus resolves the toolbox's complexity within the our libraries instead of your code. A single Tools Plus routine can be equated to the complexity and interdependencies existing in dozens of files in a class library, or thousands of lines of conventional code.

Safe…
Tools Plus routines are *safer* to use than Macintosh toolbox routines because they shield you from potential pointer and handle dereferencing problems and from numerous logical errors. Your application accesses GUI elements using numbers instead of pointers and handles (i.e., button 5 on window 18). All routines can be used on any model Macintosh running on any system, so you don't have to make special allowances if the Mac running your application has multiple monitors or doesn't have Color QuickDraw.

Self-managed interdependencies…
Every element of Tools Plus is aware of all the other elements. They all work together harmoniously so when you add a new user interface element to your application, it works as soon as it is created. This lets you concentrate on writing your application instead of trying to get all the pieces to work, and to work properly with each other.

Event processing that makes sense…
The revolutionary Event Translator in Tools Plus reports usable occurrences in a comprehensive, immediately accessible record. An example of this is telling your application "the Cancel button in the Search dialog was selected." This is much simpler than decoding event messages and tracking controls, handles and pointers, as required when dealing with the toolbox directly. Tools Plus simply calls your event handler routine and tells it everything it needs to know.

Depth versus scope…
Large class libraries often address many aspects of Macintosh's abilities. By comparison, Tools Plus focuses firmly on the user interface and event processing. We offer a great depth in this area instead of having a broad scope that may not handle any one thing extraordinarily well. Our approach lets us deliver user interface features that are unparalleled by competing products. Our picture buttons are a prime example of power melded with simplicity.

Broadens your horizons…
Tools Plus lets you easily incorporate many aspects of the Macintosh's impeccable user interface that you may otherwise have excluded due to their complexity. You'll find the unwieldy becomes possible, typically with a single line of code.

Ever expanding…
Tools Plus libraries are constantly being expanded, enhanced and optimized based largely on our users' requests, so you benefit from innovations that are happening in a community of developers around the world. And even though Tools Plus is always giving you more features and ease of use, it is always lean and fast running.

Economical…
There are no runtime costs for Tools Plus. That means registered users can distribute an unlimited number of copies of an unlimited number of applications they have written with Tools Plus without having to pay additional fees or royalties. We also offer free updates and significant discounts on major upgrades.

Well supported…
Tools Plus starts with the best support there is: by delivering a product of outstanding quality. Our user manual is frequently praised by our customers, and our technical support staff can quickly assist you wherever you may be located.

Who can benefit from Tools Plus

Just about anybody writing an application on the Macintosh in C, C++, or Pascal can benefit from using Tools Plus. It is useful to different people for different reasons:

- *Novice programmers* can start developing applications more readily and with greater confidence. The task of programming is simplified to produce quicker results with fewer bugs. And you don't have to learn a resource editor (such as Apple's ResEdit) before you start using Tools Plus.
- *Seasoned programmers* can use Tools Plus to develop an application in less time and with fewer bugs.

What kind of applications can be written with Tools Plus

Tools Plus does not limit you to writing certain *kinds* of applications, in that it does not preclude you from exercising your technical or creative skills on the Macintosh. It merely helps simplify and manage the user interface and event processing that is so prevalent in Macintosh programs. Varying programming needs are addressed by Tools Plus:

- *Quick and Dirty* applications can be written in less time. These programs can have all the features of finished Macintosh applications, which makes them easier to work with.
- *Full fledged applications* that are suitable for shrink-wrapping can be created using Tools Plus.
- Many kinds of plug-ins (available only in Tools Plus for CodeWarrior).
- Just about any program can be written more quickly and with less effort by using Tools Plus.

What is Tools Plus not suitable for

Tools Plus is intended for *application* developers. It was designed to be the event processing and translating engine within an application, so Tools Plus definitely cannot be used to create any of the following:

- Drivers, control panels, system extensions (INITs)
- Some external code modules*
- CFM68K projects
- Desk accessories (you can still use Tools Plus to write an application that looks and feels like a desk accessory while running under System 7 or later)
- It is absolutely useless for writing "faceless" applications. Why use Tools Plus if your app has no user interface at all?

* For technical developers who want to know *why* this is so, or for those astute individuals who are inclined to amaze us with what they can accomplish with Tools Plus, here are some details:

- Normally, Tools Plus gets an event from the Event Manager by calling GetNextEvent or WaitNextEvent, then processes it automatically. If you have an unusual requirement where you need to get an event directly from the Event Manager, inspect it, possibly discard it, or even apply it elsewhere without letting Tools Plus process it, you can do so. You can also call the Event Manager directly. In either case, you can pass that event to Tools Plus for processing.
- Tools Plus wants to know about all windows and menus in your application. In the case of plug-ins, you must create your plug-in's windows using Tools Plus routines, and you cannot create menus since the host application has already created them. What you put inside your plug-in's windows is completely up to you, and Tools Plus has plenty of routines to make it really easy.
- Some applications interfere with the Macintosh's natural processes. ACI's 4th Dimension, for example, makes modal windows such as the dBoxProc window behave modelessly, as can be demonstrated by creating a dBoxProc window using the toolbox's NewCWindow routine, then polling for events with the toolbox's WaitNextEvent routine. If the host application "breaks" part of the toolbox, then Tools Plus may not behave correctly as a plug-in within that host application.

Tools Plus libraries *must* be compiled into your application and cannot reside in a shared library. This requirement is for licensing purposes only to prevent other unlicensed developers from accessing Tools Plus libraries.

System Requirements

Computer

Tools Plus makes extensive use of ROM routines that are found only in the 128k ROMs (version 117) or higher. Applications written with Tools Plus will run on the Macintosh 512KE (the "E" stands for *enhanced* with the new ROMs) or higher. They will not run on a Lisa (also called a Macintosh XL), Macintosh 128K or a standard Macintosh 512K computer. These applications will also run on Power Macintoshes.

Compiler

Tools Plus can be used by programmers developing with:

CodeWarrior C/C++ (68K or PPC) from the CW6 CD or later, including CodeWarrior Pro
CodeWarrior Pascal (68K or PPC) from the CW6 CD or later, including CodeWarrior Pro
THINK C/C++ 5.0.4 or later (68K)
THINK C/C++ 6.0.1 or later (68K)
THINK C/C++ 7.0 or later (68K)
Symantec C/C++ 8.0.5 or later (68K, Symantec Project Manager, also called SPM)
Symantec C/C++ 8.6 or later (PPC, Symantec Project Manager, also called SPM)
THINK Pascal 4.0.2 or later (68K)

Note that Metrowerks ships a "Discover Programming" kit that includes a CodeWarrior IDE (Integrated Development Environment) and instructional material. Tools Plus supports this kit. It is a cost-effective alternative for those who cannot afford the full CodeWarrior IDE.

Development Tools

Every Macintosh developer should have a copy of ResEdit, Apple's resource editor application. It is easy to use, indispensable, and best of all, free from your Apple dealer. Always get the latest version (2.1.3 at this writing), since it is always being upgraded and improved. Beginners can get by without ResEdit, but you won't want to.

Power users will welcome a powerful resource editor like Resorcerer from Mathemæsthetics. It lets you fully realize the potential benefits of resource-based programming by letting you set color and styles for dialog items.

System

Applications created with Tools Plus can run under System 5, System 6 and System 7 or higher. We recommend you use at least System 6 (optimally 6.0.8) or System 7 because Apple has fixed various bugs and many features have been added to the newer versions of the system.

While *developing* applications using Tools Plus, you will still have to observe the requirements and limitations of your development environment. THINK C 5.0 requires System 5.0 or higher. THINK Pascal 4.0 requires System 6.0.5 or higher. CodeWarrior requires System 7 or higher. Consult your compiler's User Manual for details.

Memory

Applications created with Tools Plus can run with less than 200k, depending on the size and complexity of your application. The overhead associated with Tools Plus is typically about 150k for 680x0 libraries and slightly higher for PowerPC native libraries.

While *developing* applications using Tools Plus, you will still have to observe the requirements and limitations of your development environment. THINK C 5.0 and THINK Pascal 4.0 both require 2MB of RAM when using System 7. CodeWarrior requires 8MB. Consult your compiler's User Manual for details.

Disk Space

Tools Plus libraries and related files require only a few hundred K of disk space. When compiled, Tools Plus libraries add about 100k to 150k to your 680x0 application's size, and slightly more to your PPC application's size (add both 680x0 and PPC for fat binary applications).

While *developing* applications using Tools Plus, you will still have to observe the requirements and limitations of your development environment. THINK C 5.0 and THINK Pascal 4.0 both require about 2MB of disk space.

Finished Applications

Please be aware that you, as a programmer, have the capacity to write applications that have requirements far in excess of Tools Plus's minimum system requirements. It is possible that you may choose to write a program that requires a PowerPC processor, Mac 0S 8.5.1 or later, 300 megabytes of RAM, and a pair of 20 inch color monitors. Be aware that your finished application will likely have needs that exceed Tools Plus's minimum requirements.

Tools Plus Performance

Tools Plus libraries are hand-tuned, lightning-quick performers. Magazine reviews have praised Tools Plus for being smaller and faster than any competitor, but there are several things that can slow Tools Plus down:

- THINK Pascal development environment on a PowerPC: When running the THINK Pascal development environment on a Power Macintosh, THINK Pascal takes a while to load up all of Tools Plus's libraries into memory. This issue shows up on any sizable application, and not just those using Tools Plus. On a 100 MHz PowerPC 603, the time between hitting the "Run" command and seeing your first window can be around 40 seconds. You can reduce this time by getting Connectix SpeedDoubler which accelerates 680x0 code running on a PowerPC, including THINK Pascal, especially during the lengthy startup process. Alternatively, you can get a fast Power Macintosh such as any G3, the slowest of which reduces the startup time to 5 seconds. Note that this delay exists *only* in the development environment, and not in compiled applications.
- Symantec C++ 8.6 compiling PowerPC-native Tools Plus libraries: Symantec never overcame some inherent design shortcomings that would allow to recompile Tools Plus source code to a native Symantec format. Instead, they released a Metrowerks to Symantec library translator which isn't too quick. You will notice slow performance during compilation the first time you compile an application or plug-in with Tools Plus libraries. Subsequent recompiles will be quicker. The final application performs quickly, just like you'd expect.

- Background processing: If your application needs better performance during background processing (when receiving null events), see the SetNullTime routine for more information on this. Also note that windows with animated Appearance Manager controls, such as the “chasing arrows,” clock control, active editing field, or the busy “barber pole” progress indicator, will slow down background processing.

Special Considerations with Mac OS Versions

Over the years, Apple has introduced issues into the Mac OS that may affect your application or plug-in. It is important for you to realize that these issues plague *all* Macintosh developers to some degree, and not just those who use Tools Plus libraries + framework. As at the time of this publication, the following issues were still noted:

- Appearance Manager versions 1.0 through 1.0.1, shipped before Mac OS 8.5, contain deficiencies that do not permit you to create some controls correctly, such as tabs and window headers. You should upgrade to Appearance Manager 1.0.3 or later on System 7 or Mac OS 8 prior to 8.5. Mac OS 8.5 integrates the Appearance Manager into the operating system, so you need not upgrade it.
- The Appearance Manager, when running under System 7.x, exhibits cosmetic issues such as menus lists whose gray background turns to white when the mouse is moved off a selected item.
- Mac OS 8.5 has severe stability issues. You should upgrade to Mac OS 8.5.1 or later.
- In Mac OS 8.5 and later, the OS becomes unstable when applications running *within* the THINK Pascal development environment quit or are reset. This instability will cause a crash or hang later. This does not affect double-clickable applications that are created by THINK Pascal.
- Starting with Mac OS 8.5, the Memory Manager behaves somewhat differently than in prior system versions. The implications are far reaching: bugs in your source code that “ran perfectly” in prior system versions may now cause problems, or vise versa. Use a memory testing application like QC (detailed later) to test your application on an older system version than Mac OS 8.5, if possible.
- At the time of this publication, memory stress testing tools (QC, detailed later) confirmed that Tools Plus libraries were “clean” when running under Mac OS 8.1 or older. QC forces all memory errors that may potentially occur, to be triggered during testing. These tests confirm that Tools Plus source code is not performing illegal operations. These testing tools indicate that Mac OS 8.5 (and possibly later versions) are performing invalid operations that may show up as crashes, hangs, or other anomalies. It is imperative to realize that these issues are not related to Tools Plus.
- Mac OS 8.5 introduced user-changeable *themes* that change the appearance of the user interface (menus, windows and controls all adopt a different look ranging from Apple’s standard Platinum gray-scale theme, to the outrageous “Gizmo” theme). Although beta versions of themes were distributed only as part of the developer program, Apple did not release final versions of themes with Mac OS 8.5. These beta themes did, however, make their way onto the Internet and are being used by many users. Note that some of these beta themes, and possibly subsequently released themes, may have bugs that can cause problems in your applications. The Drawing Board theme, for example, causes Mac OS to become unstable when you change the keyboard focus between two list box controls. Other such bugs may be lurking in themes. Realize this when you create your application or plug-in, and make sure you document it for your users. If your application causes hangs or crashes, test the application on Apple’s default Platinum theme. It is a sure sign of a faulty theme if the problem disappears when your application runs on another theme.
- At the time of this publication, inconsistencies were observed between themes. For example, the indicator on a vertical slider moves up as its value increases, unlike a scroll bar. This is true in all themes except Drawing Board. Beware that other inconsistencies may exist in themes. This is the fault of the themes, and not of the applications and plug-ins that are using them.

Assumptions made when this manual was written

Several assumptions have been made when writing this manual:

- You are already a C, C++ or Pascal programmer, or are learning to be one on your own. This manual does not teach you how to program in C/C++ or Pascal, nor does it teach you how to use the Macintosh's toolbox routines. It teaches you how to use Tools Plus.
- This manual makes no attempt to teach you how to use THINK C/C++, THINK Pascal, CodeWarrior C/C++ or CodeWarrior Pascal. Please consult your User Manual in such matters, or contact the manufacturer of your compiler.
- This manual makes no attempt to teach you how to use ResEdit, or any other application or tool that is part of your development environment.
- You already know how to use the Macintosh, and are familiar with its terms such as clicking, dragging, selecting, etc.
- This manual does *not* assume that all programmers and users are male. For the sake of easier reading and to avoid awkward genderless grammar, the term "he" implies either gender within this manual.

Conventions used throughout this manual

1) Whenever examples of source code are provided, they appear in a different font as illustrated below.

```
pascal void WindowClose (short Window);
```

2) Any information that is exclusive to either C/C++ or Pascal is marked with the following symbols:

   (C) Information following this symbol is exclusive to C/C++

   (Pascal) Information following this symbol is exclusive to Pascal

3) Each Tools Plus routine is documented with both a C/C++ header and Pascal interface.

4) Source code examples are given in Pascal, just like Inside Macintosh. In situations where the C/C++ source code differs (usually due to differences in record structure), an example is also provided in C.

5) The Pascal terms *function* and *procedure* are used throughout this manual. For the benefit of C programmers, a function is a routine that returns a value. The following table provides an example:

	C	Pascal
Procedure	`pascal void DeleteListBox(short ListBox);`	`procedure DeleteListBox(ListBox:INTEGER);`
Function	`pascal short FirstWindowNumber(void);`	`function FirstWindowNumber: integer;`

6) Important information is highlighted with *notes* and *warnings*.

   ✍ **Note**: A note that may be interesting or useful (don't skip these)

   ☞ **Warning**: A point you need to be cautious about.

7) This manual is not intended to replace Inside Macintosh or its equivalent. It does not detail the fundamentals of Macintosh programming, such as screen versus window co-ordinate systems (local versus global), commonly used structures (points, rectangles, grafPort, etc.), using the Macintosh toolbox, etc.

8) For the benefit of new programmers, portions of this manual address the basics of building a Macintosh application, such as some differences between System 6's Finder and MultiFinder and how to complete a double-clickable application.

Software Updates

Whenever you receive an updated version of Tools Plus, replace your existing Tools Plus files with those supplied by the update. This includes all the library files (*ToolsPlus.Lib*x), *ToolsPlus.h* header and *ToolsPlus.c* source files if you are a C/C++ programmer, and *ToolsPlus.p* interface file if you are a Pascal programmer. If your application uses floating palettes, use the latest version of the *Palette WDEF* (Window definition resource). Do not mix files from different versions of Tools Plus.

Registered Tools Plus users are entitled to take advantage of free updates and major upgrades. See the Technical Support chapter for details pertaining to determining the latest version and information on how to get updates.

Water's Edge Software is enhancing Tools Plus on an on-going basis by optimizing code and adding new features. We will inform registered users of newly available updates either by mail or by electronic mail.

Note: In order to ensure uninterrupted software update notification, please inform us if your mailing address or email account changes.

Evaluation Kit Registrants

A special edition of Tools Plus is distributed as an Evaluation Kit that can be obtained, free of charge, from user groups and various electronic bulletin boards and the Internet. Users of the Tools Plus Evaluation Kit are bound by restrictive terms and conditions that do not apply to *registered* Tools Plus developers who have purchased a license. If you have obtained a Tools Plus Development Kit as a result of registering an Evaluation Kit, discontinue using the evaluation kit and take advantage of the latest Tools Plus features. You *must* recompile your applications using the licensed libraries that come with the Development Kit.

Warning: Do *not* revert to using evaluation versions of Tools Plus. If you come across a newer version of Tools Plus in an Evaluation Kit and you have not received your equivalent registered upgrade, please contact Water's Edge Software.

For your information (recommended reading)

For any Macintosh programmer, we suggest you either own or have access to the entire series of "Inside Macintosh" technical reference guides by Addison Wesley. They are the definitive Macintosh bible for programmers of any caliber. They're worth their weight in midnight oil if you want to get into serious Macintosh programming.

Another indispensable tool is THINK Reference, an on-line reference manual for C or Pascal programmers. It describes all the Mac's data structures, variables, constants, functions and procedures. It also has valuable programming tips.

To program a Macintosh, you'll have to know the basics of the Macintosh toolbox. We recommend the following:

- Get familiar with QuickDraw by reading the relevant chapter in Inside Macintosh (or equivalent). This section details drawing in the Macintosh's graphic environment.
- Get familiar with the Font Manager by reading the relevant chapter of Inside Macintosh (or equivalent). This section deals with drawing text on the Macintosh's screen
- Have a working knowledge of the Macintosh's Memory Manager. This section deals with pointers, handles, and memory fragmentation. Tools Plus manages itself nicely by eliminating memory fragmentation, but the work you do outside Tools Plus should also be clean.
- The thing that differentiates a good Macintosh application from the rest of the world is that a user will find the program easy to learn and use. These benefits can be attributed greatly to a consistent and well-designed user interface. Learn the dos and don'ts of graphic user interface (GUI) design, then learn some more! Inside Macintosh also introduces you to the Macintosh's GUI and its standards. Another good way of learning is to get exposure to (and become familiar with) a *wide* range of Macintosh applications. You'll spot the good ones and the not so good ones after a while!

How to Get Started with Tools Plus

Tools Plus does a lot of things, but remember, it is not important to know everything about Tools Plus before you start using it. Here's a quick way of getting started with Tools Plus:

1. Install Tools Plus.
2. Read the Designing Your Application chapter for some basic guidelines on designing your application.
3. Take a look at our demo application for some *basic* ideas.
4. Run through all the tutorials included on your Tools Plus disk.
5. Create a simple application that:
   - creates pull-down menus
   - has a main event handler

   Play with it and get familiar with the basic functionality.
6. Expand your application to open a single window, and update your event handler to respond to events related to windows (doRefresh and doGoAway). Play with the application and get comfortable with it.
7. Add a few buttons and update your event handler to respond to button events (doButton). Again, get a feel for the application by playing around with it.
8. Add more GUI elements. Do one type of element at a time, like list boxes first then pop-up menus later. Each time you add a new type of GUI element, update your event handler to account for the new GUI elements and familiarize yourself with the growing application.
9. Add a second window and get a feel for how Tools Plus makes multiple window's work. Update your event handler to respond to doChgWindow events (a request by the user to activate another window).
10. Create a floating palette which is just another type of window, and notice how Tools Plus takes care of making it behave like a palette.
11. Add a tool bar (again, just a specific type of window), and experience how Tools Plus keeps it all working perfectly.

As a general rule, start with a very simple application, then incrementally add features while building your familiarity with Tools Plus.

Stress Testing Applications

Many of the bugs you will encounter as an application developer will be due to memory-related issues: forgetting to lock a handle when required, using a routine designed for an ordinary handle on a resource handle or vice versa, writing to an invalid memory address, unanticipated memory movement, unanticipated resource purging, and so on. Most of these problems appear intermittently making them difficult to reproduce and even harder to isolate, identify, and resolve. Worse still, the cause of a bug may have occurred dozens or even thousands of lines earlier, and the symptoms may appear only when part of your application tries to reference memory that has been corrupted by previous operations.

If you don't already have stress testing tools, or if you are looking for a good one that's affordable, look into QC™ from Onyx Technology. QC is a control panel that monitors the execution of a target application while you are testing it. It alerts you when it notices improper or questionable behavior that may manifest as a bug immediately or later in your application's execution. QC is priced at around $100 (US), and you can get it by email. Using QC during development and testing will save you countless hours of "hunt and destroy" debugging. It will also reduce your users' need for technical support due to enhanced application reliability and stability.

For more information about QC, please contact its creators at:

Onyx Technology
7811 27th West
Bradenton, FL 34209
USA

sales@onyx-tech.com
http://www.onyx-tech.com
Phone: 1-941-795-7801
Fax: 1-941-795-5901

Spotlight and other Testing Tools

Be aware that application testing tools do not always tell you the whole story. Such is the case with Onyx Technologies' Spotlight, a product that examines a running application and warns you, the developer, if the application appears to be doing something improper. Tools Plus, like most other applications, does a number of perfectly legal operations that Spotlight and similar products may misinterpret as potential problems. An example is our CursorShape routine: when your application calls CursorShape, Tools Plus first tries to locate a *color* cursor ('crsr' resource). It uses the toolbox's GetCCursor routine that may fail if a color cursor is not found. This is where Spotlight may report a warning even though Tools Plus does the right thing by checking to see if the color cursor was loaded, and if it wasn't, CursorShape calls the toolbox's GetCursor routine and checks that cursor as well.

Similar things happen when Tools Plus tries to load other resources using GetResource, discovers they are not available, then does the right thing and moves onward.

What this means to you, the developer, is that you should not assume that such warnings delivered by your testing software are correct. We use similar testing and stressing tools as we develop Tools Plus libraries, so we discover genuine problems and rectify them in our lab. You never see them. If your testing software can turn off warnings in libraries or places where you do not have the source code, please make use of this option. This will reduce or eliminate the number of false warnings you get.

If your testing software detects a memory leak or if you are really, *really* sure that a problem you have detected is inside Tools Plus, please feel free to contact us and we'll re-check and re-test our code to eliminate your worries.

Creating and Editing Resources

One of the many things that sets Macintosh apart from all other computers is the use of *resources*. For new Macintosh developers who are not familiar with the concept, a resource is a uniquely identifiable structure of bytes that represents a specific thing such as an icon or a picture. To use resources, the programmer calls a toolbox routine to get access to a specific resource type and number, such as retrieving 'ICON' number 128. An advantage of resources is that the Macintosh's built in Resource Manager takes care of storing resources in a resource file (likely as part of your application) and retrieving them. Resources can also be used to create more complex things like menus or a dialog (a window that is populated with user interface elements like buttons, scroll bars, fields, etc.)

Although Tools Plus lets you get away from resource-based programming to a great degree if you choose to, it also facilitates resource-based programming by letting you bring resources to life with a single line of code, something the Macintosh toolbox can't do. If you want to exploit resources to their greatest potential, we recommend getting a *powerful* resource editor such as Resorcerer from Mathemæsthetics. While Apple's ResEdit is free or almost free, it lacks the basic ability to specify colors and styles in a dialog. It also lacks most of the features included in Resorcerer that make it easier and safer to create, edit, organize, and maintain resources. Commercial developers and those who want to get the most out of Macintosh resources will benefit from using Resorcerer.

For more information about Resorcerer, please contact its creators at:

Mathemæsthetics, Inc.
PO Box 298
Boulder, CO 80306-0298
USA

resorcerer@mathemaesthetics.com
http://www.mathemaesthetics.com
Phone: 1-303-440-0707
Fax: 1-303-440-0504

The List Manager, List Boxes, Tables and Beyond

Tools Plus automates list boxes and it has built-in work arounds for all known List Manager bugs. This lets you create and interact with list boxes in a much simpler and more effective manner than using the toolbox's List Manager routines. Even so, Tools Plus does not alleviate the characteristics that are inherent to the Macintosh's List Manager, those being:

- Maximum limit of about 32,000 empty lines
- Maximum limit of about 32,000 bytes of data in all cells combined
- Uses a single font, style, and size for all lines throughout a list
- Can display only text
- Your application's user cannot edit list box text directly (your application can replace the content of a cell, though)
- All columns are the same width (Tools Plus supports only a single column)

All these limitations seemed appropriate in 1984 when Macintosh was first released, but they may be too restrictive for your application. One solution is to use a third-party 'LDEF' which may get around some of the List Managers inherent qualities. Tools Plus supports the use of third party 'LDEF's, but their programming tends to bring you a lot closer to traditional toolbox coding when dealing with the list box, without providing a clean break form the List Manager's inherent limitations.

If you need list or table functionality that is beyond that which is supported by the List Manager, please investigate StoneTable™ from StoneTablet Publishing. StoneTable provides comprehensive list and table features and services ranging from a simple list with text and images, to hierarchical lists such as the one seen in the Finder's "view by name," to near spreadsheet capabilities. It also integrates easily with Tools Plus, and it is in wide use in commercial applications. We strongly recommend StoneTable for all your list and table needs.

For more information about StoneTable or to get a free demo, please contact its creators at:

stack@teleport.com
http://www.teleport.com/~stack
Phone/fax: 1-503-287-3424

Tools Plus Features

The following is a *partial* list of Tools Plus features. Nearly all can be accomplished with a single Tools Plus routine. Although some of these features may appear to be simple, their functional equivalent, when programmed in C or Pascal without Tools Plus, often requires considerable programming effort and dozens (often *hundreds*) of lines of code. Unlike object oriented class libraries that add thousands of lines of code to your application, Tools Plus *reduces* the need for most of your user interface and event management code, often by as much as 80% to 95%.

The key advantages to every Tools Plus routine are:
- Virtually all features can be implemented with a single line of code
- They are consistent across all Systems and Macintosh models
- They are fully integrated with each other
- They adopt the "set and forget" principle of self-maintenance allowing you to easily create a user interface, then forget about it. Your application responds to very specific events, such as: a Pop-up menu was selected in Window 5, Pop-Up Menu 3, Item 6.

Appearance Manager

- All Appearance Manager windows and controls are supported
- Help in making your application run with or without an Appearance Manager
- Numerous Appearance Manager benefits available even when an Appearance Manager is not available

Windows

- All standard window types are supported
- Referenced by a window number instead of a pointer (pointers can be used if required)
- Create windows dynamically in your application and/or use 'WIND,' 'DLOG' and 'DITL' resources
- Full tool bar and floating palette services
- Movable modal dialog is fully supported, and is also available on pre-System 7 Macs
- Any window can be modal to prevent the use of menus or clicking outside the active window
- Optionally move and resize windows in real time instead of dragging a dotted gray outline
- Optional modal access to the Edit menu or any menu as specified by your application
- All Tools Plus user interface elements (such as buttons, editing fields, etc.) created in a window are automatically maintained… Tools Plus takes care of drawing them and making them work.
- Window positioning options when a new window is opened:
  √ centered on main monitor
  √ tiled (down and right of frontmost standard window, title bar is visible)
  √ window must be at least partially visible (in case its co-ordinates are remembered by a document, and it is opened on a Mac with a monitor that is smaller than the document's creator)
  √ entire window must be visible
- All user interface elements are correctly disabled/deselected when a window is deactivated and restored to their original state when the window is activated
- Windows with title bars are moved automatically when the use drags them (Tools Plus ensures that windows are not dragged *completely* off the screen or underneath the tool bar)
- Minimum/maximum size limits can also limit resizing to vertical or horizontal only
- Windows with a "size box" are automatically sized when the user drags the box
- Windows with a "zoom box" in their title bar automatically zoom between a standard size/position and a user-controlled size/position
- The window's update region is protected to exclude Tools Plus's user interface elements so your application can refresh a window's contents without concern about accidentally overwriting buttons, scroll bars, etc.
- When a window needs to be refreshed, your application can redraw custom objects (such as a picture background) before and/or after Tools Plus redraws its objects
- Your application can reposition, resize, or hide/show a window and still have Tools Plus maintain the user interface correctly

- Set/change backdrop color (defaults to white)
- Various routines help your application keep track of windows: frontmost, current, active, frontmost floating palette, frontmost standard window, most recently used, containing active editing field, and more.
- Closing a window automatically releases the memory occupied by Tools Plus user interface elements (buttons, scroll bars, etc.)
- The Finder and all other applications can be hidden as is seen in many installer application
- Globally substitute specific window procIDs to take advantage of system WDEFs if they are available (such as floating palettes) or use custom WDEFs when these system resources are not available

Tool Bar

- Optional tool bar is created below the menu bar on your main monitor
- As easy to use as a regular window
- Always remains the front window and is always active
- Can contain any Tools Plus user interface element, including pop-up menus, editing fields and picture buttons
- Adjustable height (width is fixed at main monitor's width)
- Automatically hidden when your application is suspended under MultiFinder (System 5 and 6), or System 7 or higher. When your application is activated, the tool bar is displayed again.
- Can optionally shift all your application's windows downward as the tool bar opens. This prevents windows from being partially obscured by the tool bar. If this option is used, windows are shifted back up when the tool bar is closed.
- If the tool bar's size is changed by your application (to add or remove a data entry area, for example), all open windows can optionally be shifted to accommodate the difference in tool bar size
- Windows cannot be dragged beneath the tool bar

Floating Palettes

- Floating palettes (often called "palettes" or "windoids") are supported
- As easy to use as a regular window
- You can use third-party window definitions (WDEFs) or write your own to get a specific look. Tools Plus takes care of making the window *behave* like a floating palette.
- Always remain in front of standard windows (if any are open) and behind the tool bar (if there is one)
- Can contain any Tools Plus user interface element, including pop-up menus, editing fields and picture buttons
- Palettes are moved automatically when the user drags them (Tools Plus ensures that they are not dragged *completely* off the screen or underneath the tool bar)
- When the user clicks on an object (such as a picture button) in floating palette, that palette is quickly brought to the front and refreshed before the click is processed. This results in very responsive and professional looking palettes.
- Automatically hidden when your application is suspended under MultiFinder (System 5 and 6), or System 7 or higher. When your application is activated, the palettes are displayed again.
- Two styles of palettes are included with Tools Plus: one with a title bar along the top of the palette (with optional title), and a second style that has a drag bar along the left side of the palette. The second style is well suited for horizontally oriented palettes that need to be as small as possible (the drag bar takes little space).

Dialogs/Alerts

- Automates dialogs ('DLOG' and related resources) making them as easy to use as the rest of Tools Plus.
- The dialog's window and its user interface elements are immediately usable without the need for support code or event management.
- The Edit menu works automatically on the active field allowing complete editing with full undo/redo services.
- Editing fields inherit all features found in Tools Plus fields including "single line" scrolling and progressive scrolling.
- Create working "non-standard" dialog items, such as picture buttons, list boxes, fields with scroll bars, 3D panels, each with a single line of code.

- All user interface elements in a dialog (i.e., the window, buttons, scroll bars, list boxes, etc.) inherit the benefits listed throughout this section.
- You can apply font family, size, style and color information to static text, fields and controls by including an ‘ictb’ resource. Tools Plus uses those settings on all Macintosh models whereas the toolbox’s Dialog Manager ignores them on older Macs and displays all text, fields and controls using Chicago 12pt.
- Fixes a Dialog Manager bug so you can use shorter, non-redundant ‘ictb’ resources to save memory and disk space.
- Dramatically simplifies your application’s interaction with dialogs (simpler code, much less code).
- Alerts are supported by standard C or Pascal statements, however they can be easily simulated by attractive Dynamic Alerts.

Buttons

- Buttons are referenced by a button number instead of a handle
- All 3 types of Macintosh buttons are supported: push button, radio button and check box
- Create buttons dynamically in your application and/or use ‘CNTL’ resources
- Custom control definitions (CDEFs) can be used. Tools Plus makes them behave like a push button, check box or radio button (it makes CDEFs work automatically).
- Can adopt a universal color scheme or be individually colored including individual backgrounds
- Set or get the font, size and style
- Enable/disable
- Select/deselect (check boxes and radio buttons only)
- Hide/shown
- Obscure (hide without affecting the window’s image)
- Delete or kill (delete without removing image)
- Move, resize, change co-ordinates or offset (change co-ordinates without altering image)
- Optional automatic move/resize as window’s size changes
- Default push button (a black outline is drawn around the button and it is automatically selected whenever the user types Return or Enter)
- Optional selection using a command key
- Buttons are automatically disabled when their parent window is inactive and return to their normal state when the window is reactivated.
- Each button can have its own font, font size, and style
- Optional double-click detection for radio buttons (can be interpreted to mean “select button and OK”)
- “Flash” a push button to make it appear as though it was clicked
- Globally substitute specific control procIDs to take advantage of system CDEFs if they are available (such as the 3D buttons in Mac OS 8’s Appearance Manager) or use custom CDEFs when these system resources are not available (such as SuperCDEFs)

Picture Buttons

- Picture buttons are referenced by a button number instead of a handle
- Any icon or PICT can be transformed into a button (they can also contain complex sets of images)
- Enable/disable
- Select/deselect
- Hide/shown
- Obscure (hide without affecting the window’s image)
- Delete or kill (delete without removing image)
- Move or offset (change co-ordinates without altering image)
- Optional automatic move as window’s size changes
- Optional “value” with a minimum/maximum range (like a scroll bar)

- Each button’s appearance and behavior is defined by selecting from a number of choices with literally thousands of combinations
- Picture buttons can be simple like click-sensitive icons, or they can be powerful and provide the appearance of animation
- Converts a simple black and white SICN icon a into richly featured 3D color picture button
- Multiple stage picture buttons have a different appearance for each value in the button’s range, such as an “on/off” button that has a value range from 0 to 1 and has the word “on” or “off” displayed.
- Optional locking in “selected” position (to behave like a radio button)
- Button’s value can change automatically as user interacts with it and/or under your application’s control
- Button’s value can change in the following manner:
  √ constant speed
  √ accelerate at a slow, moderate, or rapid rate
  √ precise rate, such as 90° per second
- Optional repeating events generated as long as the mouse button is held down
- Optional polarization increases the value when clicked on one side and decrease when clicked on the other
- Multiple disabling effects (or alternate image)
- Multiple selection effects (or alternate image)
- Picture buttons are automatically disabled when their parent window is inactive and return to their normal state when the window is reactivated.
- “Flash” a picture button to make it appear as though it was clicked

Scroll Bars

- Scroll bars are referenced by a scroll bar number instead of a handle
- Create scroll bars dynamically in your application and/or use ‘CNTL’ resources
- Custom control definitions (CDEFs) can be used. Tools Plus makes them behave like a scroll bar (it makes CDEFs work automatically).
- Can adopt a universal color scheme or be individually colored
- Set or get the font, size and style
- Enable/disable
- Hide/shown
- Obscure (hide without affecting the window’s image)
- Delete or kill (delete without removing image)
- Move, resize, change co-ordinates or offset (change co-ordinates without altering image)
- Optional automatic move/resize as window’s size changes
- Change value, minimum/maximum limit
- Minimum/maximum limits automatically adjusts if the value is set outside the range
- Optional limiting prevents value from exceeding minimum/maximum limit
- Optional throttling lets you control each scroll bar’s speed
- Optional “live scrolling” lets you easily scroll objects in real time as the scroll bar’s thumb is dragged… just process events as you normally do without the need for complex “action” routines
- You can easily install an action routine that is repeatedly called while the scroll bar is tracked and get details about the scroll bar calling the routine
- Attach scroll bars to the window’s right and/or bottom edge to have them automatically repositioned and resized when the window is resized
- Scroll bars are automatically “framed” (empty rectangle) when their parent window is inactive and they return to their normal state when the window is reactivated
- Scroll bars can be disabled instead of being framed when their parent window is inactive and return to their normal state when the window is reactivated
- Globally substitute specific control procIDs to take advantage of system CDEFs if they are available (such as

the sliders in Mac OS 8's Appearance Manager) or use custom CDEFs when these system resources are not available (such as SuperCDEFs)

Fields

- Easily created with a single line of code
- Editing fields are referenced by a field number instead of a handle
- Each field can manipulate and store up to 32K of text
- Optional vertical and/or horizontal scroll bars (scroll bars and text are always synchronized)
- Optional "live scrolling" lets you scroll the field's text in real time as the scroll bar's thumb is dragged
- Apply text and background colors
- Set or get the font, size and style
- Enable/disable a field with a variety of visual options
- Disabling a field with no visual effects makes it a "read only" field that can't be edited by the user… great for a set of instructions in a scrolling display area
- Copy only field (user can copy text from the field, but cannot change it)
- Hide/shown
- Obscure (hide without affecting the window's image)
- Delete or kill (delete without removing image)
- Move, resize, change co-ordinates or offset (change co-ordinates without altering image)
- Optional automatic move/resize as window's size changes
- Each field can have its own font, font size and style
- Field height can be specified in pixels or lines
- User edits a copy of the field's text so your application can revert to the original text at any time
- Get/set a field's text
- Get a field's edited text (the text the user is editing and that has not been saved)
- Paste text into a field under application control
- Filters allow/disallow specified characters
- Optional shifting to upper case or lower case characters as letters are typed
- Optional length limiting allows the user to type only a certain number of characters (the field's length)
- Sophisticated selection and cursor control allows:
  √ shorten/extend a selection a single character at a time (shift-arrow)
  √ shorten/extend a selection a word at a time (shift-option-arrow)
  √ move cursor a word at a time (option-arrow)
  √ vertical cursor movement that remember the horizontal position (i.e., move up in a straight line)
  √ shift-click extends/shortens a selection
  √ double-click selects a word
  √ typed text is always in view
  √ view follows cursor
- Get/set start and end of current selection range
- Tab sensing with optional tab to next field or to previous field if shift-tab was typed
- Progressive drag-selection works in conjunction with automatic scrolling making it easy to select additional characters that are out of view -- the further you move the cursor out of the field, the quicker it scrolls
- Text in single-line fields never disappears as it does with ordinary Macintosh fields -- it always scrolls reliably
- The Edit menu automatically interacts with the active editing field. The Undo, Cut, Copy, Paste, Clear (and optionally Select All) items are automatically enabled/disabled appropriately (see Menus). Selected text is automatically copied to and from the clipboard when using the Edit Menu. All Edit menu items are functional when they are created.
- Automatic management of user interface complexities that arise when fields are included on standard windows, floating palettes, and the tool bar

- Choose between memory efficiency for small fields and speed for larger fields
- Advanced low-memory protection (continue without undo, not enough memory, low memory warnings, etc.)
- Selected characters are automatically deselected when the field's window is inactive and return to their normal state when the window is reactivated

List Boxes

- Easily created with a single line of code
- List boxes are referenced by a list box number instead of a handle.
- Each list box can have its own font, font size, and style
- Custom list definitions (LDEFs) can be used. Tools Plus makes them work automatically.
- Add, change, delete lines as required (referenced by line number)
- Load a list box to capacity up to 30 times faster than using standard routines
- Set or get the font, size and style
- Set text and background colors
- Select/deselect lines
- Hide/shown
- Obscure (hide without affecting the window's image)
- Delete or kill (delete without removing image)
- Move, resize, change co-ordinates or offset (change co-ordinates without altering image)
- Optional automatic move/resize as window's size changes
- Various methods are available for selecting lines in a list box, such as: one line only, multiple lines, select as you drag the mouse, and many more
- When you create a list box, the first selected line will always be in view (i.e. not scrolled out of view)
- Your application can determine if a specific line, or *any* lines are selected in a list box
- A sorted set of resource names (such as fonts or sounds) can be inserted into a list box with a single command
- List box height can be specified in pixels or lines
- List boxes are automatically disabled (the lines are deselected and the scroll bar is disabled) when their window is inactive, and are enabled when the window is activated

Panels

- Produce group boxes or panels to give your application a professional appearance
- Easily created with a single line of code
- Optional flat or 3D title with inset or raised text
- Panels can be simple outlines or 3D (inset or raise)
- Optional rounded corners
- Each panel can have its own font, font size, and style
- Set or get the font, size and style
- Optionally deselect other buttons in the group when one is selected (ideal for radio buttons or CDEF groups)
- Memory efficient color map is shared by all panels in your application (programmer defined)
- Custom color map can optionally be adopted by panels as they are created
- Automatic mapping to lower color depths and/or black and white (optional override)
- Hide/shown
- Obscure (hide without affecting the window's image)
- Delete or kill (delete without removing image)
- Move, resize, change co-ordinates or offset (change co-ordinates without altering image)
- Optional automatic move/resize as window's size changes

Menus

- Easily created with a single command
- Menus are referenced by a menu number instead of a handle
- Create menus dynamically in your application and/or use ‘MENU’ and ‘MBAR’ resources
- A single routine can create all you application’s pull down and hierarchical menus, including a working Apple menu and fully functioning Edit menu
- Hierarchical menus are just as easy to create and maintain as regular pull-down menus
- Menu hierarchies are easily created by simply attaching a submenu to a menu item
- Prevents hierarchy errors such as:
  √ cyclical hierarchies
  √ submenus attached to multiple parents
  √ Command keys invoking an item in an orphan submenu (i.e., submenu with no parent), and more
- Prevents logical menu errors such as overwriting a submenu link with a command key
- Get/set default colors for the application’s menus, for a specified menu, or a menu item
- Menus can be easily added, changed, deleted, renamed, appended, enabled/disabled, restyled, prefaced with a symbol or icon, etc.
- Command key equivalents, icons, check marks, other special marks, and font styling are all supported
- Create a functioning Apple menu with a single command to give your application access to desk accessories (also includes the “About…” item that names your application)
- Access to the Help menu and Applications menu (System 7 or later) is as easy as any other menu
- Hide or show menu bar (automatically shows when application is suspended)
- When using the Finder in System 5 or System 6, menus are automatically enabled/disabled appropriately when a desk accessory is active. Under MultiFinder and System 7 or higher, the menu bar is automatically replaced with the desk accessory’s menu bar when the DA is active.
- The Edit menu’s Undo, Cut, Copy, Paste, and Clear (and optionally Select All) items automatically perform editing functions on the active field and in desk accessories. These menu items are automatically enabled/disabled appropriately.
- The Edit menu’s Undo item performs Undo/Redo operations on the active editing field. It automatically changes to “Undo Cut”, “Undo Copy”, “Undo Paste”, and “Undo Typing” as required. Selecting “Undo…” changes the item to “Redo…” and automatically performs the correct action.

Pop-Up Menus

- Easily created with a single line of code
- Pop-up menus are referenced by a menu and item number instead of a handle
- Create pop-up menu within your application and/or use ‘MENU’ resources
- Choose between Tools Plus style, System’s CDEF, or 3D pop-up menu styles
- Identical across all system versions thereby providing System 7 (and higher) features to prior systems
- Optional pop-down menu has a fixed (unchanging) title inside the pop-up box, and the available items appear beneath the control’s box
- Can adopt a universal color scheme or be individually colored including individual backgrounds
- Set or get the font, size and style
- Hierarchical pop-up menus
- Enable/disable
- Hide/shown
- Obscure (hide without affecting the window’s image)
- Delete or kill (delete without removing image)
- Move, resize, change co-ordinates or offset (change co-ordinates without altering image)
- Optional automatic move/resize as window’s size changes
- Supports “down arrow” suppression, multiple fonts, single item selection, and more

- When displaying a pop-up menu using a font other than the System Font (such as Geneva 9pt), Tools Plus's pop-up menus are unaffected by other applications that may use unorthodox programming techniques (such as a famous word processor that resets other applications' pop-up menu font size).
- Optionally display the selected item's icon in the pop-up menu's box
- Items in the menu can be added, changed, deleted, renamed, appended, enabled/disabled, restyled, prefaced with a symbol or icon, etc.
- Icons, check marks, other special marks, and font styling are all supported
- Pop-up menus are automatically disabled when their parent window is inactive and return to their normal state when the window is reactivated

Mouse

- Single, double, and triple clicks, as well as dragging is automatically detected and reported
- A cursor table can be used to detect if the mouse was clicked in specific areas (such as a picture or icon). This feature effectively makes any object "click sensitive."

Event Handling

- Tools Plus keeps all automatic processes running smoothly and calls your application's event handler routine if something has occurred such as the user selecting a menu or clicking a button.
- Tools Plus's revolutionary Event Translator reports events in a highly informative, simple, concise, and ready-to-use format instead of being cryptic and requiring message decoding. Example: the "Save" button was selected in the "Add Customer" window (button 4 was selected in window 15), or "Menu Item 16 was selected in Menu 4." This is much simpler than decoding event messages, tracking controls, and using handles and pointers.
- Your application can filter, modify, process, discard, and even synthesize events before Tools Plus processes them. It's easy to write an event filter routine.
- Most events are processed entirely by Tools Plus such as typing in an active editing field or selecting its text, or using the Edit menu on a field, or running desk accessories. It is completely automatic and requires no coding at all.
- Tools Plus takes care of maintaining the user interface before it reports an event to your application. For example, in a doRefresh event (refresh a window), all Tools Plus user interface elements (buttons, scroll bars, editing fields, etc.) are redrawn automatically.
- Many events can be ignored if your application doesn't care about them, such as when the user drags or re-sizes a window or if the text in a field is changed.
- Events that are *not* processed by Tools Plus are reported to your application, which can either ignore them or process them as required. This allows advanced programmers to implement their own special features.
- Tools Plus includes special routine that keep event processing running even while your application is busy

Apple Events

- Tools Plus automatically supports all four required Apple Events: "open application", "open documents", "print documents", and "quit application".
- Your can easily override the default Apple Event Handlers in Tools Plus by installing your own Apple Event Handler routines.
- Tools Plus also automatically handles a few esoteric Apple Events to account for changes in system font, small system font, views font, and other thematic changes.
- All other Apple Events are dispatched to the Apple Event Handler routines you install. You decide which Apple Events your application responds to and how.

Timers

- Generate timed events to time things such as animation, periodic updating of progress indicators, alarms, or just about anything else.
- A Timer can be set up with a frequency (i.e., 8 events per second, 200 events per hour, etc) or a period (i.e., 11 seconds between events, 1 day between events)

- Events can be “1-shot” (execute once only)
- Optionally synchronize an event to another event (great for flashing text or a strobing picture button)
- Route a Timer event to the application’s event handler, a window’s event handler, or a Timer event handler.
- Includes a Timer index to tell your application where it *should* be, in case your processor is not quick enough to generate events at the specified frequency.

Balloon Help

- Easily add Balloon Help to Tools Plus user interface elements and custom controls.
- Add Help to user interface elements in three ways:
  √ Using standard Macintosh Help resources in a dialog
  √ Use Help resources to dynamically assign Help to an object
  √ Dynamically set help for an object without using Help resources
- Help is associated with each Tools Plus user interface element, so if an element is moved, resized, hidden, deleted or scrolled, its associated Help balloon is updated appropriately.

Opening/Printing Files at Application Startup

- For applications running on System 6 or older, and for 680x0 applications that are not Apple Event aware, Tools Plus makes it easy to open or print documents at startup (i.e., if the user launches your application by double-clicking its documents, or selecting documents and choosing the File menu’s Print item).
- Tools Plus routines let you simply step through a list of documents for opening or printing without having to write and install Apple Event handler routine, or mess around with complex Apple Event routines.

Clipboard

- The clipboard is automatically maintained when using the Edit menu on an editing field (the clipboard contains copied text)
- Your application does not need to directly interact with the clipboard because the Edit menu takes care of moving text between your application and the clipboard and vice versa
- The Edit menu’s “Undo” operation restores the clipboard to its original state, so if you accidentally copy something to the clipboard and undo the copying, the clipboard’s original contents are automatically restored

Cursors

- The shape of the cursor can be changed with a single command
- Color cursors are fully supported
- The cursor’s shape changes automatically depending on where it is on the screen (i.e., I-Beam inside a field, different shape per cursor zone, an arrow outside the active window., etc.)
- A cursor table can be set up to automatically change the cursor’s shape depending on its position in a window, so it could become a “plus” cursor when located over a grid of cells (like in a spreadsheet application) and an arrow elsewhere.
- When the wrist watch cursor is displayed, Tools Plus discards all mouse clicks and typing except ⌘-. (operator halting a lengthy process)
- Optionally, your application can permit the clicking of a push button when the wrist watch cursor is displayed. This is useful if you have a Cancel button displayed on a window during a lengthy process.
- Multiple cursor animation sequences (like the Finder’s spinning wrist watch) are supported
- Your application is informed when the cursor moves into a new cursor zone in case you want to display a message as the user points to something

Desk Accessories

- Access to desk accessories is made possible by creating the Apple menu with a single command
- The Edit menu’s Undo, Cut, Copy, Paste, and Clear items interact automatically with desk accessories
- Desk accessories are handled automatically by Tools Plus (you don’t have to program anything to use them)

Dynamic Alerts

- Dynamic Alerts are alerts that are automatically sized in relation to the alert's contents -- they grow as big as needed to always appear aesthetically pleasing, and buttons are sized to accommodate their text. It's like having *hundreds* of customized alert boxes available at your disposal without having to create any resources.
- Dynamic Alerts are created with a *single* command and do not require the use of resources
- Full control over the alert's background color, text color and font settings for the message, and buttons' procIDs, colors and text settings
- You can use various combinations of Yes, No, OK, and Cancel buttons, or define your own combinations. A default button can be optionally specified.
- An icon can be optionally displayed and the alert can optionally beep when displayed
- Buttons can be selected using command keys (i.e., a "Yes" button can be selected with ⌘-Y)
- Dynamic Alerts, unlike Macintosh alerts, are unaffected by screen savers
- Dynamic Alerts are always centered perfectly on the main monitor regardless of the monitor's size or the number of monitors used

Custom Windows and Controls

- Third party WDEFs (window definitions) can be used for windows and CDEFs (control definitions) can be used for buttons and scroll bars. Tools Plus makes them work automatically.
- Custom CDEFs that do not relate to buttons or scroll bars and require specific application processing can also be used. Tools Plus hands events pertaining to those controls directly to your application.

Extras

- C/C++ programmers can use Pascal strings (the default), C and Pascal strings, or C strings exclusively as parameters in Tools Plus routines.
- Multiple language support (English, French, German, etc.)
- Zoom lines, such as those displayed by the Finder when a document is opened, are available to make objects appear to zoom out from the screen or zoom back down again.
- All icon types (cicn, icl8, icl4, ics8, ics4, ics#, SICN, ICON, and ICN#) are drawn with a single command.
  √ selected/unselected
  √ enabled/disabled
  √ drawn correctly across multiple monitors
- Indexed SICN drawing
- Macintosh-standard thermometer is system independent (identical to the Finder's)
- Numerous color text drawing routines
- Numerous picture drawing routines
- 'STR#' structure maintenance (create new structure, get, add, change and delete string)
- Create and destroy off-screen BitMap or PixMap
- System independent BitMap to region conversion
- Does the Mac running your application have an Appearance Manager?
- Is the Appearance Manager running? (i.e., not in "System 7 compatibility" mode)
- Initialize a structure (set to zero)
- Compare two structures for equality
- System version
- Tools Plus version

Monitors

- Color, gray-scale and monochrome (black and white) monitors are supported, as are multiple-monitor setups
- Specialized routines facilitate color-dependent drawing and inform your application of its environment
- All Tools Plus objects are displayed correctly even when spanning multiple monitors

- All objects support dynamic monitor resolution changes as made possible with today's multi-scan monitors
- All objects support dynamic monitor setup changes as made possible with System 7.5 or later

Memory

- Memory fragmentation due to opening and closing windows is eliminated regardless of the number of windows your application uses or has open at the same time
- Tools Plus is memory efficient requiring little of your application's memory for its own overhead
- Tools Plus does not fragment memory

Systems

- Tools Plus can be compiled into applications intended for System 5 and System 6 (Finder and MultiFinder), System 7 and higher, and Power Macintosh (in 68040 emulation and/or native mode).

Custom CDEFs

- Custom CDEFs (buttons and other controls) are available from third parties and Water's Edge Software. Tools Plus can make them as easy to implement and use as regular buttons and scroll bars.
- The Tools Plus Developer Kit includes SuperCDEFs, the world-class controls for discerning developers that give your applications a professional look. They include:
  √ replacement for standard Apple buttons with a white center in check box and radio buttons
  √ check box with an additional "undefined" state
  √ check box with a programmer-defined icon in place of the "x" in the box
  √ variety of buttons including optional 3D bodies and/or 3D title (inset or raised text)
  √ thermometer with optional "busy state" (the moving barber pole effect as seen in the Finder)
  √ variety of tabs including optional 3D bodies and/or 3D title (inset or raised text)
  √ variety of sliders including optional 3D bodies and/or 3D text for the scale (inset or raised text)

2 Installing Tools Plus

Installing Tools Plus in CodeWarrior C (68K)

C Tools Plus arrives a CD-ROM. To install Tools Plus on your hard disk, double-click the installer and select where you want to save Tools Plus (you can move the files later). Tools Plus for CodeWarrior C/C++ is made up of the following items:

Item	Description
ToolsPlus.Lib1 ToolsPlus.Lib2 ToolsPlus.Lib3 ToolsPlus.Lib4 ToolsPlus.Lib5 ToolsPlus.Lib6 ToolsPlus.Lib7	Libraries containing Tools Plus routines for 680x0 applications
ToolsPlus.Lib	Single 32-bit (large code model) library containing Tools Plus routines for 680x0 applications. This one library is equivalent to the seven numbered libraries.
ToolsPlus Plug-In.Lib	Library containing Tools Plus routines. This A4-relative 32-bit (large code model) library is equivalent to the seven numbered libraries. You use it only when creating 680x0 plug-ins and external code modules (i.e., not applications).
ToolsPlus.CW6&7.68K.Lib	Library containing additional routines required only when writing 680x0 applications compiled with CodeWarrior 6 and 7.
ToolsPlus.CW6&7.68K.A4.Lib	Library containing additional routines required only when writing 680x0 plug-ins and external code modules (i.e., not applications) compiled with CodeWarrior 6 and 7.
ToolsPlus.h	Header file for Tools Plus (includes defines, structures, and routines' prototypes)
ToolsPlus.c	Source code for routines that must be compiled as part of your application. These routines will be compiled according to your project's compiler settings for 680x0 processor and/or math co-processor optimization.
Palette WDEF	Optional resource file containing the WDEF resource (window definition) for floating palettes. Found in the "Optional Resources" folder.
Demos	Folder containing a demo application and its source code
Read Me	Important, late breaking news that may not be included in this manual

This User Manual is also part of the Tools Plus package. CodeWarrior is very flexible as to where it looks for files, so you can put Tools Plus libraries and support files just about anywhere on your disk as long as you define an *access path* to those files in your project's preferences. A good idea is to create a folder named "Tools Plus (68K) C/C++" (without the version number) in a convenient place on your hard disk. When you create a new project, set an access path to this folder. Your projects will automatically use newer versions of Tools Plus as long as you place the new files in this folder.

Drag the files containing the name "ToolsPlus" into your Tools Plus folder. You can also put the WDEF and all your other resources supplied with Tools Plus in the same folder for convenience.

Adding Tools Plus to a CodeWarrior C (68K) Project

Your Tools Plus installation contains a folder named "Starter Files." Inside this folder you will find prepared "Tools Plus ready" projects that are ready for you to use. You may also create your own projects. Projects using Tools Plus typically contain the following files:

Seg #	CodeWarrior Pro	CodeWarrior 11
1	Mac OS.lib MSL C.68K (2i).Lib PASCAL.68K.Lib ⇐Note!	Mac OS.lib MSL C.68K(2i).Lib PASCAL.68K.Lib ⇐Note!
2	ToolsPlus.Lib1	ToolsPlus.Lib1
3	ToolsPlus.Lib2	ToolsPlus.Lib2
4	ToolsPlus.Lib3	ToolsPlus.Lib3
5	ToolsPlus.Lib4	ToolsPlus.Lib4
6	ToolsPlus.Lib5	ToolsPlus.Lib5
7	ToolsPlus.Lib6	ToolsPlus.Lib6
8	ToolsPlus.Lib7 ToolsPlus.c	ToolsPlus.Lib7 ToolsPlus.c
9	*(your source code)*	*(your source code)*

Seg #	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	Mac OS.lib ANSI (2i) C.68K.Lib PASCAL.68K.Lib ⇐Note! console.stubs.c	Mac OS.lib ANSI (2i) C.68K.Lib PASCAL.68K.Lib ⇐Note!	Mac OS.lib ANSI (2i) C.68K.Lib P/RT.68K.lib ⇐Note!
2	ToolsPlus.Lib1	ToolsPlus.Lib1	ToolsPlus.Lib1
3	ToolsPlus.Lib2	ToolsPlus.Lib2	ToolsPlus.Lib2
4	ToolsPlus.Lib3	ToolsPlus.Lib3	ToolsPlus.Lib3
5	ToolsPlus.Lib4	ToolsPlus.Lib4	ToolsPlus.Lib4
6	ToolsPlus.Lib5	ToolsPlus.Lib5	ToolsPlus.Lib5
7	ToolsPlus.Lib6	ToolsPlus.Lib6	ToolsPlus.Lib6
8	ToolsPlus.Lib7 ToolsPlus.c	ToolsPlus.Lib7 ToolsPlus.CW6&7.68K.Lib ToolsPlus.c	ToolsPlus.Lib7 ToolsPlus.CW6&7.68K.Lib ToolsPlus.c
9	*(your source code)*	*(your source code)*	*(your source code)*

Adding Tools Plus to a CodeWarrior C (68K) Plug-In

If you are writing a plug-in or an external code module, you need to use A4 libraries in both CodeWarrior and in Tools Plus as indicated below:

Seg #	CodeWarrior Pro	CodeWarrior 11
1	Mac OS.lib MSL C.68K (2i).A4. Lib PASCAL.A4.68K.Lib ⇐Note! *(your main source code)*	Mac OS.lib MSL C.68K(2i).A4.Lib PASCAL.A4.68K.Lib ⇐Note! *(your main source code)*
2	ToolsPlus Plug-In.Lib ToolsPlus.c	ToolsPlus Plug-In.Lib ToolsPlus.c
3	*(your additional source code)*	*(your additional source code)*

Seg #	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	Mac OS.lib ANSI (2i) C.A4.68K.Lib PASCAL.A4.68K.Lib ⇐Note! console.stubs.c *(your main source code)*	Mac OS.lib ANSI (2i) C.A4.68K.Lib PASCAL.A4.68K.Lib ⇐Note! *(your main source code)*	(not supported)
2	ToolsPlus Plug-In.Lib ToolsPlus.c	ToolsPlus Plug-In.Lib ToolsPlus.CW6&7.68K.A4.Lib ToolsPlus.c	
3	*(your additional source code)*	*(your additional source code)*	

Note: The Pascal library is a component of your CodeWarrior Pascal compiler. It is also required by C/C++ applications.

☞ **Warning**: Before CodeWarrior 9, applications using Tools Plus *must* use 2-byte integers. Make sure your "4-byte ints" option is turned off in your project's preferences (processor options). If your application needs 4-byte integers, redeclare integers to be longs throughout your code.

☞ **Warning**: The segments containing the Tools Plus libraries and the ToolsPlus.c file will be constantly accessed while your 68K application is running. To reduce memory fragmentation, flag these segments as "Preload" and "Locked." Do not unload the segments containing Tools Plus libraries. You can ensure that this doesn't happen accidentally by flagging them as not "Purgeable."

If your 68K application will be integrated into a fat binary application (both 680x0 and PowerPC code in one application), do *not* flag your 68K segments as "Preload." When your application is running the PowerPC-native code, the 68K segments ('CODE' resources) are completely ignored, so preloading them just takes up memory. If you do preload them and your application is running PowerPC-native code, InitToolsPlus releases these code resources and frees up the memory they used to consume.

Installing Tools Plus in CodeWarrior Pascal (68K)

Pascal

Tools Plus arrives a CD-ROM. To install Tools Plus on your hard disk, double-click the installer and select where you want to save Tools Plus (you can move the files later). Tools Plus for CodeWarrior Pascal is made up of the following items:

ToolsPlus.Lib1 — Libraries containing Tools Plus routines for 680x0 applications
ToolsPlus.Lib2
ToolsPlus.Lib3
ToolsPlus.Lib4
ToolsPlus.Lib5
ToolsPlus.Lib6
ToolsPlus.Lib7

ToolsPlus.Lib — Single 32-bit (large code model) library containing Tools Plus routines for 680x0 applications. This one library is equivalent to the seven numbered libraries.

ToolsPlus Plug-In.Lib — Library containing Tools Plus routines. This A4-relative 32-bit (large code model) library is equivalent to the seven numbered libraries. You use it only when creating 680x0 plug-ins and external code modules (i.e., not applications).

ToolsPlus.CW6&7.68K.Lib — Library containing additional routines required only when writing 680x0 applications compiled with CodeWarrior 6 and 7.

ToolsPlus.CW6&7.68K.A4.Lib — Library containing additional routines required only when writing 680x0 plug-ins and external code modules (i.e., not applications) compiled with CodeWarrior 6 and 7.

ToolsPlus.p — Pascal interface file to Tools Plus routines (includes constants and types). This file also includes source code for routines that must be compiled as part of your application. These routines will be compiled according to your project's compiler settings for 680x0 processor and/or math co-processor optimization.

Palette WDEF — Optional resource file containing the WDEF resource (window definition) for floating palettes. Found in the "Optional Resources" folder.

Demos — Folder containing a demo application and its source code

Read Me — Important, late breaking news that may not be included in this manual

This User Manual is also part of the Tools Plus package. CodeWarrior is very flexible as to where it looks for files, so you can put Tools Plus libraries and support files just about anywhere on your disk as long as you define an *access path* to those files in your project's preferences. A good idea is to create a folder named "Tools Plus (68K) Pascal" (without the version number) in a convenient place on your hard disk. When you create a new project, set an access path to this folder. Your projects will automatically use newer versions of Tools Plus as long as you place the new files in this folder.

Drag the files containing the name "ToolsPlus" into your Tools Plus folder. You can also put the WDEF and all your other resources supplied with Tools Plus in the same folder for convenience.

Warning: Even if you are programming exclusively in Pascal, you may also need to install the CodeWarrior C/C++ compiler. If you are using CodeWarrior 8 or later and you are not using CodeWarrior's standard I/O libraries (SIOUX), then you must include the "console.stubs.c" file in your project. Our demo and tutorials do this. The "console.stubs.c" file is part of your CodeWarrior C/C++ setup, and you will also need to install CodeWarrior's C/C++ compiler and linker because of this file. The same applies if you are writing plug-ins in Pascal because one of the PowerPC "glue" files (PPCglue.c) is written in C and cannot be translated into Pascal. These are Metrowerks' requirements. They are not specific to Tools Plus.

Adding Tools Plus to a CodeWarrior Pascal (68K) Project

Your Tools Plus installation contains a folder named “Starter Files.” Inside this folder you will find prepared “Tools Plus ready” projects that are ready for you to use. You may also create your own projects. Projects using Tools Plus typically contain the following files:

Seg #	CodeWarrior Pro	CodeWarrior 11
1	Mac OS.lib MSL C.68K (2i).Lib PASCAL.68K.Lib	Mac OS.lib MacIntf(UPI).68K.lib MSL C.68K(2i).Lib PASCAL.68K.Lib
2	ToolsPlus.Lib1	ToolsPlus.Lib1
3	ToolsPlus.Lib2	ToolsPlus.Lib2
4	ToolsPlus.Lib3	ToolsPlus.Lib3
5	ToolsPlus.Lib4	ToolsPlus.Lib4
6	ToolsPlus.Lib5	ToolsPlus.Lib5
7	ToolsPlus.Lib6	ToolsPlus.Lib6
8	ToolsPlus.Lib7 ToolsPlus.p	ToolsPlus.Lib7 ToolsPlus.p
9	*(your source code)*	*(your source code)*

Seg #	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	Mac OS.lib MacIntf(UPI).68K.lib ANSI (2i) C.68K.Lib PASCAL.68K.Lib console.stubs.c	Mac OS.lib MacIntf(UPI).68K.lib ANSI (2i) C.68K.Lib PASCAL.68K.Lib	Pascal/Mac OS.lib MacIntf(UPI).68K.lib P/ANSI.68K.lib P/RT.68K.lib
2	ToolsPlus.Lib1	ToolsPlus.Lib1	ToolsPlus.Lib1
3	ToolsPlus.Lib2	ToolsPlus.Lib2	ToolsPlus.Lib2
4	ToolsPlus.Lib3	ToolsPlus.Lib3	ToolsPlus.Lib3
5	ToolsPlus.Lib4	ToolsPlus.Lib4	ToolsPlus.Lib4
6	ToolsPlus.Lib5	ToolsPlus.Lib5	ToolsPlus.Lib5
7	ToolsPlus.Lib6	ToolsPlus.Lib6	ToolsPlus.Lib6
8	ToolsPlus.Lib7 ToolsPlus.p	ToolsPlus.Lib7 ToolsPlus.CW6&7.68K.Lib ToolsPlus.p	ToolsPlus.Lib7 ToolsPlus.CW6&7.68K.Lib ToolsPlus.p
9	*(your source code)*	*(your source code)*	*(your source code)*

Adding Tools Plus to a CodeWarrior Pascal (68K) Plug-In

If you are writing a plug-in or an external code module, you need to use A4 libraries in both CodeWarrior and in Tools Plus as indicated below:

Seg #	CodeWarrior Pro	CodeWarrior 11
1	Mac OS.lib MSL C.68K (2i).A4. Lib PASCAL.A4.68K.Lib PascalA4.p *(your main source code)*	Mac OS.lib MacIntf(UPI).68K.lib MSL C.68K(2i).A4.Lib PASCAL.A4.68K.Lib PascalA4.p *(your main source code)*
2	ToolsPlus Plug-In.Lib ToolsPlus.p	ToolsPlus Plug-In.Lib ToolsPlus.p
3	*(your additional source code)*	*(your additional source code)*

Seg #	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	Mac OS.lib MacIntf(UPI).68K.lib ANSI (2i) C.A4.68K.Lib PASCAL.A4.68K.Lib PascalA4.p console.stubs.c *(your main source code)*	Mac OS.lib MacIntf(UPI).68K.lib ANSI (2i) C.A4.68K.Lib PASCAL.A4.68K.Lib PascalA4.p *(your main source code)*	(not supported)
2	ToolsPlus Plug-In.Lib ToolsPlus.p	ToolsPlus Plug-In.Lib ToolsPlus.CW6&7.68K.A4.Lib ToolsPlus.p	
3	*(your additional source code)*	*(your additional source code)*	

☞ **Warning**: The segments containing the Tools Plus libraries and the ToolsPlus.c file will be constantly accessed while your 68K application is running. To reduce memory fragmentation, flag these segments as “Preload” and “Locked.” Do not unload the segments containing Tools Plus libraries. You can ensure that this doesn’t happen accidentally by flagging them as not “Purgeable.”

If your 68K application will be integrated into a fat binary application (both 680x0 and PowerPC code in one application), do *not* flag your 68K segments as “Preload.” When your application is running the PowerPC-native code, the 68K segments (‘CODE’ resources) are completely ignored, so preloading them just takes up memory. If you do preload them and your application is running PowerPC-native code, InitToolsPlus releases these code resources and frees up the memory they used to consume.

Installing Tools Plus in CodeWarrior C (PPC)

C

Tools Plus arrives a CD-ROM.. To install Tools Plus on your hard disk, double-click the installer and select where you want to save Tools Plus (you can move the files later). Tools Plus for CodeWarrior C/C++ is made up of the following items:

ToolsPlus.Lib — Library containing Tools Plus routines for PowerPC applications

ToolsPlus Plug-In.Lib — Library containing Tools Plus routines. You use it only when creating PowerPC plug-ins and external code modules (i.e., not applications).

ToolsPlus.CW6&7.PPC.Lib — Library containing additional routines required only when writing PowerPC applications compiled with CodeWarrior 6 and 7

ToolsPlus.h — Header file for Tools Plus (includes defines, structures, and routines' prototypes)

ToolsPlus.c — Source code for routines that must be compiled as part of your application. These routines are kept in a separate file only to provide consistency with Tools Plus for 680x0 processors.

Palette WDEF — Optional resource file containing the WDEF resource (window definition) for floating palettes. Found in the "Optional Resources" folder.

Demos — Folder containing a demo application and its source code

Read Me — Important, late breaking news that may not be included in this manual

This User Manual is also part of the Tools Plus package. CodeWarrior is very flexible as to where it looks for files, so you can put Tools Plus libraries and support files just about anywhere on your disk as long as you define an *access path* to those files in your project's preferences. A good idea is to create a folder named "Tools Plus (PPC) C/C++" (without the version number) in a convenient place on your hard disk. When you create a new project, set an access path to this folder. Your projects will automatically use newer versions of Tools Plus as long as you place the new files in this folder.

Drag the files containing the name "ToolsPlus" into your Tools Plus folder. You can also put the WDEF and all your other resources supplied with Tools Plus in the same folder for convenience.

Adding Tools Plus to a CodeWarrior C (PPC) Project

Your Tools Plus installation contains a folder named “Starter Files.” Inside this folder you will find prepared “Tools Plus ready” projects that are ready for you to use. You may also create your own projects. Projects using Tools Plus typically contain the following files:

Grp	CodeWarrior Pro	CodeWarrior 11
1	MSL RuntimePPC.Lib InterfaceLib MSL C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note!	MWCRuntime.Lib InterfaceLib MSL C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note!
2	ToolsPlus.Lib ToolsPlus.c	ToolsPlus.Lib ToolsPlus.c
3	*(your source code)*	*(your source code)*

Grp	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	MWCRuntime.Lib InterfaceLib ANSI C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note! console.stubs.c	MWCRuntime.Lib InterfaceLib ANSI C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note!	MWCRuntime.Lib Interfacelib ANSI C.PPC.Lib P/Rt.PPC.lib ⇐Note! AppearanceLib ⇐Note!
2	ToolsPlus.Lib ToolsPlus.c	ToolsPlus.Lib ToolsPlus.CW6&7.PPC.Lib ToolsPlus.c	ToolsPlus.Lib ToolsPlus.CW6&7.PPC.Lib ToolsPlus.c
3	*(your source code)*	*(your source code)*	*(your source code)*

Adding Tools Plus to a CodeWarrior C (PPC) Plug-In

Grp	CodeWarrior Pro	CodeWarrior 11
1	InterfaceLib MSL C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note!	InterfaceLib MSL C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note!
2	ToolsPlus Plug-In.Lib ToolsPlus.c	ToolsPlus Plug-In.Lib ToolsPlus.c
3	PPCglue.c *(your source code)*	PPCglue.c *(your source code)*

Grp	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	InterfaceLib ANSI C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note! console.stubs.c	InterfaceLib ANSI C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note!	(not supported)
2	ToolsPlus Plug-In.Lib ToolsPlus.c	ToolsPlus Plug-In.Lib ToolsPlus.CW6&7.PPC.Lib ToolsPlus.c	
3	PPCglue.c *(your source code)*	PPCglue.c *(your source code)*	

Note: This is a component of your CodeWarrior Pascal compiler. It is also required by C/C++ applications.

Note: Make sure the AppearanceLib is set to “link weak” to allow your Appearance-savvy app to run on a Macintosh without an Appearance Manager.

Installing Tools Plus in CodeWarrior Pascal (PPC)

Pascal

Tools Plus arrives a CD-ROM. To install Tools Plus on your hard disk, double-click the installer and select where you want to save Tools Plus (you can move the files later). Tools Plus for CodeWarrior Pascal is made up of the following items:

ToolsPlus.Lib	Library containing Tools Plus routines for PowerPC applications
ToolsPlus Plug-In.Lib	Library containing Tools Plus routines. You use it only when creating PowerPC plug-ins and external code modules (i.e., not applications).
ToolsPlus.CW6&7.PPC.Lib	Library containing additional routines required only when writing PowerPC applications compiled with CodeWarrior 6 and 7
ToolsPlus.p	Pascal interface file to Tools Plus routines (includes constants and types). This file also includes source code for routines that must be compiled as part of your application. These routines are kept in a separate file only to provide consistency with Tools Plus for 680x0 processors.
Palette WDEF	Optional resource file containing the WDEF resource (window definition) for floating palettes. Found in the “Optional Resources” folder.
Demos	Folder containing a demo application and its source code
Read Me	Important, late breaking news that may not be included in this manual

This User Manual is also part of the Tools Plus package. CodeWarrior is very flexible as to where it looks for files, so you can put Tools Plus libraries and support files just about anywhere on your disk as long as you define an *access path* to those files in your project’s preferences. A good idea is to create a folder named “Tools Plus (PPC) Pascal” (without the version number) in a convenient place on your hard disk. When you create a new project, set an access path to this folder. Your projects will automatically use newer versions of Tools Plus as long as you place the new files in this folder.

Drag the files containing the name “ToolsPlus” into your Tools Plus folder. You can also put the WDEF and all your other resources supplied with Tools Plus in the same folder for convenience.

Warning: Even if you are programming exclusively in Pascal, you may also need to install the CodeWarrior C/C++ compiler. If you are using CodeWarrior 8 or later and you are not using CodeWarrior’s standard I/O libraries (SIOUX), then you must include the “console.stubs.c” file in your project. Our demo and tutorials do this. The “console.stubs.c” file is part of your CodeWarrior C/C++ setup, and you will also need to install CodeWarrior’s C/C++ compiler and linker because of this file. The same applies if you are writing plug-ins in Pascal because one of the PowerPC “glue” files (PPCglue.c) is written in C and cannot be translated into Pascal. These are Metrowerks’ requirements. They are not specific to Tools Plus.

Adding Tools Plus to a CodeWarrior Pascal (PPC) Project

Your Tools Plus installation contains a folder named "Starter Files." Inside this folder you will find prepared "Tools Plus ready" projects that are ready for you to use. You may also create your own projects. Projects using Tools Plus typically contain the following files:

Grp	CodeWarrior Pro	CodeWarrior 11
1	MSL RuntimePPC.Lib InterfaceLib MSL C.PPC.Lib PASCAL.PPC.lib AppearanceLib ⇐Note!	MWCRuntime.Lib InterfaceLib MacIntf(UPI).PPC.lib MSL C.PPC.Lib PASCAL.PPC.lib AppearanceLib ⇐Note!
2	ToolsPlus.Lib ToolsPlus.p	ToolsPlus.Lib ToolsPlus.p
3	*(your source code)*	*(your source code)*

Grp	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	MWCRuntime.Lib InterfaceLib MacIntf(UPI).PPC.lib ANSI C.PPC.Lib PASCAL.PPC.lib AppearanceLib ⇐Note! console.stubs.c	MWCRuntime.Lib InterfaceLib MacIntf(UPI).PPC.lib ANSI C.PPC.Lib PASCAL.PPC.lib AppearanceLib ⇐Note!	MWPRuntime.Lib InterfaceLib MacIntf(UPI).PPC.lib P/ANSI.PPC.Lib P/Rt.PPC.lib AppearanceLib ⇐Note!
2	ToolsPlus.Lib ToolsPlus.p	ToolsPlus.Lib ToolsPlus.CW6&7.PPC.Lib ToolsPlus.p	ToolsPlus.Lib ToolsPlus.CW6&7.PPC.Lib ToolsPlus.p
3	*(your source code)*	*(your source code)*	*(your source code)*

Adding Tools Plus to a CodeWarrior Pascal (PPC) Plug-In

Grp	CodeWarrior Pro	CodeWarrior 11
1	InterfaceLib MSL C.PPC.Lib PASCAL.PPC.lib AppearanceLib ⇐Note!	InterfaceLib MacIntf(UPI).PPC.lib MSL C.PPC.Lib PASCAL.PPC.lib AppearanceLib ⇐Note!
2	ToolsPlus Plug-In.Lib ToolsPlus.p	ToolsPlus Plug-In.Lib ToolsPlus.p
3	PPCglue.c *(your source code)*	PPCglue.c *(your source code)*

Grp	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	InterfaceLib MacIntf(UPI).PPC.lib ANSI C.PPC.Lib PASCAL.PPC.lib console.stubs.c	InterfaceLib MacIntf(UPI).PPC.lib ANSI C.PPC.Lib PASCAL.PPC.lib	(not supported)
2	ToolsPlus Plug-In.Lib ToolsPlus.p	ToolsPlus Plug-In.Lib ToolsPlus.CW6&7.PPC.Lib ToolsPlus.p	
3	PPCglue.c *(your source code)*	PPCglue.c *(your source code)*	

Note: Make sure the AppearanceLib is set to "link weak" to allow your Appearance-savvy app to run on a Macintosh without an Appearance Manager.

Installing Tools Plus in THINK C/C++ (68K) 5, 6 and 7

C Tools Plus arrives on a CD-ROM. To install Tools Plus on your hard disk, double-click the installer and select where you want to save Tools Plus (you can move the files later). Tools Plus for THINK C/C++ is made up of the following items:

ToolsPlus.Lib1 *through to* ToolsPlus.Lib7	Libraries containing Tools Plus routines
ToolsPlus.h	Header file for Tools Plus (includes defines, structures, and routines' prototypes)
ToolsPlus.c	Source code for routines that must be compiled as part of your application. These routines will be compiled according to your project's compiler settings for 680x0 processor and/or math co-processor optimization.
Palette WDEF	Optional resource file containing the WDEF resource (window definition) for floating palettes. Found in the "Optional Resources" folder.
Demos	Folder containing a demo application and its source code
Read Me	Important, late breaking news that may not be included in this manual

This User Manual is also part of the Tools Plus package. Drag the *ToolsPlus.Lib1* through *ToolsPlus.Lib7* libraries into the *Mac Libraries* folder, or wherever you keep your other libraries. Drag *ToolsPlus.h* into the *Mac #includes* folder, or wherever you keep your other header files. You can drag the *ToolsPlus.c* file into your *Mac Libraries* folder, even though it is not a library. Keeping it with the ToolsPlus.Lib1 through ToolsPlus.Lib7 libraries will help to remind you to include *ToolsPlus.c* in your project.

Adding Tools Plus to a THINK C (68K) Project

Your Tools Plus installation contains a folder named "Starter Files." Inside this folder you will find prepared "Tools Plus ready" projects that are ready for you to use. You may also create your own projects. Projects using Tools Plus typically contain the following files:

Segment 1:	MacTraps MacTraps2
Segment 2:	ANSI
Segment 3:	ToolsPlus.Lib1
Segment 4:	ToolsPlus.Lib2
Segment 5:	ToolsPlus.Lib3
Segment 6:	ToolsPlus.Lib4
Segment 7:	ToolsPlus.Lib5
Segment 8:	ToolsPlus.Lib6
Segment 9:	ToolsPlus.c ToolsPlus.Lib7
Segment 10:	*(your source code)*

☞ **Warning**: The segments containing the Tools Plus libraries and the ToolsPlus.c file will be constantly accessed while your application is running. To reduce memory fragmentation, flag these segments as "Preload" and "Locked." Do not unload the segments containing Tools Plus libraries. You can ensure that this doesn't happen accidentally by flagging them as not "Purgeable."

Installing Tools Plus in Symantec C/C++ (68K) 8.0.5 or later

C Tools Plus arrives a CD-ROM. To install Tools Plus on your hard disk, double-click the installer and select where you want to save Tools Plus (you can move the files later). Tools Plus for THINK C/C++ is made up of the following items:

ToolsPlus.Lib1.o *through to* ToolsPlus.Lib7.o	Libraries containing Tools Plus routines
QDGlobals.a.o	Compatibility library for Symantec Project Manager's (SPM) new MPW-style linker
ToolsPlus.h	Header file for Tools Plus (includes defines, structures, and routines' prototypes)
ToolsPlus.c	Source code for routines that must be compiled as part of your application. These routines will be compiled according to your project's compiler settings for 680x0 processor and/or math co-processor optimization.
Palette WDEF	Optional resource file containing the WDEF resource (window definition) for floating palettes. Found in the "Optional Resources" folder.
Demos	Folder containing a demo application and its source code
Read Me	Important, late breaking news that may not be included in this manual

This User Manual is also part of the Tools Plus package. Drag the *ToolsPlus.Lib1.o* through *ToolsPlus.Lib7.o* and *QDGlobals.a.o* libraries into the *Standard Libraries* folder, and *ToolsPlus.h* into the *Macintosh Libraries* folder, or where ever you keep your other header files. You can drag the *ToolsPlus.c* file into your *Standard Libraries* folder, even though it is not a library. Keeping it with the ToolsPlus.Lib1.o through ToolsPlus.Lib7.o libraries will help to remind you to include *ToolsPlus.c* in your project.

Adding Tools Plus to an SPM C/C++ (68K) Project

Your Tools Plus installation contains a folder named "Starter Files." Inside this folder you will find prepared "Tools Plus ready" projects that are ready for you to use. You may also create your own projects. SPM projects using Tools Plus typically contain the following files. Note that grouping is purely for convenience and tidiness.

	YourApp.c	*(your source code)*
	YourProjName.π.lo	*(your project's link order, created by SPM)*
Libraries:	<68Kansi-small.o> <Interface.o> <Mathlib.o> <MPW68KRuntime.o>	
Tools Plus	QDGlobals.a.o ToolsPlus.c ToolsPlus.Lib1.o ToolsPlus.Lib2.o ToolsPlus.Lib3.o ToolsPlus.Lib4.o ToolsPlus.Lib5.o ToolsPlus.Lib6.o ToolsPlus.Lib7.o	

Starting with release 5 of Symantec C/C++ 8 (version 8.0.5), Symantec has departed from its traditional use of the THINK linker in SPM. SPM now uses an MPW linker which has some inherent differences that prevent it from being able to use standard Symantec 'PROJ' files (the standard format of Tools Plus libraries). The ".o" Tools Plus files have been compiled specifically for SPM.

The QDGlobals.a.o library lets your application access QuickDraw globals in the traditional manner (i.e., thePort) instead of using the newer universal method (i.e., qd.thePort). QDGlobals.a.o must be linked first. You can change the link order of your project by compiling your application and letting SPM create a link order file (YourProjectName.lo) for your project. Open the link order file and move QDGlobals.a.o such that it appears on the first line.

Installing Tools Plus in Symantec C/C++ (PPC) 8.6 or later

Tools Plus arrives a CD-ROM. To install Tools Plus on your hard disk, double-click the installer and select where you want to save Tools Plus (you can move the files later). Tools Plus for Symantec C/C++ is made up of the following items:

ToolsPlus.Lib	Library containing Tools Plus routines for PowerPC applications
QDGlobals.c	Gives your application access to QuickDraw globals
ToolsPlus.h	Header file for Tools Plus (includes defines, structures, and routines' prototypes)
ToolsPlus.c	Source code for routines that must be compiled as part of your application. These routines are kept in a separate file only to provide consistency with Tools Plus for 680x0 processors.
Palette WDEF	Optional resource file containing the WDEF resource (window definition) for floating palettes. Found in the "Optional Resources" folder.
Demos	Folder containing a demo application and its source code
Read Me	Important, late breaking news that may not be included in this manual

This User Manual is also part of the Tools Plus package. Symantec C++ is very flexible as to where it looks for files, so you can put Tools Plus libraries and support files just about anywhere on your disk as long as you define an *access path* to those files in your project's preferences. A good idea is to create a folder named "Tools Plus (PPC) C/C++" (without the version number) in a convenient place on your hard disk. When you create a new project, set an access path to this folder. Your projects will automatically use newer versions of Tools Plus as long as you place the new files in this folder.

Drag the files containing the name "ToolsPlus" into your Tools Plus folder. You can also put the WDEF and all your other resources supplied with Tools Plus in the same folder for convenience.

Adding Tools Plus to an SPM C/C++ (PPC) Project

Your Tools Plus kit contains a folder named "Starter Files." Inside this folder you will find prepared "Tools Plus ready" projects that are ready for you to use. You may also create your own projects. SPM projects using Tools Plus typically contain the following files. Note that grouping is purely for convenience and tidiness.

Glue:	<PPCMW_Compatibility.o>	
Runtime:	<InterfaceLib> <MathLib> <AppearanceLib> <str.c>	 ⇐Note!
Tools Plus	ToolsPlus.c ToolsPlus.Lib	
Main:	*YourApp.c* QDGlobals.c	*(your source code)*

Starting with release 6 of Symantec C/C++ 8 (version 8.6), the Symantec Project Manager can make use of CodeWarrior libraries. This new capability is what lets us bring Tools Plus libraries to users of the Symantec PowerPC compiler. Make sure that you have installed the "PPC .lib Converter" in the "(Translators)" folder before you try to compile any Tools Plus project. The PPCMW_Compatibility.o library provides the glue between Tools Plus libraries and the routines found in Symantec's libraries. There is no perceivable performance penalty for using this glue.

Note: Make sure the AppearanceLib is set to "link weak" to allow your Appearance-savvy app to run on a Macintosh without an Appearance Manager.

Installing Tools Plus in THINK Pascal (68K)

Pascal

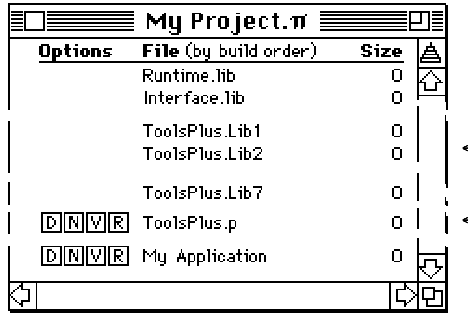
Tools Plus arrives a CD-ROM. To install Tools Plus on your hard disk, double-click the installer and select where you want to save Tools Plus (you can move the files later). Tools Plus for THINK Pascal is made up of the following items:

ToolsPlus.Lib1 *through to* ToolsPlus.Lib7	Libraries containing Tools Plus routines
ToolsPlus.p	Pascal interface file to Tools Plus routines (includes constants and types). This file also includes source code for routines that must be compiled as part of your application. These routines will be compiled according to your project's compiler settings for 680x0 processor and/or math co-processor optimization.
Palette WDEF	Optional resource file containing the WDEF resource (window definition) for floating palettes. Found in the "Optional Resources" folder.
Demos	Folder containing a demo application and its source code
Read Me	Important, late breaking news that may not be included in this manual

This User Manual is also part of the Tools Plus package. Drag the *ToolsPlus.Lib1* through *ToolsPlus.Lib7* libraries into the *Libraries* folder, and *ToolsPlus.p* into the *Interface* folder, or where ever you keep your other libraries and interfaces.

Adding Tools Plus to a THINK Pascal (68K) Project

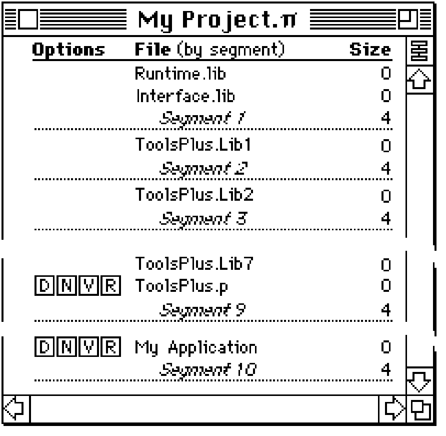
To add Tools Plus to a THINK Pascal project, first open your project, then use the Project menu's Add File command to add the files *ToolsPlus.Lib1* through *ToolsPlus.Lib7* and *ToolsPlus.p* to your project. The following illustrations explain the required placement of these files.



Build Order

Though libraries can be located anywhere in your build order, placing them near the top (early in the compiling order) will organize your project a little better.

The ToolsPlus.p interface has to be compiled before your source code makes reference to it. Placing it immediately below the libraries is a good, safe place.



By Segment

Drag each of the Tools Plus libraries into their own segment.

For convenience, drag the ToolsPlus.p interface file into the segment containing the last Tools Plus library.

After Compiling

So far, you've told THINK Pascal what files to use and the order in which they should be compiled. When you compile your project the first time, THINK Pascal loads the specified libraries and integrates them into your project file, and it compiles source files (including ToolsPlus.p) and integrates them in your project file. After your first compile, you'll notice that THINK Pascal automatically added a number of new items to your project file:

«ToolsPlus.Lib1:1»
«ToolsPlus.Lib2:1»
⇓
«ToolsPlus.Lib7:1»
«%_MethTables»
«%_SelProcs»
«%_Profiler»

My Project.π

Options	File (by segment)	Size
	Runtime.lib	18374
	Interface.lib	12812
	«%_MethTables»	2
	«%_SelProcs»	0
	«%_Profiler»	4460
	Segment 1	35652
	ToolsPlus.Lib1	0
	«ToolsPlus.Lib1:1»	31310
	Segment 2	31314
	ToolsPlus.Lib2	0
	«ToolsPlus.Lib2:1»	31048
	Segment 3	31050
	ToolsPlus.Lib7	0
D N V R	ToolsPlus.p	68
	«ToolsPlus.Lib7:1»	6448
	Segment 8	6520
D N V R	Sound.p	0
D N V R	My Application	10022
	Segment 9	10026

The «ToolsPlus.Lib1:1» through «ToolsPlus.Lib7:1» items contain the object code from the Tools Plus libraries, while «%_MethTables», «%_SelProcs» and «%_Profiler» items are part of THINK Pascal's overhead (consult your THINK Pascal User Manual for details).

Drag «ToolsPlus.Lib1:1» into the same segment as ToolsPlus.Lib1, «ToolsPlus.Lib2:1» into the same segment as ToolsPlus.Lib2, and so on for all the libraries.

Drag «%_MethTables», «%_SelProcs» and «%_Profiler» to any segment that won't be unloaded while your application is running, such as the one containing the Runtime.Lib library.

Even though the project window indicates that Segment 1 exceeds the 32K limit imposed on segments, the project *will* compile and run. When you build your application, the smart linker will strip away unneeded code and significantly reduce the size of this segment.

Note: Tools Plus does not have a dependency on the Runtime.Lib library. Unless *your* application needs routines that are found only in the full Runtime.Lib library, you can use the smaller µRuntime.Lib library instead.

☞ **Warning**: The segments containing the Tools Plus libraries and the ToolsPlus.p file will be constantly accessed while your application is running. To reduce memory fragmentation, flag these segments as "Preload" and "Locked." Do not unload the segments containing Tools Plus libraries. You can ensure that this doesn't happen accidentally by flagging them as not "Purgeable."

Compiling the Tools Plus demo

Compiling the CodeWarrior C (68K) Demo Application

The easiest way to compile the CodeWarrior C demo application included with Tools Plus is to have the following files in the same folder:

Demo.µ	ToolsPlus.Lib1	ToolsPlus.h
Demo.π.rsrc	ToolsPlus.Lib2	ToolsPlus.c
Demo.c	ToolsPlus.Lib3	PascalStrHandles.c
	ToolsPlus.Lib4	
	ToolsPlus.Lib5	
	ToolsPlus.Lib6	
	ToolsPlus.Lib7	
	ToolsPlus.CW6&7.68K.Lib	

Double-click the Demo.µ project file to launch CodeWarrior C/C++, then run your project.

If you are creating a new project (instead of using the one included as part of Tools Plus), you first need to install Tools Plus in your project file as described earlier in this chapter. Copy the files "Demo.µ", "Demo.π.rsrc" and "PascalStrHandles.c" to the folder containing your project. Add the following files to your new project:

Seg #	CodeWarrior Pro	CodeWarrior 11
1	Mac OS.lib MSL C.68K (2i).Lib PASCAL.68K.Lib ⇐Note!	Mac OS.lib MSL C.68K(2i).Lib PASCAL.68K.Lib ⇐Note!
2	ToolsPlus.Lib1	ToolsPlus.Lib1
3	ToolsPlus.Lib2	ToolsPlus.Lib2
4	ToolsPlus.Lib3	ToolsPlus.Lib3
5	ToolsPlus.Lib4	ToolsPlus.Lib4
6	ToolsPlus.Lib5	ToolsPlus.Lib5
7	ToolsPlus.Lib6	ToolsPlus.Lib6
8	ToolsPlus.Lib7 ToolsPlus.c	ToolsPlus.Lib7 ToolsPlus.c
9	Demo.π.rsrc Demo.c	Demo.π.rsrc Demo.c

Seg #	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	Mac OS.lib ANSI (2i) C.68K.Lib PASCAL.68K.Lib ⇐Note! console.stubs.c	Mac OS.lib ANSI (2i) C.68K.Lib PASCAL.68K.Lib ⇐Note!	Mac OS.lib ANSI (2i) C.68K.Lib P/RT.68K.lib ⇐Note!
2	ToolsPlus.Lib1	ToolsPlus.Lib1	ToolsPlus.Lib1
3	ToolsPlus.Lib2	ToolsPlus.Lib2	ToolsPlus.Lib2
4	ToolsPlus.Lib3	ToolsPlus.Lib3	ToolsPlus.Lib3
5	ToolsPlus.Lib4	ToolsPlus.Lib4	ToolsPlus.Lib4
6	ToolsPlus.Lib5	ToolsPlus.Lib5	ToolsPlus.Lib5
7	ToolsPlus.Lib6	ToolsPlus.Lib6	ToolsPlus.Lib6
8	ToolsPlus.Lib7 ToolsPlus.c	ToolsPlus.Lib7 ToolsPlus.CW6&7.68K.Lib ToolsPlus.c	ToolsPlus.Lib7 ToolsPlus.CW6&7.68K.Lib ToolsPlus.c
9	Demo.π.rsrc Demo.c	Demo.π.rsrc Demo.c	Demo.π.rsrc Demo.c

Note: This library is a component of your CodeWarrior Pascal compiler. It is also required by C/C++ applications.

You can now build your project and run your application.

Compiling the CodeWarrior Pascal (68K) Demo Application

The easiest way to compile the CodeWarrior Pascal demo application included with Tools Plus is to have the following files in the same folder:

Demo.µ	ToolsPlus.Lib1	ToolsPlus.p
Demo.π.rsrc	ToolsPlus.Lib2	
Demo.p	ToolsPlus.Lib3	
	ToolsPlus.Lib4	
	ToolsPlus.Lib5	
	ToolsPlus.Lib6	
	ToolsPlus.Lib7	
	ToolsPlus.CW6&7.68K.Lib	

Double-click the Demo.µ project file to launch CodeWarrior Pascal, then run your project.

If you are creating a new project (instead of using the one included as part of Tools Plus), you first need to install Tools Plus in your project file as described earlier in this chapter. Copy the files “Demo.µ” and “Demo.π.rsrc” to the folder containing your project. Add the following files to your new project:

Seg #	CodeWarrior Pro	CodeWarrior 11
1	Mac OS.lib MSL C.68K (2i).Lib PASCAL.68K.Lib	Mac OS.lib MacIntf(UPI).68K.lib MSL C.68K(2i).Lib PASCAL.68K.Lib
2	ToolsPlus.Lib1	ToolsPlus.Lib1
3	ToolsPlus.Lib2	ToolsPlus.Lib2
4	ToolsPlus.Lib3	ToolsPlus.Lib3
5	ToolsPlus.Lib4	ToolsPlus.Lib4
6	ToolsPlus.Lib5	ToolsPlus.Lib5
7	ToolsPlus.Lib6	ToolsPlus.Lib6
8	ToolsPlus.Lib7 ToolsPlus.p	ToolsPlus.Lib7 ToolsPlus.p
9	Demo.π.rsrc Demo.p	Demo.π.rsrc Demo.p

Seg #	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	Mac OS.lib MacIntf(UPI).68K.lib ANSI (2i) C.68K.Lib PASCAL.68K.Lib console.stubs.c	Mac OS.lib MacIntf(UPI).68K.lib ANSI (2i) C.68K.Lib PASCAL.68K.Lib	Pascal/Mac OS.lib MacIntf(UPI).68K.lib P/ANSI.68K.lib P/RT.68K.lib
2	ToolsPlus.Lib1	ToolsPlus.Lib1	ToolsPlus.Lib1
3	ToolsPlus.Lib2	ToolsPlus.Lib2	ToolsPlus.Lib2
4	ToolsPlus.Lib3	ToolsPlus.Lib3	ToolsPlus.Lib3
5	ToolsPlus.Lib4	ToolsPlus.Lib4	ToolsPlus.Lib4
6	ToolsPlus.Lib5	ToolsPlus.Lib5	ToolsPlus.Lib5
7	ToolsPlus.Lib6	ToolsPlus.Lib6	ToolsPlus.Lib6
8	ToolsPlus.Lib7 ToolsPlus.p	ToolsPlus.Lib7 ToolsPlus.CW6&7.68K.Lib ToolsPlus.p	ToolsPlus.Lib7 ToolsPlus.CW6&7.68K.Lib ToolsPlus.p
9	Demo.π.rsrc Demo.p	Demo.π.rsrc Demo.p	Demo.π.rsrc Demo.p

You can now build your project and run your application.

Compiling the CodeWarrior C (PPC) Demo Application

The easiest way to compile the CodeWarrior C demo application included with Tools Plus is to have the following files in the same folder:

Demo.µ	ToolsPlus.Lib	PascalStrHandles.c
Demo.π.rsrc	ToolsPlus.CW6&7.PPC.Lib	
Demo.c	ToolsPlus.h	
	ToolsPlus.c	

Double-click the Demo.µ project file to launch CodeWarrior C/C++, then run your project.

If you are creating a new project (instead of using the one included as part of Tools Plus), you first need to install Tools Plus in your project file as described earlier in this chapter. Copy the files “Demo.µ”, “Demo.π.rsrc” and “PascalStrHandles.c” to the folder containing your project. Add the following files to your new project:

Grp	CodeWarrior Pro	CodeWarrior 11
1	MSL RuntimePPC.Lib InterfaceLib MSL C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note!	MWCRuntime.Lib InterfaceLib MSL C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note!
2	ToolsPlus.Lib ToolsPlus.c	ToolsPlus.Lib ToolsPlus.c
3	Demo.π.rsrc Demo.c	Demo.π.rsrc Demo.c

Grp	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	MWCRuntime.Lib InterfaceLib ANSI C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note! console.stubs.c	MWCRuntime.Lib InterfaceLib ANSI C.PPC.Lib PASCAL.PPC.lib ⇐Note! AppearanceLib ⇐Note!	MWCRuntime.Lib Interfacelib ANSI C.PPC.Lib P/Rt.PPC.lib ⇐Note! AppearanceLib ⇐Note!
2	ToolsPlus.Lib ToolsPlus.c	ToolsPlus.Lib ToolsPlus.CW6&7.PPC.Lib ToolsPlus.c	ToolsPlus.Lib ToolsPlus.CW6&7.PPC.Lib ToolsPlus.c
3	Demo.π.rsrc Demo.c	Demo.π.rsrc Demo.c	Demo.π.rsrc Demo.c

Note: This is a component of your CodeWarrior Pascal compiler. It is also required by C/C++ applications.

Note: Make sure the AppearanceLib is set to “link weak” to allow your Appearance-savvy app to run on a Macintosh without an Appearance Manager.

You can now build your project and run your application.

Compiling the CodeWarrior Pascal (PPC) Demo Application

The easiest way to compile the CodeWarrior Pascal demo application included with Tools Plus is to have the following files in the same folder:

Demo.µ	ToolsPlus.Lib
Demo.π.rsrc	ToolsPlus.CW6&7.PPC.Lib
Demo.p	ToolsPlus.p

Double-click the Demo.µ project file to launch CodeWarrior Pascal, then run your project.

If you are creating a new project (instead of using the one included as part of Tools Plus), you first need to install Tools Plus in your project file as described earlier in this chapter. Copy the files "Demo.µ" and "Demo.π.rsrc" to the folder containing your project. Add the following files to your new project:

Grp	CodeWarrior Pro	CodeWarrior 11
1	MSL RuntimePPC.Lib InterfaceLib MSL C.PPC.Lib PASCAL.PPC.lib AppearanceLib ⇐Note!	MWCRuntime.Lib InterfaceLib MacIntf(UPI).PPC.lib MSL C.PPC.Lib PASCAL.PPC.lib AppearanceLib ⇐Note!
2	ToolsPlus.Lib ToolsPlus.p	ToolsPlus.Lib ToolsPlus.p
3	Demo.π.rsrc Demo.p	Demo.π.rsrc Demo.p

Grp	CodeWarrior 8, 9, 10	CodeWarrior 7	CodeWarrior 6
1	MWCRuntime.Lib InterfaceLib MacIntf(UPI).PPC.lib ANSI C.PPC.Lib PASCAL.PPC.lib AppearanceLib ⇐Note! console.stubs.c	MWCRuntime.Lib InterfaceLib MacIntf(UPI).PPC.lib ANSI C.PPC.Lib PASCAL.PPC.lib AppearanceLib ⇐Note!	MWPRuntime.Lib InterfaceLib MacIntf(UPI).PPC.lib P/ANSI.PPC.Lib P/Rt.PPC.lib AppearanceLib ⇐Note!
2	ToolsPlus.Lib ToolsPlus.p	ToolsPlus.Lib ToolsPlus.CW6&7.PPC.Lib ToolsPlus.p	ToolsPlus.Lib ToolsPlus.CW6&7.PPC.Lib ToolsPlus.p
3	Demo.π.rsrc Demo.p	Demo.π.rsrc Demo.p	Demo.π.rsrc Demo.p

Note: Make sure the AppearanceLib is set to "link weak" to allow your Appearance-savvy app to run on a Macintosh without an Appearance Manager.

You can now build your project and run your application.

Errors when compiling the CodeWarrior demos (or applications):

At the time of this writing, Water's Edge Software has made every effort to ensure that our demo application will compile successfully the first time. Unfortunately, Apple (who produces the C/C++ headers and Pascal interfaces into the Macintosh's toolbox) and Metrowerks development environments are undergoing ongoing revisions. As a result, some inconsistencies may arise between compiler versions. Fortunately, these differences are simple to resolve.

Access Paths

When using Tools Plus for CodeWarrior C/C++ (680x0 or PowerMac), a Pascal runtime library is required. You may have to change the Access Path (in your project's preferences) to locate the required file. This applies to the demo application as well as the project in the "starter files" folder.

File Names

Metrowerks occasionally changes the names of their libraries. If your demo project (or starter project) can't locate a file, remove the problem file from your project then add the equivalent, correctly named file. An Access Path to the correctly named file will be created automatically if it is required.

Link Errors and Warnings

There are several situations where your application may get link errors or warnings. In all cases, it is because of human error in either the files you have added to your project, or the way you have set up your project.

Link error… 16-bit code reference to *'RoutineName'* is out of range.

Make sure that your 680x0 project does not have a segment that exceeds 32K. Start by confirming that each Tools Plus library is in its own segment, then recompile your project. If the problem persists, check your source code files to make sure that no segment exceeds 32K, then recompile. As a last resort, check other libraries to make sure the segments they are in do not exceed 32K. A simpler alternative is to compile your project using a "large" code model (this is set in your project's preferences) and use a single "large" Tools Plus library instead of multiple "small" Tools Plus libraries. Your executable will be larger, but it makes compilation and memory management a little simpler. Large (32-bit) libraries are available in the Tools Plus Developer Kit.

Link error… *'RoutineName'* referenced from *'CallingRoutine'* is undefined.

You have forgotten to include one or more libraries that are needed in your project. Check the instructions in this chapter and add the necessary libraries to your project. If IdleControls is listed as one of the offending routines, then you have intentionally or inadvertently specified that your application will be Appearance Manager-savvy. You can rectify this by adding the AppearanceLib library to your PowerPC project if you want it to be Appearance Manager-savvy, or change your application such that it does not use the Appearance Manager (details on how to do this are provided in the "Designing Your Application" chapter).

Link warning… ignored *'RoutineName'* (descriptor) in FileName, previously defined in *SourceFileName*

This happens if you include either a routine with the same name as one that has already been defined elsewhere, of if you include multiple libraries that both contain the same routine name. If IdleControls appears as one of the offending routines, then you have added the AppearanceLib library to your PowerPC project without telling Tools Plus that your application is Appearance Manager-savvy. You can correct this problem by either removing the AppearanceLib library from your project, or by informing Tools Plus that your application is Appearance Manager-savvy (details on how to do this are provided in the "Designing Your Application" chapter).

Compiling the THINK C (68K) 5, 6 or 7 Demo Application

The easiest way to compile the THINK C (version 5, 6 or 7) demo application included with Tools Plus is to have the following files in the same folder:

Demo.π	ToolsPlus.Lib1	ToolsPlus.Lib5	ToolsPlus.h
Demo.π.rsrc	ToolsPlus.Lib2	ToolsPlus.Lib6	ToolsPlus.c
Demo.c	ToolsPlus.Lib3	ToolsPlus.Lib7	PascalStrHandles.c
	ToolsPlus.Lib4		

Double-click the Demo.π project file to launch THINK C, then run your project.

Your project file keeps track of each file's location as you add it to your project, so you may want to create a new project file after you have put all the Tools Plus libraries and related files in their permanent folders. Create a new project named "Demo.π" in the same folder as "Demo.π.rsrc" and "PascalStrHandles.c". Add the following files to your new project:

Segment 1:	MacTraps
	MacTraps2
Segment 2:	ANSI
Segment 3:	ToolsPlus.Lib1
Segment 4:	ToolsPlus.Lib2
Segment 5:	ToolsPlus.Lib3
Segment 6:	ToolsPlus.Lib4
Segment 7:	ToolsPlus.Lib5
Segment 8:	ToolsPlus.Lib6
Segment 9:	ToolsPlus.Lib7
	ToolsPlus.c
Segment 10:	Demo.c

You can now build your project and run your application.

Note: Make sure you allocate sufficient memory to the debugger if you are going to run the Tools Plus demo application with the debugger on. The Tools Plus demo was written as one large source file, making it easier to compile and study, but making it hungry for debugger memory. Allocate at *least* 500K to the debugger. If you don't have enough memory, turn the debugger off (Project menu, deselect "Use Debugger") when running the demo.

Your applications will likely be written in a more intelligent fashion, abandoning one large source file in favor of several smaller ones.

Errors when compiling the THINK C Demo:

Symantec C/C++ compilers have undergone a series of revisions and some inconsistencies have arisen between compiler versions. Fortunately, these differences are simple to resolve. If your compiler gives you an error that states "argument to function 'x' does not match prototype," it indicates that Symantec has made a minor revision to that function's prototype (in the error message, 'x' will be replaced by the function's name). To correct this error, inspect the offending line in the source file, which is likely a line like:

```
PenPat(&gray);
```

and revise it to match the prototype in the related header file. In the example above, the correction is as simple as changing the line to:

```
PenPat(gray); /* Remove ampersand (&) from the variable */
```

If you have problems getting the demo compiled, see the "Technical Support" chapter for information on how to contact Water's Edge Software for assistance.

Compiling the SPM C/C++ (68K) 8 Demo Application

The easiest way to compile the SPM C/C++ (version 8.0.5 or later) demo application included with Tools Plus is to have the following files in the same folder:

Demo.π	ToolsPlus.Lib1.o	ToolsPlus.Lib5.o	ToolsPlus.h
Demo.π.lo	ToolsPlus.Lib2.o	ToolsPlus.Lib6.o	ToolsPlus.c
Demo.π.rsrc	ToolsPlus.Lib3.o	ToolsPlus.Lib7.o	PascalStrHandles.c
Demo.c	ToolsPlus.Lib4.o	QDGlobals.a.o	

Double-click the Demo.π project file to launch SPM, then run your project.

Your project file keeps track of each file's location as you add it to your project, so you may want to create a new project file after you have put all the Tools Plus libraries and related files in their permanent folders. Create a new project named "Demo.π" in the same folder as "Demo.π.rsrc" and "PascalStrHandles.c". Add the following files to your new project:

Libraries:	<68Kansi-small.o>
	<Interface.o>
	<Mathlib.o>
	<MPW68KRuntime.o>
Tools Plus:	QDGlobals.a.o
	ToolsPlus.c
	ToolsPlus.Lib1.o
	ToolsPlus.Lib2.o
	ToolsPlus.Lib3.o
	ToolsPlus.Lib4.o
	ToolsPlus.Lib5.o
	ToolsPlus.Lib6.o
	ToolsPlus.Lib7.o
Main:	Demo.c

You can now build your project and run your application.

Note: Make sure you allocate sufficient memory to the debugger if you are going to run the Tools Plus demo application with the debugger on. The Tools Plus demo was written as one large source file, making it easier to compile and study, but making it hungry for debugger memory. Allocate at *least* 500K to the debugger. If you don't have enough memory, turn the debugger off (Project menu, deselect "Use Debugger") when running the demo.

Your applications will likely be written in a more intelligent fashion, abandoning one large source file in favor of several smaller ones.

After Compiling:

You will almost certainly get link errors the first time you compile the demo application because the "QDGlobals.a.o" library must be linked first in a Tools Plus application.

SPM automatically creates a link order file named Demo.π.lo and adds it to your project. Open the Demo.π.lo file and move QDGlobals.a.o such that it appears on the first line. Save the changes in the Demo.π.lo file, then build your application again. This time the project will link as expected and build the demo application.

Compiling the SPM C/C++ (PPC) 8 Demo Application

The easiest way to compile the SPM C/C++ (version 8.6 or later) demo application included with Tools Plus is to have the following files in the same folder:

Demo.π	ToolsPlus.Lib	ToolsPlus.h
Demo.rsrc	QDGlobals.c	ToolsPlus.c
Demo.c		PascalStrHandles.c

Double-click the Demo.π project file to launch SPM, then run your project.

Your project file keeps track of each file's location as you add it to your project, so you may want to create a new project file after you have put all the Tools Plus libraries and related files in their permanent folders. Create a new project named "Demo.π" in the same folder as "Demo.π.rsrc" and "PascalStrHandles.c". Add the following files to your new project:

Glue:	<PPCMW_Compatibility.o>	
Runtime:	<InterfaceLib>	
	<MathLib>	
	<AppearanceLib>	⇐Note!
	<str.c>	
Tools Plus:	ToolsPlus.c	
	ToolsPlus.Lib	
Main:	Demo.c	
	QDGlobals.c	
	Demo.rsrc	

Note: Make sure the AppearanceLib is set to "link weak" to allow your Appearance-savvy app to run on a Macintosh without an Appearance Manager.

You can now build your project and run your application.

Compiling the THINK Pascal (68K) Demo Application

The easiest way to compile the THINK Pascal demo application included with Tools Plus is to have the following files in the same folder:

Demo.π	ToolsPlus.Lib1	ToolsPlus.Lib4	ToolsPlus.Lib7
Demo.π.rsrc	ToolsPlus.Lib2	ToolsPlus.Lib5	ToolsPlus.p
Demo.p	ToolsPlus.Lib3	ToolsPlus.Lib6	

Double-click the Demo.π project file to launch THINK Pascal, then run your project.

Your project file keeps track of each file's location as you add it to your project, so you may want to create a new project file after you have put all the Tools Plus libraries and related files in their permanent folders. Create a new project named "Demo.π" in the same folder as "Demo.π.rsrc". Add the following files to your new project:

Segment 1:	Runtime.Lib
	Interface.Lib
Segment 2:	ToolsPlus.Lib1
Segment 3:	ToolsPlus.Lib2
Segment 4:	ToolsPlus.Lib3
Segment 5:	ToolsPlus.Lib4
Segment 6:	ToolsPlus.Lib5
Segment 7:	ToolsPlus.Lib6
Segment 8:	ToolsPlus.Lib7
	ToolsPlus.p
Segment 9:	Sound.p
	Demo.p

You can now build your project. After the initial compile, you will notice THINK Pascal created some additional entries in your project file:

«ToolsPlus.Lib1:1»
«ToolsPlus.Lib2:1»
«ToolsPlus.Lib3:1»
«ToolsPlus.Lib4:1»
«ToolsPlus.Lib5:1»
«ToolsPlus.Lib6:1»
«ToolsPlus.Lib7:1»
«%_MethTables»
«%_SelProcs»
«%_Profiler»

The «ToolsPlus.Lib1:1» through «ToolsPlus.Lib7:1» items contain the object code from the Tools Plus libraries, while «%_MethTables», «%_SelProcs» and «%_Profiler» items are part of THINK Pascal's overhead (consult your THINK Pascal User Manual for details).

Drag «ToolsPlus.Lib1:1» into the same segment as ToolsPlus.Lib1, «ToolsPlus.Lib2:1» into the same segment as ToolsPlus.Lib2, and so on for all the Tools Plus libraries.

Drag «%_MethTables», «%_SelProcs» and «%_Profiler» to any segment that won't be unloaded while your application is running, such as the one containing the Runtime.Lib library.

Even though the project window indicates that Segment 1 exceeds the 32K limit imposed on segments, the project *will* compile and run. When you build your application, the smart linker will strip away unneeded code and significantly reduce the size of this segment.

3 Designing Your Application

Generally, applications that are written with Tools Plus follow the basic structure outlined below. It is also useful to know this when you are writing plug-ins or external code modules which are detailed near the end of this chapter.

1. **#include** or **uses statement:** Your program must be made aware of Tools Plus…

   C — If your C source code file refers to any Tools Plus routines, defines, structures, or type definitions, it must have an “include” statement, such as the one below, at the beginning of the file.

   ```
   #include "ToolsPlus.h"
   ```

   Place ToolsPlus.h as the last file in your #include section (if you don’t care why, skip to the next paragraph). The reason for this placement of ToolsPlus.h is that other headers may already exist or may be subsequently created with routines whose formal parameters coincide with #defines in Tools Plus. Another benefit is that Tools Plus makes your application more tolerant to changes made in Apple’s interfaces. For example, Apple redefined inUpButton to kInUpButtonControlPart. ToolsPlus.h defines inUpButton to the original Apple value, so your source code will continue to work with the original inUpButton or with the newer kInUpButtonControlPart.

   As an alternative to adding `#include "ToolsPlus.h"` to each of your source files, you can add the #include statement to your project’s prefix thereby giving each source file access to the Tools Plus header. “The C Header File” section later in this chapter provides details on how to do this.

   Pascal — If your Pascal source code file refers to any Tools Plus routines, constants, records, or types, it must have a “uses” statement, such as the one below, at the beginning of the file. The uses statement may include other items as well.

   ```
   uses ToolsPlus
   ```

2. **Global Variables:** Declare a global variable of TPEventRecord type. A good name for this variable is *Event*, since it’s used to get events from Tools Plus. If you want to keep your application’s global variable memory to a minimum, you can use the TPEventPointer type, which is a pointer to a Tools Plus event record.

3. **Initialization:** Your application will start by initializing the various managers in the Macintosh’s toolbox, then initializing Tools Plus. See the Initialization chapter for details.

4. **Initial Conditions:** Your application creates its initial conditions for operation. This includes displaying windows, opening files, etc. These are the things your application has to do before responding to any events.

5. **Main Event Handler:** This is where your application responds to events. Events are generated as a result of the user’s actions (typing, clicking, etc.) as well as system actions (refresh a window, inserting a disk, etc.). Typically, after setting up the initial conditions as described above, your application won’t do anything unless it’s in direct response to an event. You write an event handler routine that responds to events, and Tools Plus calls this routine when it has an event. An example of a application’s main event handler routine is provided later in this chapter. To start processing events, your application calls the ProcessEvents routine.

6. **Quitting:** After the user quits your application, certain house cleaning should be done, such as updating and closing files. At the very least, the wrist watch cursor should be displayed to let the user know that the Macintosh is busy while it returns to The Finder.

If you have written Macintosh applications before, then Tools Plus will be a simple transition. You can take an identical approach to your original programming style except:

- You no longer need to call the toolbox’s GetNextEvent or WaitNextEvent routines to get an event.
- You no longer need an event loop. Instead, you just write an event handler routine that looks like one cycle of an event loop inside. Tools Plus will call your routine when it has an event that needs to be processed.
- You use Tools Plus routines to create and maintain your user interface instead of most of the Macintosh toolbox’s routines.

High Level Structure of a Tools Plus Application

At a very high level, all applications that are written with Tools Plus will have a structure that is similar to the following pseudo-code:

`if InitToolsPlus(…) then`	If Tools Plus libraries were initialized…
`  if MyStartupCode then`	If your application's startup code executed without errors…
`    ProcessEvents`	Process events until the user wants to quit. Tools Plus calls your event handlers.

While your application is processing events, at some point it will receive a request to quit, usually coming from the user selecting the File menu's Quit command, or by way of a "quit application" Apple Event. When such a request is made, your application should call a routine that does the following type of work:

- Close unchanged documents
- For every changed document, ask the user if he wants to save changes before quitting (provide the options for saving changes, not saving changes, and canceling the quitting process). If the user has not cancelled quitting…
- Save preference files
- Close remaining windows
- Deallocate dynamic objects like handles, pointers and UPPs
- Call the QuitToolsPlus routine. This instructs the ProcessEvents routine to stop processing events.

In applications (not plug-ins), you don't *have* to close all windows and deallocate dynamic objects before your application quits because MacOS does this automatically. It's just a good housekeeping habit.

A Macintosh Event, in Brief

The following is a *very* brief synopsis of the Toolbox Event Manager, as described in Inside Macintosh. The Toolbox Event Manager is the link between your application and its user. Whenever the user types a key on the keyboard or numeric keypad, presses the mouse button, or inserts a disk in the disk drive, your application is made aware of this by means of an *event*.

In addition to monitoring the user's actions, the Toolbox Event Manager also reports other types of events that serve to inform your application as to "what is happening." Such an event is reported when a partially obscured window is uncovered and its contents need to be redrawn.

The Toolbox Event Manager reports only "bare-bone" events. When a "mouse-down" event occurs, the Event Manager reports the click's location in the screen's global co-ordinates, and the time of the click. It's up to your application to determine which window, and which object in the window was clicked. For those who want to pursue this further, read about the Event Manager in Inside Macintosh.

Fortunately, Tools Plus events are easier to use, and are described later in this manual.

Macintosh Event Queue

As events are generated, they are placed in an *event queue*. When your application is ready to process them, the oldest event is processed first. This journaling mechanism lets the Macintosh remember a series of rapidly occurring events and store them until your application is ready to process them.

Events have a certain *priority*, meaning that some events will be reported before others regardless of when they actually occurred. Their priority is as follows:

1. Activate/deactivate a window
2. Mouse-down/up, key down/up, disk insert, network driver, application-defined events (first in first out)
3. Auto-key (key pressed and held, causing it to repeat)
4. Update a window (refresh a window in front-to-back order)

The priority of events insures that illogical events are not reported. For example, the user may click *twice* in a window's "close away" box before your application gets around to processing the event. The first click will signal your application to close the window. The *second* click will not be reported as a click in a non-existent (closed) window. The click's location will be analyzed *after* the window is closed, and reported accordingly.

Warning: The event queue can store a maximum of 20 events. If your application is so busy that it lets more than 20 events accumulate in the queue, the oldest events will be discarded to make room for the new ones. This may have negative consequences for your application.

Key Up Events

Although events are detailed in Inside Macintosh, and later in this manual in the Event Management chapter, you should be aware that key-up events are ignored by default. If your application needs to be informed when a key is released, the following statement should appear in your application immediately following InitToolsPlus:

```
SetEventMask(EveryEvent);
```

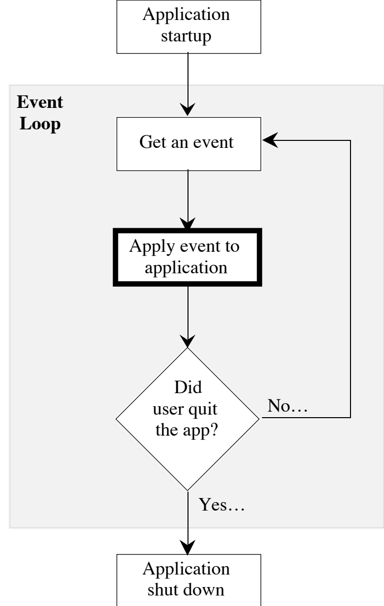
This SetEventMask statement tells the Event Manager to report key-up events as well as all others to your application. Reporting key-up events *may* cause problems in Finder (not MultiFinder) in System 5.x and 6.x since some desk accessories may not expect key up events.

Warning: Your application shouldn't set the event mask to prevent any events other than key-up events.

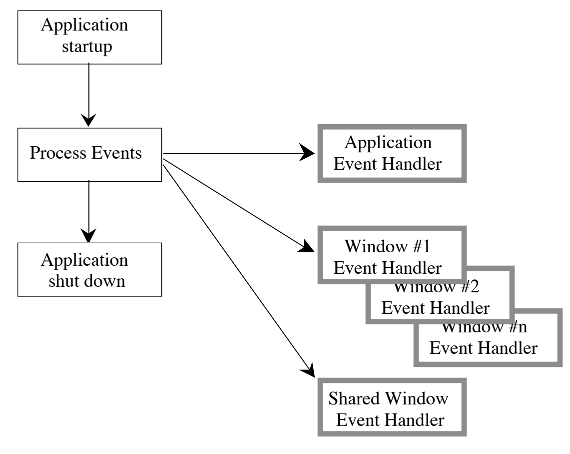
Tools Plus Events, and the Event Loop versus an Event Handler

Tools Plus events are a lot easier to use than the Macintosh's toolbox Event Manager's events. You don't need to get events, and you don't need an event loop. Instead, you write a simple event handler routine that looks like one cycle of an event loop, and Tools Plus calls this routine when it has an event that needs to be processed by your application. The event reported to your event handler routine is already translated into something that is immediately usable. The Event Management chapter details these events.

In a traditional application (below left), the programmer writes an "event loop" that gets an event, applies it to the application, tests to see if the user has quit the application, then loops back to get another event. The task of applying the event to the application becomes increasingly complex as the user interface's complexity grows, and as the number of windows increase. While the event loop approach worked well when Macintosh was introduced in 1984, at a time when graphic user interfaces were simple, it has proved to be lacking in today's state-of-the art applications.



Traditional application with an event loop



Tools Plus application with event handler routines

In Tools Plus, you create an event handler routine that receives Tools Plus events as they need to be processed. You must create a default event handler, also called the application event handler because it receives all events that are not handled by any specialized event handlers. You make Tools Plus aware of the default event handler with the InitToolsPlus routine (see the Initialization chapter for details). You can optionally create an event handler for one or more windows. Events that are specific to a window, such as clicking a button or refreshing the window, are sent to the window's event handler. Although creating an event handler routine and associating it with a window is highly recommended, it is not essential. See the SetWindowEventHandler routine in the Event Management chapter for details on how to assign an event handler routine to a window.

When you design an event handler routine for a window, keep in mind that you can use the same routine to handle events in multiple windows of the same kind. For example, if you are creating a "Search…" dialog, one event handler routine can be used for a dozen concurrently open search dialogs because they all look and behave the same way.

The Event Handler Routine

In a Tools Plus application, you write an event handler routine to respond to events. Like an event loop, your application's main event handler routine is the central hub of your application. A typical event handler has a structure that is similar to the example below. Each window can have its own event handler routine to handle events that are specific to that window. For now, don't concern yourself with the contents of the "Event" record. All aspects of event management are explained by the Event Management chapter of this manual.

```
pascal void MainEventHandler (Ptr CustomDataPtr)
  {
  switch (Event.What)                /*Respond to each type of event…            */
    {
    case doActivate:                 /*User wants to activate the window…        */
      MyActivateRoutine();
      break;
    case doRefresh:                  /*Window needs to be refreshed…             */
      MyRefreshRoutine();
      break;
    case doGoAway:                   /*User clicked window's close box           */
      MyCloseRoutine();
      break;
    case doButton:                   /*User clicked a button…                    */
      switch (Event.Button.Num)      /*Respond to specific type of button…       */
        {
        case kOKbutton:              /*User clicked OK button…                   */
          myOKroutine();             /*                                          */
          break;
        case kCancelButton:          /*User clicked Cancel button                */
          myCancelRoutine();         /*                                          */
          break;
        /*cases for other buttons*/
        }
      break;
    case doMenu:                     /*User selected a menu…                     */
      MyMenuRoutine();
      break;
    case doNothing:                  /*No event available. If your app does any      */
      MyBackgroundRoutine();         /*  background processing, execute one "cycle."  */
      break;

    /*cases for other events*/
    default:                         /*Ignore events that are not listed in the cases */
      break;
    }
  }
```

There are two big differences between an event handler routine as a traditional event loop: (1) a traditional event loop is always testing for a condition that indicates that the user has quit your application, and (2) a polling routine such as the toolbox's WaitNextEvent or Tools Plus's obsolete PollSystem routine, gets an event and returns with a value that indicates if it got an event (true) or not (false). The following pseudo-code shows a traditional event loop:

`while not done do`	Keep getting and processing events until the user quits
`  if WaitNextEvent(myEvent) then`	If you obtained an event…
`    ProcessTheEvent`	Process the event (a case for each event)
`  else`	Otherwise, if an event was not obtained…
`    CallBackgroundProcess`	Execute one cycle of the background process

In the example above, you can think of the ProcessTheEvent routine as an event handler that is called upon to process an event (although Tools Plus events are much easier to work with than traditional toolbox events). As you can see, your event handler routine simply responds to an event. One thing to note is that Tools Plus does not differentiate between "having" an event and "not having" an event, like a polling routine does. In Tools Plus, your event handler routine is called if any event is available, including a null event (i.e., when WaitNextEvent returns with a value of false). The doNothing case in your application's main event handler routine takes care of null events, and is equivalent to "not having" an event.

When reviewing an example of event handler routines in this manual, keep in mind that this is only an *example*, and not a Tools Plus prerequisite. You will probably write a main event handler that is more suitable to your own style of programming and to your application's unique requirements. Your application can also have an event handler for any or all of its windows. See the Event Management chapter for details about event handler routines.

Recursion in the Event Handler Routine

Tools Plus's use of an event handler routine instead of a traditional event loop lends itself to simpler and more structured coding. It also introduces one potential hazard that does not exist in an event loop. In situations where your event handler code takes a while to execute, you will hopefully decide to make your application a good citizen and share the processor with other applications during this lengthy process. Tools Plus provides the Process1EventWhileBusy routine to handle this by reading an event from the Macintosh's Event Manager, processing it internally, and then calling your event handler routine.

The issue that arises is that your application will call Process1EventWhileBusy from within the event handler routine, and Process1EventWhileBusy will likely call your event handler routine to process an event. It is important that you write your event handler routine with awareness of this possibility, and make provisions accordingly. In the case where your application calls the Process1EventWhileBusy, ask yourself "what should this code do if my event handler gets called while this code is executing?" In many cases, your lengthy code will be running in response to a doNothing (idle) event, so it's a simple matter of calling Process1EventWhileBusy and telling it to ignore doNothing events, thus preventing the calling code from being reexecuted. In more complex cases, you may need to set a flag that says "I'm running already, so don't reexecute me." You could accomplish this as follows:

- Create a global boolean named alreadyRunning, and set it to false
- Before you start executing the lengthy process, test to determine if it is already running, as shown in the following pseudo-code:

```
if not alreadyRunning then
  alreadyRunning = true;
  repeat
    do_something;
    Process1EventWhileBusy(true);
  until length_task_is_done;
  alreadyRunning = false;
end;
```

It is fairly easy to avoid problematic situations by exercising a little forethought. Another consideration to keep in mind when calling Process1EventWhileBusy, is that the user can interact with the user interface by doing such things as selecting a pull-down menu, activating another window, or clicking a button. Doing so will certainly resulting in calling your event handler routine (which just called Process1EventWhileBusy). You can avoid this situation by having your application display a wrist watch cursor, thereby preventing user interaction with other user interface elements. Most developers will not want to use this inelegant option, opting instead to prevent recursion in a friendly manner. For example, if a pull-down menu is used to start a lengthy process such as a sort, and your application calls Process1EventWhileBusy during this process, an easy way to prevent recursion of that code is to disable the triggering pull-down menu item until the sort is completed.

System 5 and 6's Finder/MultiFinder, and System 7 and higher

There are some subtle differences between applications that run under Finder (System 5 and 6) and MultiFinder (System 5 and 6) and System 7 and higher. Fortunately, Tools Plus runs under Finder, MultiFinder, and System 7 and higher with minimal consideration on your part. Please note that there is a distinction between Finder and MultiFinder when reading this manual. There are entire chapters dedicated to the Finder and MultiFinder in other books, as well as in THINK Reference. Here is an overview.

Finder

Prior to System 5, the Macintosh had only the Finder to present and maintain its desk top metaphor. Finder lets you run only one application at a time, so your application has access to all the memory the Macintosh has available. Desk accessories (DAs) share the same heap memory with your application, and their windows can be intermingled with windows in your application.

When the user opens or activates a desk accessory, Tools Plus automatically modifies your menus to disable all menus and menu items that don't pertain to the desk accessory: only the File menu's Quit item is enabled, as are the Edit menu's Undo, Cut, Copy, Paste and Clear items. When the desk accessory is closed or your application's window is activated, the menus are automatically restored to their normal settings.

If your application has a tool bar and/or floating palettes, Tools Plus automatically creates a "Desk Accessory Layer" in which all desk accessory windows are kept together to prevent their intermingling with your application's windows. This is done to prevent the confusing condition that arises when the foremost window is a floating palette or tool bar (belonging to your application), behind which is a desk accessory, followed by another window belonging to your application. To the user, it may appear that the palette's operations apply to the desk accessory.

Programming for the Finder is the simplest case, since you can consider your application to be always active and the only application running.

MultiFinder

With the advent of System 5, MultiFinder made cooperative multitasking a reality on the Macintosh. Cooperative, or "switched" multitasking as it is often called, lets several applications run simultaneously by cycling amongst all the tasks. The term "cooperative" is used because each application must cooperate with all others by relinquishing control to give the others some processing time.

Under MultiFinder, the user can launch and run several applications. MultiFinder itself is an application that is always running, busily presenting and maintaining the desk top metaphor. When an application is launched, it is allocated a finite amount of memory that is specified by its SIZE resource.

Only one application can be active (the frontmost window) at a time under MultiFinder, even though, potentially, you may be able to see dozens of windows from multiple applications simultaneously. Therefore, the active application is temporarily "suspended" when another application (or desk accessory) is activated. Please note that suspended applications can also receive events and processing time. Some of the SIZE resource's settings specify how your application behaves when it is suspended or resumed.

Tools Plus takes care of task switching. This includes "minor" switches in which your application gets some processing cycles then allows other applications to do the same, and "major" switches in which your application is either activated or deactivated. The Event Management chapter covers this topic in detail.

Desk accessories are handled slightly differently under MultiFinder in that they don't share memory with your application. Instead, they inhabit the "DA layer." The DA layer is like a single application in which all desk accessory windows exist. Whenever the user clicks a desk accessory, the *DA layer* is activated. Tools Plus takes care of interaction with desk accessories automatically. Whenever a desk accessory is activated, the menu bar is replaced with the DA layer's menu bar, and your application's tool bar and floating palettes are hidden. Your application's menu bar is restored when your application is activated, and the tool bar and floating palettes are displayed.

System 7 and higher

Programming for System 7 and higher is identical to programming for MultiFinder. Each desk accessory, however, behaves like a separate application. Tools Plus takes care of this automatically.

Warning: All MultiFinder and System 7 (and higher) compatible applications must have a SIZE resource. The Completing Your Application chapter details the requirements of the SIZE resource. THINK Pascal users must create their own SIZE resource whereas THINK C/C++ and CodeWarrior users only need to correctly set the settings within their compiler, and it will add the SIZE resource to your application. It is best to always include a SIZE resource in your application.

The C Header file (ToolsPlus.h)

When the Macintosh was originally created, Apple made a strong commitment to Pascal as the “language of choice” for Macintosh programming. Therefore, the Macintosh’s toolbox was designed to work with Pascal calling conventions and Pascal strings.

Fortunately, C allows you to access the Macintosh toolbox’s Pascal functions and procedures from C by declaring prototypes as having Pascal calling conventions:

```
pascal void WindowTitle (short Window, Str255 Title);
```

Pascal Strings versus C Strings

The one thing you should pay particular attention to is Pascal strings. In Pascal, a string is a constant or variable that is from 1 to 255 bytes in length, prefixed with an additional length byte (byte zero). Pascal strings, unlike C strings, are not null terminated. The Str255 structure is available in C to accommodate Pascal-style length-prefixed strings. It is defined in the Types.h header file as:

```
typedef unsigned char Str255[256];
```

When you populate a Pascal string, remember to prefix the string’s text with “\p” to indicate that it is a Pascal string. The example below illustrates this:

```
WindowTitle (18, "\pCustomer Inquiry");
```

You can use the P2CStr and C2PStr routines (defined in the pascal.h header file) to convert Pascal strings to C strings, and C strings to Pascal strings.

Using C and/or Pascal strings in Tools Plus parameters

By default, Tools Plus assumes that you will use only Pascal strings for parameters in Tools Plus routines. This is consistent with the Macintosh toolbox. C/C++ programmers have the option of using Pascal strings only, C strings only, or a mix of either as desired when calling Tools Plus routines. Furthermore, you can make the decision to use Pascal and/or C strings on a per-project basis, or you can set it once and apply those settings to all your projects.

The ToolsPlus.h header and the ToolsPlus.c source file both recognize two #defines that describe how you want to use C strings in parameters to Tools Plus routines:

`TOOLSPLUS_ALLOWS_CSTRINGS`	When set to 1, Tools Plus routines that have a string parameter optionally accept a C string in place of a Pascal string
`TOOLSPLUS_USES_ONLY_CSTRINGS`	When set to 1, Tools Plus routines that have a string parameter accept *only* a C string (Pascal strings are not accepted)

If you want to…	Do the following…
Use Pascal strings only in *all* Tools Plus projects	No changes required (this is the default)
Use C and/or Pascal strings in *all* Tools Plus projects	(1) In the ToolsPlus.h file, un-comment the following line: `#define TOOLSPLUS_ALLOWS_CSTRINGS 1` (2) When you want to use a C string as a parameter in a Tools Plus routine, use the routine name in lower case letters (e.g., use alertbox instead of AlertBox)
Use C strings only in *all* Tools Plus projects	(1) In the ToolsPlus.h file, un-comment the following line: `#define TOOLSPLUS_USES_ONLY_CSTRINGS 1` (2) Use Tools Plus routines as you normally do, except with C strings in place of Pascal strings.
Use Pascal strings only on a per-project basis	No changes required
Use C and/or Pascal strings on a per-project basis	(1) Add the following line to your prefixes*: `#define TOOLSPLUS_ALLOWS_CSTRINGS 1` (2) When you want to use a C string in a parameter in a Tools Plus routine, use the routine name in lower case letters (e.g., use alertbox instead of AlertBox)
Using C strings only on a per-project basis	(1) Add the following line to your prefixes*: `#define TOOLSPLUS_USES_ONLY_CSTRINGS 1` (2) Use Tools Plus routines as you normally do, except with C strings in place of Pascal strings.

*Setting your prefixes

CodeWarrior uses a file to store prefixes (information that is processed at the beginning of each source file). You can set the name of your project's prefix file in your project's Preferences under the Edit menu. The C/C++ Language preferences panel defaults to a prefix file named "MacHeaders.h". Create a prefix file in the same folder as your project and enter its name in the C/C++ Language preferences panel. A good name would be "MyProject.prefix" or something similar. Typically, your prefix file will contain the following two lines:

```
#include <MacHeaders.h>
#include "ToolsPlus.h"   // Optional… may be in your source file instead
```

If you want your CodeWarrior project to use C strings only as parameters in Tools Plus routines, insert a line just before the `#include "Tools Plus"` that reads:

```
#define TOOLSPLUS_USES_ONLY_CSTRINGS 1
```

Symantec C/C++ stores its prefixes as part of the project. The Edit menu's Options lets you set your prefixes (a THINK C or Symantec C++ sub-option may be available). Typically, your prefix file will contain the following two lines:

```
#include <MacHeaders.h>
#include "ToolsPlus.h"   // Optional… may be in your source file instead
```

If you want your Symantec C/C++ project to use C strings only as parameters in Tools Plus routines, insert a line just before the `#include "Tools Plus"` that reads:

```
#define TOOLSPLUS_USES_ONLY_CSTRINGS 1
```

Remember, these prefix changes are required only if you want to specify the use of C and/or Pascal strings on a per-project basis. If you want to use a specific setting for all your projects (such as C strings only), modify the required single line in the ToolsPlus.h header file.

Appearance Manager

Applications written with Tools Plus will run under Mac OS 8 and its Appearance Manager. Tools Plus also provides services that facilitate the development of applications that can run on Macs with the Appearance Manager or without, in the Appearance Manager's "System 7 compatibility" mode (a user interface that looks like System 7) or in its gray scale theme taking advantage of Apple's new windows and controls. In most cases, negligible effort is required to accomplish this compatibility.

Multi-system compatibility with custom window & controls

Even though the Appearance Manager includes great looking windows, floating palettes, 3D buttons, tabs, sliders and other controls, your application will likely need to use custom WDEFs (window definitions) and CDEFs (control definitions) to get similar level of polish when your application runs on a Mac without the Appearance Manager. A floating palette like the attractive Infinity Windoid (included in the Tools Plus Developer Kit) is the most commonly used WDEF. CDEFs that are in the highest demand are 3D buttons, tabs, and sliders, like those found in SuperCDEFs (also included in the Tools Plus Developer Kit).

When you create your application, assign your CDEFs and WDEFs resource IDs that do not conflict with Apple's standard resources. Assigning resource IDs of 128 or higher to your custom definitions is a good idea. Write your entire application such that it uses the custom resources throughout. You accomplish this by using custom procIDs instead of Apple's standard procIDs wherever you want to use a custom window or button. ProcIDs are explained in this manual in the chapters detailing windows, buttons and scroll bars. If you are creating your window layouts using dialogs ('DLOG' resources), create controls ('CNTL' resources) in place of standard push buttons, radio buttons and check boxes because controls let you specify a procID that is different from Apple's standard procIDs.

When you have your application working with the custom WDEFs and CDEFs, modify your application's initialization routine shortly after using InitToolsPlus. Tools Plus's UsingAppearanceManager routine lets your application know if it is running with the benefit of the Appearance Manager. If it is, then Apple's standard window and control procIDs will be mapped to the Appearance Manager's 3D windows and controls. The following sample code shows how you can replace your custom window and control procIDs with Apple's standard procIDs through your application:

```
if UsingAppearanceManager then
   begin
      ReplaceWindowProcID(ordPaletteProc, 1985);
      ReplaceWindowProcID(ordPaletteProc + 2, 1993);
      ReplaceControlProcID(myCheckBoxProc, checkBoxProc);
      ReplaceControlProcID(myRadioButProc, radioButProc);
   end;
```

Using the Appearance Manager

Your application can take advantage of the extended set of user interface elements and services that are offered only in the Appearance Manager. This manual does not detail the Appearance Manager, but it does explain how to prepare your Tools Plus application to use the Appearance Manager, and later, how to make use of many of the new user interface elements. For complete information about the Appearance Manager, please obtain an Appearance Manager SDK (Software Developer Kit) from Apple, or refer to the most recent edition of Inside Macintosh.

Tools Plus 680x0 libraries automatically have access to the Appearance Manager if it exists on the Macintosh that is running your application. This means that all Tools Plus routines that can take advantage of the Appearance Manager will do so if one is available. All of Tools Plus's PowerPC components are set up to default to the same behavior. In all projects generating PowerPC code, you must add the AppearanceLib library into your project, and set the AppearanceLib library to "import weak" (this lets your Appearance-savvy PowerPC application launch on a Mac that doesn't have an Appearance Manager). Your application can use Tools Plus's HasAppearanceManager routine to determine if the Appearance Manager is available. The files you need to support the Appearance Manager should be included with your compiler. They are AppearanceLib (PowerPC stub), Appearance.h (C/C++ header) or Appearance.p (Pascal interface). If these files were not shipped with your compiler, you can get them from Apple.

If you don't have the Appearance Manager files you need to compile your PowerPC application, make sure you do not include the AppearanceLib library in your project. You will also need to remove (or comment out) one line of code in your ToolsPlus.h header file or your ToolsPlus.p interface file, as indicated below:

C
```
#define USE_APPEARANCE_MANAGER 0
```

Pascal
```
{$SETC USE_APPEARANCE_MANAGER := false}
```

Embedding Controls

The Appearance Manager introduces a concept of *control embedding* in which a control becomes a container for one or more other controls. An example of this is a tab control which looks like a panel with multiple tabs across the top, like a paper file folder. Each tab control will likely have one logical "layer" of controls corresponding to each tab at the top of the control. To accomplish this, you create the tab control, then create a "user pane" control (which is invisible) and imbed it into the tab control. The tab control now *owns* the user pane control. Next, create all the controls for the one layer that corresponds to a single tab. Each of these controls falls upon the user pane control, and are automatically embedded into the user pane. The user pane control now owns, say, two list boxes and three check boxes. When you hide a user pane control, all its subcontrols are hidden automatically. This way, you can hide an entire layer of controls with a single routine. Similarly, if you disable the tab control, all the controls contained therein are automatically disabled. At the time of this writing, the following controls are containers that can own other controls: tab control, group box control, placard control, window header control, and the user pane control. Apple may create new controls later than can be container controls.

Tools Plus provides the following routines for embedding: SetAutoEmbed, EmbedButtonInButton, EmbedButtonInScrollBar, EmbedScrollBarInButton, and EmbedScrollBarInScrollBar.

Note: For complete information on Appearance Manager concepts, the Appearance Manager's features, and how to best use the Appearance Manager's new controls, please read the documentation pertaining to the Appearance Manager. It is available from Apple or in the latest issue of Inside Macintosh. This manual does not duplicate that material.

Dialogs and the Dialog Manager

When designing an application that uses Tools Plus libraries, avoid the toolbox's Dialog Manager routines. You may still use alerts, however. The Dialog Manager is effective at creating simple windows, but it quickly reaches its limits when trying to implement the diversity or complexity of features available on most windows seen in today's commercial software. By comparison to Tools Plus, the Dialog Manager is also much more difficult and cumbersome to use.

Tools Plus performs numerous tasks behind the scenes to ensure that all elements of your user interface work together seamlessly, and when the Dialog Manager is introduced into the equation, the simplicity and elegance of Tools Plus can be easily overshadowed by the Dialog Manager's idiosyncrasies.

As a rule, use Tools Plus windows instead of using the Dialog Manager's modal or modeless dialogs. Tools Plus also includes routines that let you use standard resources such as 'DLOG', 'DITL', 'WIND', 'MENU' (etc.) in your application. This gives you all the advantages of Tools Plus without inheriting any of the disadvantages of the awkward and limiting Dialog Manager.

Power Macintosh Performance

Tools Plus libraries are available in native Power Macintosh format, meaning they have been compiled specifically to take advantage of the enhanced performance offered by a PowerPC processor. You may notice, however, that parts of your application do not experience any performance improvements, and in some cases, native Power Macintosh applications may actually be *slower* than their 680x0 counterparts.

One of the reasons for this is that significant portions of the Power Macintosh's System 7 toolbox (written by Apple) are made up of 680x0 code, and are therefore emulated by the PowerPC processor. This phenomenon is similar to

running a standard Macintosh application on a Power Macintosh: it works, but it's not quick. In early versions of System 7, much of QuickDraw was being emulated on PowerPC's resulting in unflattering graphics performance.

We urge you to write your applications using standard Macintosh toolbox calls in spite of the short-term performance degradation. Doing so will help to ensure that your applications continue to run on newer Power Macintoshes, and on new PowerPC processors that will be created in the future. In System 7.6, for example, the Resource Manager was rewritten and became fully PowerPC native, and Mac OS 8 was rewritten to be entirely PowerPC native. This provided additional performance without having to make any changes to your applications.

Writing your own work-arounds may result in an immediate performance improvement while sacrificing compatibility with future Power Macintoshes, or not taking advantage of improvements that will be available as the Power Mac toolbox matures.

Off-screen GrafPorts and GWorlds

If your application creates off-screen grafPorts or GWorlds, which is the case for printing and animation, you can reduce many risks by making sure your grafPort is a valid window when using Tools Plus routines. This is very easy to do with the following code:

1. Before you work on your off-screen grafPort, do the following (shown in C. Pascal coders exclude the ampersand):
```
GetPort(&savedPort);       /*Store current grafPort            */
SetPort(myOffscreenPort);  /*Make your off-screen port current  */
```
2. Perform the necessary work on your off-screen grafPort.
3. When you are finished working on your off-screen grafPort, and before you resume using Tools Plus routines, do the following:
```
SetPort(savedPort);        /*Restore the original grafPort      */
```

Writing Plug-Ins or External Code Modules

Some applications such as Adobe's Photoshop support externally written code modules typically called "plug-ins." They let a developer create a "mini application" that augments the host application's feature set, such as adding an image processor or filter to a graphics application. Within this user manual, we use the term *plug-in* as a generic term for an external code module. Tools Plus for CodeWarrior can be used to write plug-ins. Other Tools Plus libraries can not.

The host application for which you are writing your plug-in determines how your plug-in is structured. Plug-ins are typically structured in one of two ways:

- The host application surrenders control to the plug-in which does all its work including getting and processing events. When the plug-in quits, control is returned to the host application. The plug-in is seen as the "master" while it is running because it gets and distributes events. It may even give events to the host application for processing such as when the host application's windows need updating. We call this a "plug-in master" structure.
- The host application loads the plug-in then gives it commands such as "initialize" or "process this toolbox event." The plug-in does nothing other than respond to commands because the host application gets and distributes events. We call this a "host master" structure.

Before you start writing your plug-in, you first need to obtain a "plug-in development kit" and/or documentation from the host application's author. The kit teaches you how to write plug-ins for that application and it will likely include libraries containing routines to let your plug-in communicate with the host application. Hopefully, it will include information about compiling your source code into a plug-in, and how to make the plug-in accessible to the host application. After you become familiar and comfortable with this information, you can move on to using Tools Plus to write your plug-in.

A plug-in written with Tools Plus sees only its own windows and not those belonging to the host application. Think of the plug-in as a stand-alone application and you will understand how Tools Plus is working. Plug-ins can open as many modal windows as they want because the plug-in runs, for all practical purposes, as an application that is always active. Plug-ins that have a "host master" structure can also open a modeless window providing that it is the plug-in's sole window, but this introduces additional complexities as detailed later.

Tools Plus Plug-In Libraries

The Tools Plus library used in plug-ins, namely "ToolsPlus Plug-In.Lib", is functionally equivalent with the regular Tools Plus libraries except in the following ways:

- The plug-in cannot initialize the Macintosh toolbox. This is a safeguard to prevent double initialization.
- The plug-in cannot allocate more master handle blocks (MoreMasters) because they can't be deallocated later.
- The plug-in has indirect access to the host application's pull-down menus. The Edit menu, for example, is always referenced as menu number 2 (using the mEditMenu constant), regardless of the 'MENU' resource ID used by the host application to create the Edit menu.
- The plug-in's ability to alter the host application's pull-down menus is limited (i.e., no adding or deleting).

680x0 Plug-Ins

The plug-in's "main" routine should be as small as possible, as it will have to fit closely along with several CodeWarrior libraries that were compiled using the small code model (16-bit) addressing. Compile your project with the "multi-segment" option on. Newer CodeWarrior compilers offer an "extended resource" option that should be on. Compile your project using the smart code model (mixed 16-bit and 32-bit), or large code model (32-bit). All these options combined let you create a plug-in that exceeds the 32K code limit for single segment 16-bit code resources.

680x0 plug-ins, like 'CODE' resources, reference their globals by using the A4 register instead of the A5 register that is used by applications. To account for this, your plug-in needs to execute the following code immediately upon entering its "main" routine:

C/C++ plug-ins:

```
long oldA4 = SetCurrentA4();
RememberA4();
```

Pascal plug-ins (oldA4 is declared as a longint):

```
oldA4 := SetCurrentA4;
RememberA4;
```

Just before your plug-in leaves its "main" routine, it must execute the following code:

```
SetA4(oldA4);
```

PowerPC Plug-Ins

A PowerPC plug-in is easier to create than a 680x0 plug-in because its project file does not need special CodeWarrior libraries (although it does need to used a special "ToolsPlus Plug-In.Lib" library), and your code does not have to interact with the A4 register. You also don't need to be concerned with the size of your "main" routine or its placement in proximity to CodeWarrior libraries.

Writing The Plug-In

Your plug-in should be written as a routine that will be called by the host application. The name of this main routine and its parameter list are defined in the plug-in's documentation (available from the host application's author). Your plug-in's main routine can call other routines that are defined in your plug-in, Tools Plus, the Macintosh toolbox, or even the host application providing you have the C/C++ headers and/or Pascal interfaces to those routines. Inside your plug-in's main routine, it will have one of two generic structures, a "plug-in master" or a "host master."

Example of a “Plug-In Master” Structure

1. The host application calls your plug-in (host loses control to your module)
2. Your plug-in must initialize itself by doing the following:
   • 680x0 plug-ins conduct A4 preparation
   • Initialize Tools Plus using InitToolsPlus
   • Initialize your plug-in’s variables
   • Allocate your plug-in’s dynamic objects
3. Your plug-in opens a modal window, such as a dBoxProc or a movableBoxProc, and populates the window with the required user-interface elements using Tools Plus routines. Do not create any menus or alter the host’s menus.
4. Your plug-in calls the ProcessEvents routine which starts Tools Plus’s cycle of getting events, applying them to your plug-in, and calling your event handler routine to respond to events. In more complex cases, your plug-in can inspect an event before Tools Plus processes it. This is done in an event filter routine which is detailed in the Event Management chapter of this manual.
5. When the user quits your plug-in, your plug-in must deinitialize itself by doing the following:
   • Deinitialize Tools Plus (DeinitToolsPlus). This closes any open windows and deallocates dynamic objects.
   • Deallocate the plug-ins dynamic objects
   • 680x0 plug-ins calls the toolbox’s SetA4 routine.
6. Your plug-in terminates and the host application regains control.

Example of a “Host Master” Structure

Your plug-in’s “main” routine needs a case for each command that can be invoked by the host application. Here are some examples of commands that can be issued by the host application and how your plug-in would respond to them:

Initialize: Your plug-in initializes itself by doing the following…
   • Initialize Tools Plus using InitToolsPlus
   • Allocate your plug-in’s dynamic objects
   • Initialize your plug-in’s global variables

Deinitialize (or quit): Your plug-in deinitializes itself by doing the following…
   • Deallocate your plug-in’s dynamic objects
   • Deinitialize Tools Plus using DeinitToolsPlus

Process an event: Your plug-in processes the supplied toolbox event (detailed in the Event Management chapter)…

```
ProcessToolboxEvent(&theEvent);
```

Deactivate modeless window: Your plug-in tells Tools Plus to become “suspended,” thereby deactivate the window by doing the following…
   • Define a variable of EventRecord type
   • Set the “what” field to osEvt (15) and the “message” field to $01000000
   • Call the same code you do in response to the “process an event” command (above)

Activate modeless window: Your plug-in tells Tools Plus to “resume,” thereby activate the window by doing the following…
   • Define a variable of EventRecord type
   • Set the “what” field to osEvt (15) and the “message” field to $01000001
   • Call the same code you do in response to the “process an event” command (above)

doManualEvent in a Plug-In

Most applications can ignore Tools Plus’s doManualEvent event, but chances are that your plug-in will need to respond to them. Whenever Tools Plus detects an event in a foreign window, that being one that was not created with a Tools Plus routine, it reports it to your plug-in as a doManualEvent event. Your host application’s windows fall into this category. The most common occurrence of this is when your host application needs to have a window refreshed, in which case the raw toolbox event record within Tools Plus’s event record will report an update event and a pointer to the target window.

Be certain that your plug-in responds appropriately to this situation by informing the host application that one of its windows needs to be updated. If your plug-in does not do this, it will stop receiving doNothing events (null events) while the Window Manager frantically reports the need to update the target window.

Note: Plug-in specifics vary from one host application to another. Water's Edge Software can assist you with queries and information about Tools Plus, but we don't have expertise for all the possible host applications that support plug-ins.

What to read next

A synopsis of each chapter can be found at the beginning of each section of this manual. Familiarize yourself with the basic concepts in all the remaining sections before you start programming. You may then want to learn about the intricacies of each Tools Plus routine.

Devote considerable attention to the chapter on Event Management. It explains task switching, and details each kind of event that can be reported by Tools Plus, as well as how to respond to those events. You will be better equipped to design your application when you know what to expect in your event handler routine.

The *Special Routines* section lists Macintosh Toolbox routines that require special attention, in that they should be used with caution, or not at all. Using some of these routines will interfere with Tools Plus's normal operations, whereas other routines are obsolete by Tools Plus's wealth of services and features.

The section on Completing Your Application is not news to Macintosh programming veterans. It is there for the benefit of new developers to help them finish their application and make it a double-clickable program. The SIZE resource and its required settings are detailed there.

4 Initialization

All Macintosh applications begin with very similar code that is needed to initialize the Macintosh toolbox's various managers. This code must be executed at the beginning of your application before doing anything else. You can optionally have Tools Plus do this for you in the InitToolsPlus routine.

C
```
InitGraf(&qd.thePort);                    /* Initialize Macintosh toolbox…   */
InitFonts();                              /*  (can be done by InitToolsPlus) */
InitWindows();                            /*                                 */
InitMenus();                              /*                                 */
TEInit();                                 /*                                 */
InitDialogs(0L);                          /*                                 */
SetApplLimit(value of A7 - stack size);   /*Set stack size (details later)   */
                                          /*Initialize Tools Plus…           */
if (InitToolsPlus(&Event, &MyEventHandler, &MyEventFilter, 10, 5, initTE32KBuffer,
       initUseColor)){
```

See the InitToolsPlus routine for details on initializing Tools Plus.

Pascal
```
InitGraf(@qd.thePort);                    {Initialize Macintosh toolbox…     }
InitFonts;                                {  (can be done by InitToolsPlus)  }
InitWindows;                              {                                  }
InitMenus;                                {                                  }
TEInit;                                   {                                  }
InitDialogs(nil);                         {                                  }
SetApplLimit(value of A7 - stack size);   {Set stack size (details later).   }
                                          {Initialize Tools Plus…            }
if InitToolsPlus(@Event, @MyEventHandler, @MyEventFilter, 10, 5, initTE32KBuffer,
       initUseColor) then
```

See the InitToolsPlus routine for details on initializing Tools Plus.

THINK Pascal performs all initialization automatically providing you leave the “Initialization” compiler directive on (this is the default). All you need to do is initialize Tools Plus with InitToolsPlus at the beginning of your application. THINK Pascal's automatic initialization finishes off by doing the following additional tasks:

```
MaxApplZone;
for i := 1 to 10 do
  MoreMasters;
```

If you turn the Initialization directive off by adding `{$I-}` before your `begin` statement in your main program, you will have to initialize the various toolbox managers yourself or let InitToolsPlus do it for you.

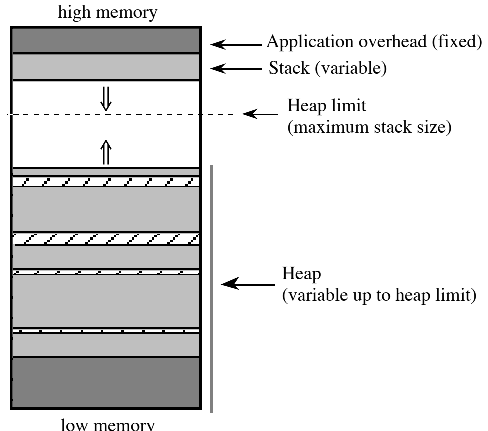
Note: THINK Pascal users as well as older versions of Symantec's C/C++ don't use the new C/C++ Universal Headers and Universal Pascal Interfaces (UPIs), so you will need to use `thePort` in place of `qd.thePort` in the InitGraf routine.

Your application initializes Tools Plus libraries with the InitToolsPlus routine. Call InitToolsPlus as early as possible since InitToolsPlus creates unrelocatable objects in memory (pointers), and doing so at the the start of your application eliminates memory fragmentation. In 680x0 applications, InitToolsPlus also loads all the ‘CODE’ segments containing Tools Plus libraries into memory where they remain as long as your application is running.

Stack and heap

If you are already familiar with the stack and the heap in your application, you can skip this section and just review the Set68KStackSize and ChangeStackSize routines at the end of this chapter. This section is a very condensed description of the stack and heap. Most applications will not need to concern themselves with these details.

The total amount of memory that is available to your application is shared by objects your application creates and controls, and by things that happen automatically when your application is running.



Application's total memory

When your application is first launched, QuickDraw globals and other fixed application overhead is allocated high in memory. These are things that are fixed in sized and are automatically maintained by the various toolbox managers.

Your application's **stack** is also automatically maintained, but it is affected by things that are happening in your application. The stack is a "last in first out" (LIFO) queue and contains temporary information only like local variables and return addresses. When a routine is called, the stack temporarily grows by a certain amount. The stack is reduced by the same amount when the routine returns control to the caller. Routines that have lots of local data (such as Str255 local variables) and recursive routines consume more stack space. The stack starts from high memory and grows downward.

The **heap** contains your executable code, resources that are loaded into memory, and other dynamically allocated objects. The heap is populated starting from low memory and grows upward as required. You can think of this as your application's memory.

Macintosh's Memory Manager allows the heap to grow up to a specified limit that is set when your application is launched. The **heap limit** is commonly referred to as the *stack size* because this limit not only defines the maximum size to which the heap can grow, but it also defines the maximum size of the stack before a collision occurs between the stack and the heap.

It is important that you define a large enough stack space because although the heap won't grow beyond the heap limit, the stack grows as required and may collide with the heap. If this happens, objects in your heap can become corrupted. The Memory Manager has a collision sniffer that causes a system error 28 when the stack moves into application heap. Unfortunately, the sniffer only checks the stack sixty times a second and can miss quick stack transgressions.

All Power Macintosh compilers and THINK Pascal let you set your stack size (and heap limit) from your development environment. The 680x0 THINK C/C++ compilers and the 680x0 CodeWarrior compilers do not have this facility built into the development environment, so you can use the Tools Plus routines Set68KStackSize or ChangeStackSize.

Although the Memory Manager allows the heap to grow upward to the heap limit, InitToolsPlus calls the toolbox's MaxApplZone routine thus forcing the heap to grow to its maximum size. This is a good practice because it makes the most use of the available memory, reduces memory fragmentation, and reduces purging and subsequent reloading of purgeable objects in the heap. This approach also makes stack collisions with the heap show up while your application is still under development where you can remedy the problem.

Note: Most applications will do perfectly well with the default heap limit setting. If you suspect your application's stack is getting close to the limit during stand-alone execution, use the toolbox's StackSpace routine during development to determine the amount of unused space available to the stack, then increase the stack space if required.

Other application initializing activities

After your application has initialized the Macintosh toolbox (optionally done by InitToolsPlus), and it has initialized Tools Plus (using the InitToolsPlus routine), it may need to carry out a number of other activities that are associated with starting up an application, such as:

- Opening preferences and settings files
- Searching directories for optional files
- Creating temporary work files
- Allocating dynamic objects (using pointers and handles)
- "Personalizing" your application the first time it is run (the user must enter their name and a serial number)
- User name and password

Be careful to avoid calling any event processing routines during your application's initialization code because doing so may inadvertently process the "open application", "open documents" or "print documents" Apple Events before your application is ready to deal with them. The Tools Plus routines to avoid during your startup code are Process1EventWhileBusy, ProcessToolboxEvent, AlertBox and AlertBox3. If you must call any of these routines, you'll need a global "application is ready to process events" flag that is set to true only when your application's initialization is complete, and your main event handler's and window event handlers' code is bypassed if the flag is not set.

InitToolsPlus

Initialize Tools Plus.

C
```
pascal Boolean InitToolsPlus (Ptr Event, ProcPtr MyEventHandler,
             ProcPtr MyEventFilter, short MoreHandles, short MaxWindows,
             short TEBufferSize, long InitSpec);
```

Pascal
```
function InitToolsPlus (Event: PTR; MyEventHandler ProcPtr;
             MyEventFilter ProcPtr; MoreHandles: INTEGER; MaxWindows:
             INTEGER; TEBufferSize: INTEGER; InitSpec: LONGINT): BOOLEAN;
```

This routine initializes variables and records that are required by Tools Plus. It must be called once at the beginning of your program.

Event is the address of your global Tools Plus event record. Tools Plus populates this record with event information whenever Tools Plus reports an event to your application. You should define a global Tools Plus event record (of TPEventRecord type) for your application, then use this record throughout your application. In C/C++ applications, the address of your event record is passed as &Event, assuming that the global variable is named "Event." In Pascal it is passed as @Event.

MyEventHandler is the address of your application's main (default) event handler routine. Tools Plus calls this routine to respond to Tools Plus events. In C/C++, simply enter your event handler routine name. In Pascal, preface your routine name with "@" to pass the address. Internally, InitToolsPlus allocates and uses a UPP if required. The UPP is deallocated when DeinitToolsPlus is called. Your event handler routine has the following C/C++ prototype or Pascal interface:

C
```
pascal void MyEventHandler (Ptr CustomDataPtr)
   {
   }
```

Pascal
```
procedure MyEventHandler (CustomDataPtr: Ptr);
   begin
   end;
```

Tools Plus always passes a nil custom data pointer. See the Event Management chapter for details on how to write your event handler routine.

MyEventFilter is the address of your application's event filter routine. Use nil if your application does not have an event filter routine. Tools Plus calls this routine to process a toolbox event *before* it is passed to Tools Plus for processing. In C/C++, simply enter your event filter routine name. In Pascal, preface your routine name with "@" to pass the address. Internally, InitToolsPlus allocates and uses a UPP if required. The UPP is deallocated when DeinitToolsPlus is called. Your event filter routine has the following C/C++ prototype or Pascal interface:

C
```
pascal Boolean MyEventFilter (EventRecord *theEvent)
  {
    /* Inspect and possibly modify the toolbox event record */

    return(1);                                  /*Should Tools Plus process the event?        */
  }
```

Pascal
```
function MyEventFilter (var theEvent: EventRecord): Boolean;
  begin
    {Inspect and possibly modify the toolbox event record}

    MyEventFilter := true;                      {Should Tools Plus process the event?          }
  end;
```

See the Event Management chapter for details on how to write your event filter routine.

MoreHandles specifies the number of additional "Handle Blocks" that are created during initialization. THINK C, and CodeWarrior compilers don't automatically create any handle blocks, while THINK Pascal automatically creates 11 blocks, each containing 64 handles and consuming 512 bytes per block. The number of additional blocks created during Tools Plus initialization is specified by *MoreHandles*, which can have a value between 0 and 128. In THINK Pascal, the default number of blocks is usually enough, so you will specify 0. In THINK C and CodeWarrior compilers, a dozen blocks is usually enough.

MaxWindows declares the maximum number of windows that may be simultaneously open in your application. This number should be kept to a realistic minimum, since a small amount of memory (less than 300 bytes x *MaxWindows*) is consumed, regardless if a window is ever opened or not. InitToolsPlus allocates one window record for each window specified by *MaxWindows*, plus one additional window that is used exclusively by Tools Plus. This preallocation of window records is done in order to eliminate memory fragmentation. It is a small price to pay, in terms of memory consumption, to prevent memory fragmentation, and it carries no negative side effects.

Although Tools Plus supports up to 250 windows, Mac OS slows down progressively as more windows are opened. The realistic limit, due to Mac OS performance and to the user's adverse experience in managing too many windows, is somewhere around thirty open windows. If you allow a limit of 50 windows, the user will perceive this to be an "unlimited" number of windows.

TEBufferSize specifies the size of text editing buffers (maintained by Tools Plus) used for cutting, copying, pasting, and storing copies of text for the Undo/Redo feature. Use a value of 255 to 32767, which represents the largest field you will have in your application. When you reduce this figure, you can conserve as much as 128K of your application's memory, but you limit the size of text that is copied or pasted. For example, if you set TEBufferSize to 255, only the first 255 characters of the clipboard are pasted into your application's fields. Conversely, you can select several hundred characters of text typed into your application's field, and when you select the Edit menu's Copy command, only the first 255 characters are copied to the clipboard. The constants *initTEStr255Buffer* and *initTE32KBuffer* are provided for your convenience. If you are unsure what value to give, use initTE32KBuffer and allocate plenty of memory to your application.

InitSpec specifies various Tools Plus initialization options. The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result.

Optionally choose only one of the following Color QuickDraw options…

`initUseColor`	Use Color QuickDraw if it is available on the Macintosh running your application. If your application is running on a Macintosh SE or higher with a color or gray-scale monitor, Tools Plus will take advantage of the available colors or shades of gray. Tools Plus will still run perfectly if Color QuickDraw is unavailable. Tools Plus uses Color QuickDraw by default, so you can omit this option.

`initIgnoreColor` — Ignore Color QuickDraw. All drawing is done in black and white. This option saves memory and simulates your application running on a Macintosh without Color QuickDraw. Note that Tools Plus does not patch the system, and in some cases the system may draw some objects, like highlights in list boxes and editing fields, using color.

Optionally choose only one of the following TextEdit scrap options…

`initUseTEScrap` — Tools Plus creates and maintains a local TextEdit[1] scrap. This is more costly in terms of memory (often 32K or more), and is necessary only if your application has editing fields created by anything other than Tools Plus. At initialization, Tools Plus copies the desk scrap[2] to the TextEdit scrap.

`initIgnoreTEScrap` — Tools Plus does not create a local TextEdit scrap. Instead, it works directly with the desk scrap. This option can save you about 32K of application memory, providing you use only Tools Plus fields or no editing fields at all.

Optionally choose only one of the following desk scrap options…

`initDontUnloadDeskScrap` — Do not unload the desk scrap to disk when initializing Tools Plus. If this option is *excluded*, InitToolsPlus first determines if the amount of free memory is dangerously low (which can happen if a particularly large object was copied to The Clipboard by another application). If memory is low (around 90K or less at startup), the desk scrap is copied to disk thereby freeing up that memory in your application.

`initUnloadDeskScrap` — Always unload the desk scrap to disk when initializing Tools Plus, even if there is ample memory. This option makes your application more memory conservative. Before your application quits, remember to load the scrap back using LoadScrap.

Optionally choose any of the following options…

`initFasterWinDrag` — When the user clicks on an inactive window's title bar to both activate it and drag it to a new position, Tools Plus normally activates the window, refreshes it if required, then lets the user drag it. Add this option if you need faster performance, and windows will refresh *after* the user finishes dragging them.

`initMacToolbox` — The InitToolsPlus routine initializes the Macintosh's toolbox before it does anything else. Toolbox initialization is performed by calling the following routines as described at the beginning of this chapter:

```
InitGraf(@qd.thePort);
InitFonts;
InitWindows;
InitMenus;
TEInit;
InitDialogs(nil);
```

If you use this option, make sure that you do not initialize the toolbox in your own code. THINK Pascal initializes the toolbox automatically by default (you can turn automatic initialization off by adding `{$I-}` just before the `begin` statement in your main program).

`initInheritHelp` — If a user interface element does not have the required Balloon Help message, Tools Plus should search the parent object for the Help message. The default is to exclude this option so that if a Help message is missing for an object, no Help message is displayed. See the Balloon Help chapter's "Help Inheritance" section for details.

`initReleaseResources` — When this option is used, Tools Plus calls ReleaseResource for any purgeable and unlocked resources that it uses, such as icons and pictures. This keeps

---

1 The TextEdit scrap is a local copy of the desk scrap. It contains only text data, ignoring images and other kinds of data. Text scrap is necessary only if your application uses editing fields that are not created by Tools Plus.

2 Desk (or System) scrap is equivalent to The Clipboard in that it is used to transfer text, images, or other kinds of data between applications and desk accessories.

memory as free as possible but may result in reduced performance, especially when large resources are used in conjunction with slow media like floppies or CD-ROMs.

By default, Tools Plus does not purge resources from memory. This results in better performance because the first time a resource is accessed, it is loaded from disk into memory and it stays there until there is a memory shortage. In that case, when a new resource is loaded from disk, one or more old resources are purged from memory and will be automatically loaded from disk the next time they are accessed. The only negative impact to the default behavior is that more master pointers (see InitToolsPlus) are required to permanently reference all accessed resources.

`initAutoSaveFieldString` Automatically save the user-edited text in an active editing field before it is deactivated. This option makes for much simpler coding, but it does not give your application the opportunity to validate fields' contents on a field-by-field basis. Instead, your application can edit the fields' contents in a batch when the user clicks the OK button to process the entire window.

`initAutoFocusChanges` Automatically let the user tab to the next/previous active editing field, and automatically process the user's click to an inactive (but enabled) editing field. The same applies to other user interface elements that accept the keyboard focus. This option makes for much simpler coding, but it does not give your application the opportunity to validate fields' contents on a field-by-field basis. When this option is used, the initAutoSaveFieldString option is automatically used too to ensure that user-edited text in active editing fields is saved and therefore not inadvertently lost.

`initAppearanceManagerSavvy`

This option works only if the Appearance Manager is available, otherwise it is ignored. When you use this option, Tools Plus automatically substitutes standard user interface elements for their equivalent elements in the Appearance Manager. This lets you design your application for Macs that don't have an Appearance Manager, and have the same application make use of the Appearance Manager's 3D user interface elements if they are available. The controls that are affected by this option are:

- standard push buttons (pushButProc)
- check boxes (checkBoxProc)
- radio buttons (radioButProc)
- scroll bars (scrollBarProc).

All variants for these controls are also converted to the Appearance Manager's controls. The windows that are affected by this option are:

- standard document window (noGrowDocProc) with or without zoom box
- growable document window (documentProc) with or without zoom box
- standard modal dialog window (dBoxProc)
- plain dialog window (plainDBox)
- alternate, or shadowed dialog window (altDBoxProc)
- movable modal dialog window (movableBoxProc)
- dynamic alerts assume an appropriate Appearance Manager theme

The menu bar and all menus, including the lists in pop-up menus, all use the Appearance Manager's 3D menu definition. List boxes use the Appearance Manager's 3D scroll bars.

This option must be used if you want your list boxes, fields and pop-up menus to take advantage of the Appearance Manager's new controls.

`initPureAppearanceManager`

This option removes a number of inconsistencies between non-Appearance-Savvy and Appearance-Savvy applications, specifically:

- Custom colors are not applied to pull-down menus because the Appearance Manager's theme takes care of all coloring
- Custom colors are not applied to pop-up menus because the Appearance

Manager's theme takes care of all coloring

- Custom colors are not applied to any controls because the Appearance Manager's theme takes care of all coloring
- On dialogs, 'icon items are translated into non-selectable icon controls that dim on an inactive window.
- On dialogs, 'picture' items are translated into non-selectable picture controls that dim on an inactive window.

If the initPureAppearanceManager option is used, it automatically turns on the initAppearanceManagerSavvy option.

`initAllWindowsHaveBackgroundTheme`
This option works only if the Appearance Manager is available, otherwise it is ignored. When this option is used all windows you create using Tools Plus routines are assigned an appropriate background theme. Alternatively, you can assign background themes to individual windows.

Optionally choose only one of the following Live Window dragging and resizing options…

`initLiveWindowDrag` *Drag and resize windows in real time. By default (when this option is not used), a dotted outline tracks the window while the mouse button is down, then, when the user releases the mouse button, the window is moved or resized. This option looks best on faster Macintoshes like G3s.

`initLiveWindowDrag040` *Drag and resize windows in real time if your application is running on a Macintosh with an '040 processor of faster. By default (when this option is not used), a dotted outline tracks the window while the mouse button is down, then, when the user releases the mouse button, the window is moved or resized. This option looks best on faster Macintoshes like G3s.

`initLiveWindowDragPPC` *Drag and resize windows in real time if your application is running on a Macintosh with a PowerPC processor of faster. By default (when this option is not used), a dotted outline tracks the window while the mouse button is down, then, when the user releases the mouse button, the window is moved or resized. This option looks best on faster Macintoshes like G3s.

*See the SetLiveWindowDragging routine in the Windows chapter to turn live window dragging on or off under your application's control.

InitToolsPlus returns a value of *true* if initialization was successful, otherwise *false* is returned.

There are a number of other Tools Plus routines that can help your application ascertain its runtime environment, such as SystemVersion (what is the system version on the Mac that is running your application), HasAppearanceManager (is an Appearance Manager available), and HasAppearanceManagerRoutines (is an Appearance Manager available and does your application have access to its routines). You may want to take a few moments to peruse the Miscellaneous Routines chapter to see what is available.

Note: The initAppearanceManagerSavvy option may interfere with some visual aspects of your development environment while you are programming with an Appearance Manager running, and with System Wide Platinum Appearance turned off. This does not occur in the final application. In THINK Pascal, for example, standard Mac OS controls, windows and menus will sometimes or partially take on the Appearance Manager's theme. Make sure that your application calls DeinitToolsPlus before quitting to help eliminate this. Keep in mind that this anomaly does not occur in the final double-clickable application.

Warning: By default, THINK Pascal automatically initialized the Macintosh toolbox and allocates 11 master handle blocks as detailed at the beginning of this chapter. Do not include the `initMacToolbox` option in the InitToolsPlus routine unless you have instructed THINK Pascal not to automatically initialize the toolbox by including `{$I-}` just before the `begin` statement in your main program. Initializing the toolbox twice will certainly cause errors and crashes.

Initialization Failure

InitToolsPlus will fail initialization for only two possible reasons: [1] a severe memory shortage exists, under which Tools Plus cannot allocate sufficient memory for the additional handles or for its own structures, or [2] your application is trying to run on a Macintosh with ROMs older than version 117. Those are the old 64k ROMs found in the Macintosh 128K, Macintosh 512K, and Lisa (Macintosh XL). This will not occur in the Macintosh 512KE (enhanced), or higher. Tools Plus will display an appropriate alert if the wrong ROMs are used.

If initialization fails, do not attempt to use any Tools Plus routines.

Other Initialization

InitToolsPlus also performs other initializing tasks that your application would normally have to do:

- All events are automatically flushed (cleared) from the event queue, such as keystrokes which may have been typed from The Finder.
- The “random number seed” is initialized to ensure that your application does not generate an identical pseudo-random numeric sequence each time it runs.
- MaxApplZone is called to expand the application’s heap zone to its limit.

The Cursor

The cursor is displayed as a wrist-watch when your application is launched by The Finder, and will remain as such until it is changed by your application. Note, however, that Tools Plus behaves as though a normal cursor is displayed in that it does not filter out clicks and typing (such as when your application sets the cursor to a wrist-watch).

```
CONST                                                   {Color QuickDraw…                         }
       initUseColor            = $00000000;             {Use Color QuickDraw if available         }
       initIgnoreColor         = $00000001;             {Don't use Color QuickDraw                }

                                                        {TextEdit Scrap…                          }
       initUseTEScrap          = $00000002;             {Use TextEdit scrap                       }
       initIgnoreTEScrap       = $00000000;             {Don't use TextEdit scrap                 }
       initDontUnloadDeskScrap = $00000004;             {Don't unload desk scrap                  }
       initUnloadDeskScrap     = $00000008;             {Unload desk scrap to disk                }

       initFasterWinDrag                 = $00000020;   {When activating a window, drag before    }
                                                        {  refreshing for faster performance.     }
       initInheritHelp                   = $01000000;   {Inherit Help messages from parent object }
       initReleaseResources              = $02000000;   {Release resources when done              }
       initPureAppearanceManager         = $08000000;   {Adhere to 'pure' Appearance Manager      }
                                                        {  principles (no color controls or menus)}
       initAutoSaveFieldString           = $00000080;   {Automatically save edited text when a    }
                                                        {  field is deactivated.                  }
       initAutoFocusChanges              = $00000400;   {Automatically tab or click to the new    }
                                                        {  keyboard focus.                        }
       initAppearanceManagerSavvy        = $20000000;   {Convert standard ProcIDs to Appearance   }
                                                        {  Manager's procIDs.                     }
       initAllWindowsHaveBackgroundTheme = $10000000;   {Fill Appearance-savvy windows with       }
                                                        {  background theme.                      }

       initMacToolbox          = $80000000;             {Initialize Mac toolbox (InitGraf etc.)   }

       initLiveWindowDrag      = $00002000;             {Drag/resize windows in real time…        }
       initLiveWindowDrag040   = $00001000;             {   …only if running on an 040 or better  }
       initLiveWindowDragPPC   = $00000800;             {   …only if running on a PowerMac        }

                                                        {TextEdit and Undo/Redo buffer size:      }
       initTEStr255Buffer      = 255;                   {255 characters                           }
       initTE32KBuffer         = 32767;                 {32K characters (maximum)                 }
```

DeinitToolsPlus

Deinitialize Tools Plus.

C
```
pascal void DeinitToolsPlus (void);
```

Pascal
```
procedure DeinitToolsPlus;
```

Under rare circumstances, you may want to "deinitialize" Tools Plus, that being to deallocate objects that were dynamically allocated in your application's heap when Tools Plus was initialized. An example of when you would use DeinitToolsPlus is if you are writing a plug-in using Tools Plus. The normal sequence of operations is as follows:

1. The host application (i.e., Photoshop) calls a plug-in that is written with Tools Plus.
2. As the plug-in is loaded, it consumes some stack space for its global variables including Tools Plus's globals that consume about 2K.
3. The plug-in calls InitToolsPlus which dynamically allocates some structures that are used globally throughout Tools Plus.
4. The plug-in opens a modal window and creates its user interface.
5. The plug-in processes events until the user dismisses the dialog.
6. The user dismisses the plug-in's dialog.
7. The plug-in closes the window.
8. The plug-in calls DeinitToolsPlus to deallocate Tools Plus's dynamically allocated structures and leave the application's heap in the same condition it was found before InitToolsPlus was called.

The next time Photoshop calls the plug-in, the same series of steps are executed.

DeinitToolsPlus starts off by disposing of any color cursor information you may be using. It then closes all Tools Plus windows, thereby deleting all the objects in the windows and reclaiming their memory. The window records that were created by InitToolsPlus are then deallocated. Next, all cursor tables are deleted along with their zones, then finally the remaining Tools Plus dynamic objects are deallocated.

Pull-down menus and hierarchical menus are not deleted by DeinitToolsPlus because in all likelihood, they were created by the host application that calls your plug-in. Use the Tools Plus routines RemoveMenus or RemoveAllMenus if you want to delete menus and reclaim their memory.

Applications that use DeinitToolsPlus will likely pass a value of zero (0) in the MoreHandles parameter when calling InitToolsPlus. This is because the MoreHandles parameter specifies the number of master pointer blocks that are allocated by InitToolsPlus. There is no way to deallocate these master pointer blocks, so it's best not to create a new set each time that InitToolsPlus is called.

Normally, you never have to call DeinitToolsPlus because when an application quits, it destroys all the objects that existed in the application's heap, and Tools Plus's dynamically allocated objects are automatically destroyed along with your application.

Set68KStackSize

Set the maximum stack size in a 680x0 application.

C
```
pascal void Set68KStackSize (long Bytes);
```

Pascal
```
procedure Set68KStackSize (Bytes: LONGINT);
```

Power Macintosh compilers and THINK Pascal let you specify your application's stack size within your development environment. The remaining compilers default to a (usually) safe limit. You can use Set68KStackSize to define the stack's maximum size and thereby prevent the heap from growing beyond that point. This routine does not do anything when compiled into a native Power Macintosh application because all Power Macintosh compilers let you define this setting from within your development environment.

Bytes indicates the maximum size of your application's stack in bytes.

Use Set68KStackSize at the end of your toolbox initialization, likely instead of the toolbox's more complex SetApplLimit routine. Set68KStackSize must be executed before InitToolsPlus. You don't need to call MaxApplZone because it is called by InitToolsPlus. Most applications will not need to use this routine.

ChangeStackSize

Change the maximum stack size in an application.

C
```
pascal void ChangeStackSize (long Bytes);
```

Pascal
```
procedure ChangeStackSize (Bytes: LONGINT);
```

This routine increases or decreases the stack's maximum limit by the indicated number of *Bytes*. Positive numbers increase the stack's maximum size while negative numbers decrease it. Your application can use ChangeStackSize if you know that your application will temporarily need to increase the stack's size, likely due to calling a recursive function or a function with lots of local data. Most applications will not need to use this routine.

☞ **Warning**: There is no guarantee that you will be successful in increasing the stack size. The heap may be very full or it may contain locked or unrelocatable objects that prevent it from being reduced in size, and thereby preventing the safe advance of the stack. Call the toolbox's StackSpace immediately after calling ChangeStackSize to determine how much memory is available for the stack to grow.

SetParamRangeErrProc

Set the parameter range error action routine.

C
```
pascal void SetParamRangeErrProc (ProcPtr userRoutine);
```

Pascal
```
procedure SetParamRangeErrProc (userRoutine: ProcPtr);
```

By default, when you call a Tools Plus routine with a parameter that is out of range, such as attempting to open window number 11 after you have initialized Tools Plus to allow a maximum of only 10 windows, Tools Plus delivers an alert that states "Error: Parameter passed to a Tools Plus routine is not within the legal range of values." To facilitate debugging, you can install your own action routine that will be called instead of displaying the parameter range alert. If you have a stop point in this routine, you can step out of the routine line by line and eventually return to the offended Tools Plus routine.

UserRoutine is the address of an action proc that is called by Tools Plus instead of displaying the parameter range alert.

If you decide to use this routine, do so shortly after initializing Tools Plus to eliminate memory fragmentation. Tools Plus takes care of allocating and deallocating UPPs as required in PowerPC applications and plug-ins. This is how you set the parameter error action routine in C/C++:

```
SetParamRangeErrProc(myActionProc);
```

In Pascal, a similar statement is used except the "@" symbol indicates the address of a routine:

```
SetParamRangeErrProc(@myActionProc);
```

The parameter error action routine is written as a Pascal procedure that has no parameters. Here is an example of how your routine should be written:

C

```
pascal void myParamErrHandlerProc (void)
  {
  // Your code goes here
  }
```

Pascal

```
procedure myParamErrHandlerProc;
  begin
    {Your code goes here}
  end;
```

5 Windows

Windows opened by Tools Plus are identical to those opened by conventional Macintosh toolbox routines, except that they and the objects on them inherit the benefit of being automatically maintained by Tools Plus. Some additional features can also be found in Tools Plus windows that aren't available in ordinary Macintosh windows.

Before you can use a window, you must first open it with the WindowOpen routine. Each window is referenced by a unique window number. This number is specified when the window is opened, and refers to the specific window until that window is closed. After a window is opened, your application can create objects in it such as buttons, lists, scroll bars, editing fields, etc. These items are detailed later in this manual. Windows and the user interface elements in them can be created dynamically within your application, or in the traditional manner by using resource templates (detailed later).

Your application can also have a tool bar located just below the menu bar. It is created with the ToolBarOpen routine. From a programming perspective, a tool bar is treated just like any other window.

At any time, your application can obtain information about a window's location, size, and status with the WindowStatus routine.

Routines that should be used infrequently include WindowMove which lets your application reposition a window, WindowSize which lets your application resize a window, and WindowDisplay which is used to hide and show (unhide) a window. In all cases, Tools Plus correctly maintains your application's user interface to accommodate the changes brought about by these routines.

You can use Tools Plus's windows instead of creating alerts and dialogs with ResEdit. Tools Plus provides the functionality that real alerts and dialogs can, plus it provides additional benefits such as letting you easily incorporate pop-up menus, list boxes, and alternate fonts. Tools Plus also provides routines to let you create your interface using resources if you prefer. See the section on Dynamic Alerts for details on Tools Plus alerts.

When a window is no longer required, it is closed by the WindowClose routine, which releases the memory used by the window's buttons (including radio buttons and check boxes), picture buttons, scroll bars, editing fields, pop-up menus, list boxes, and custom controls.

Note: Much of the Macintosh's power lies in QuickDraw, the part of the toolbox that allows Macintosh programmers to perform highly complex graphic operations quickly and easily. There is an entire chapter in Inside Macintosh dedicated to QuickDraw that is compulsory reading for all Macintosh programmers. The chapter on the Font Manager explains how to display text on a window.

Resource-Based Programming

Tools Plus lets you define user interface elements both in your application's source code (dynamically), and by using resources like the toolbox's Dialog Manager (resource-based). The clear advantage that Tools Plus provides over the Dialog Manager is that it greatly simplifies resource-based programming. While creating user interface elements dynamically is sometimes a preferred method of programming, resource-based programming has some inherent advantages too:

- The user interface can be designed visually using an inexpensive resource editor such as Apple's ResEdit
- The user interface definition can be separated from your application's source code, thus allowing you to make changes to the interface without having to recompile your application
- It facilitates localization
- You can apply custom colors to windows and controls without writing any code
- You can accomplish more using less source code
- It can save memory

Tools Plus routines completely replace the need to use the toolbox's Dialog Manager because they let you use resource templates to create windows, dialogs (windows with user interface elements), menus, and other GUI objects. Once these elements are created, Tools Plus automatically makes them work. This lets you avoid the Dialog Manager's numerous complexities and short-comings that are typically encountered when trying to make your application's user interface work and behave like a Macintosh should.

While Tools Plus's WindowOpen routine opens a window using parameters supplied by your application's code, LoadWindow opens a window using a 'WIND' resource that specifies the window's type, co-ordinates, title, and positioning options. Similarly, the LoadDialog routine opens a window using a 'DLOG' (dialog) resource, then populates the window with user interface elements that are defined in a related 'DITL' (dialog item list) resource.

In many cases, you will be able to open a completely functioning dialog with a single call to the LoadDialog routine. For situations where you want to prepare a window in some way before populating it with the dialog's items, the LoadDialogList routine loads a 'DITL' (dialog item list) resource, attaches it to an already open window, and creates the user interface elements defined in the dialog item list.

The GetDialogItemRect routine reads a dialog's item list and retrieves an item's display rectangle. This is most often utilized in dialog "user items" that provide co-ordinates for list boxes, pop-up menus and other user interface elements that are not automatically created by LoadDialog. Your application uses GetDialogItemRect to retrieve a dialog item's display rectangle, then it uses that rectangle to create the appropriate GUI element. SetDialogItemRect changes the item's display rectangle.

GetDialogFontInfo and SetDialogFontInfo retrieve and set the font, font size and font style used in a dialog window. By default, a dialog displays its text using Chicago 12pt. When you change these settings, new dialogs adopt these font settings and display their static text and editing fields using these settings.

Tools Plus completely circumvents the Dialog Manager when you use resources to create your application's user interface. Instead, it uses its own processes to create and maintain the user interface elements defined by resources. Using Tools Plus instead of the Dialog Manager's routines gives you the following advantages:

- You can still use resources to design and create your user interface, just like the Dialog Manager.
- All system versions and Macintosh models can take advantage of the latest resource structures. The Dialog Manager offers some features only on Macs running Color QuickDraw, and yet others only on Macs running System 7.
- It's easy to create complex and attractive dialogs.
- Your dialogs have all the advantages of windows and interface elements that are created with Tools Plus routines.
- All elements in a dialog work automatically as soon as they are created.
- It's much easier for your application to interact with Tools Plus's user interface elements than to use complex and awkward Dialog Manager routines.
- Tools Plus reports mouse-down "hits" in dialog items if those items don't automatically work when Tools Plus creates them. For example, if you define a user item to create your own custom object, Tools Plus's doClick event will tell you when a mouse-down event occurs in that user item. Other user interface elements, like buttons, list boxes and pop-up menus, work automatically just like they do in a standard Tools Plus window.

Designing Dialogs

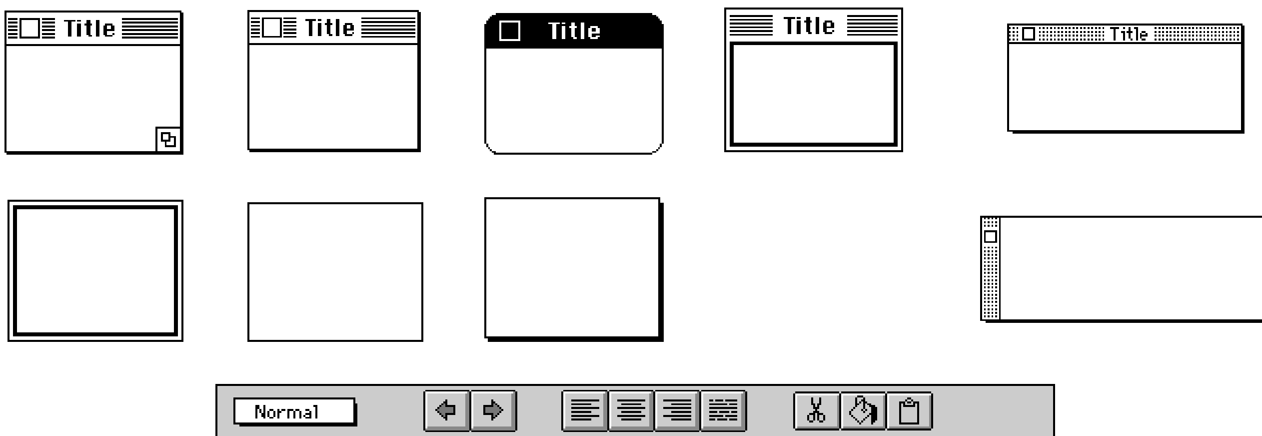
It is almost certain that your existing resources will work perfectly with Tools Plus, and that you will be able to create new dialog-related resources using the same process you have in the past. Here are some tips to ensure that the process goes smoothly. See the LoadDialog routine for more details.

1. You'll need a powerful resource editor such as Resorcerer to exploit dialogs to their fullest potential.
2. If you want to draw text over a background object like a picture, have the picture as a lower numbered dialog item so it draws first. Set the text to draw in srcOr mode and don't set a background color for the control.
3. Static text items can be drawn using srcCopy (text over solid background) or srcOr mode (text over existing objects), also known as drawing with a transparent background. You can edit static text items using Tools Plus's "field" routines providing the static text item uses the srcCopy text transfer mode (the default).
4. You can use "short" format 'ictb' resources to save memory and disk space. Tools Plus corrects a Dialog Manager bug that requires all dialog items to have a separate color table entry. In Tools Plus, items that use identical color and style settings can share the same entry in an 'ictb' structure.

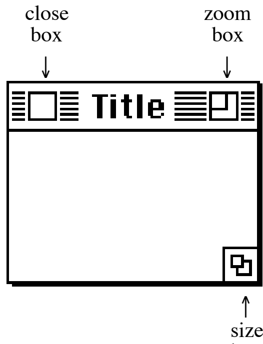
5 If you need to change a dialog window’s font or color settings, it’s a good idea to call GetDialogFontInfo before you make the changes, and call SetDialogFontInfo after the changes are done. This insures that dialog items using the default color and/or text settings will be drawn correctly instead of using the most recent settings.
6 Be careful not to use resource IDs that coincide with those in the System file or your development environment.
7 In Tools Plus, the Appearance Manager’s controls are supported as controls (‘CNTL’ resources) in a dialog (‘DLOG’ resource), just like all other controls. See this manual’s chapters on Buttons and Scroll Bars for more information on how to implement specific controls that are part of the Appearance Manager.
8 You can design your dialogs using standard push buttons, check boxes, and radio buttons, and have those controls remapped to their equivalent Appearance Manager controls by using the ReplaceControlProcID routine. This can be done for you automatically when you initialize your application using the InitToolsPlus routine.
9 Controls that are created using any of the dialog related routines (AppendDialogList, LoadDialog, LoadDialogList, LoadSpecDialog, and LoadSpecDialogBehind) are AutoEmbedded if the Appearance Manager’s routines are available to your application.

Window Types

Various “types” of windows (including a tool bar and palettes) can be created, some of which have title bars and some which resemble alert or dialog boxes. Details are provided in the WindowOpen and ToolBarOpen routines.



Title Bar, Close box, and Zoom box



Any window that contains a title bar can have a “close box.” Also, a document window type can be resized by the user by either clicking on the “zoom box” or dragging the “size box.” The close box, when clicked by the user, instructs your application to close the window.

A window containing a zoom box has two different states: [1] the standard state, and [2] the user state. The user can change the window’s size and/or location, thereby defining the user state. When the zoom box is clicked, the window automatically “zooms” back to the standard state (which can be defined by your application). Clicking the zoom box again reverts to the user state. (Floating palettes can’t have a zoom box).

Your application can define the minimum and maximum size of a window with SetWindowSizeLimits. It can also define the size and position the window assumes when the zoom box is clicked by using the SetWindowZoom routine. GetWindowZoom is used to retrieve the window's current size and location, as well as the size and location that is assumed when the zoom box is clicked.

Any window with a title bar can be re-positioned by dragging it with the mouse. An inactive window can be dragged without being activated by holding the ⌘ key down during the drag. A window's title can be changed by the WindowTitle routine.

Size Box

A size box is a square located in the bottom right corner of a document window. The user can change a window's size by dragging the size box. By default, the window can be sized horizontally to any width from a minimum 120 pixels to the full width of the screen. Vertically, the window can be sized to any height from a minimum of 68 pixels to the full height of the screen (excluding the height of the menu bar). When the window is opened, the default sizing limits for the window are automatically adjusted to the window's size. For example, the minimum size will never be larger than the window's dimensions, and the maximum size will never be smaller than the window's dimensions. These sizing limits can be changed by the SetWindowSizeLimits routine.

Color Backdrops and Background Themes

Each window has a backdrop color that is used when a part of the window is revealed and requires refreshing. The backdrop color is also used to fill an object's region when an object is deleted. By default, the window's backdrop is white. You can set the backdrop color for new windows using the BackdropColor routine. As new windows are opened, they adopt the specified backdrop color. The NoBackdropColor routine resets the color for new windows to the default white.

Color backdrops are particularly useful in tool bars and floating palettes because, as these specialized windows are opened, the user won't see a flash of white while the window is being populated, thus creating an illusion of greater speed.

You can also use the Appearance Manager to draw its background in all or selected windows. An option in the InitToolsPlus routine lets you apply an Appearance Manager background (which is different depending on the window type) to all windows. Alternatively, each Tools Plus routine that opens a window lets you optionally apply the Appearance Manager background to that window. You can also apply a background that is usually associated with a different type of window by using the SetBackgroundTheme routine or the SetNextWindowBackgroundTheme routine. You cannot change the backdrop color after you assign a background theme to a window.

Maximum Number of Open Windows

The maximum number of windows that can be opened simultaneously is specified in the InitToolsPlus routine. See the InitToolsPlus routine (Initialization chapter) for details about specifying the maximum number of windows your application even needs to have open.

Tool Bar and Floating Palettes

Your application can incorporate floating palettes that are always active and “float” above your application’s standard windows. It is best to have floating palettes look different from standard windows to provide the user with a visual cue as to its behavior. This is done by using a special window definition (WDEF) resource that you include in your application. The WDEF makes the window *look* different while Tools Plus takes care of making any window *behave* like a floating palette.

From a programming perspective, the tool bar and floating palettes are just another kind of window. The tool bar is created with the ToolBarOpen routine, whereas floating palettes are opened with the WindowOpen routine (just like ordinary windows). Tools Plus takes care of making them behave properly by ensuring that they are always active and that they stay in front of standard windows.

The tool bar and all open floating palettes in your application are active as long as your application is active. When it is suspended under MultiFinder or System 7 or higher, the tool bar and floating palettes are automatically hidden. The user can make use of the controls located in the tool bar and floating palettes at any time. The actions invoked by those controls usually apply to the frontmost window that is not a tool bar or floating palette. Your application can determine the frontmost floating palette with the FirstPaletteNumber routine.

When your application creates a tool bar, it is always created just below the menu bar across the entire width of the main monitor. If the user changes the main monitor’s size or resolution, the Tools Plus automatically resizes the tool bar to the width of the main monitor.

Even though Tools Plus makes it easy to add a tool bar and floating palettes to your application, you should have an awareness of what Tools Plus *does* to make them work. This information is detailed in this chapter.

Standard Windows

In Tools Plus, a *standard window* is any window in your application that is not a tool bar or floating palette. If your application does not use a tool bar or floating palettes, then all your windows are standard windows. Your application can determine the frontmost standard window by using the FirstStdWindowNumber routine.

Active Window

The *active* window is the window that is acted upon whenever the user types, gives commands, or does whatever is appropriate within the application being used. Almost invariably, this is your application’s frontmost window when your application is active (i.e., when it is not suspended under MultiFinder or System 7 or higher). Note that the active window *may* be a desk accessory when running Finder under System 5 or 6. Since desk accessories are handled automatically by Tools Plus, you only need to be concerned about your own windows.

In Macintosh toolbox terms, the active window is represented by the global WindowPtr constant FrontWindow. Although this information is explained in detail in Inside Macintosh, the basic rule is *only one window is active, and it’s the frontmost one*. Your application can activate a window by using ActivateWindow, and it can determine the active window’s number by using the ActiveWindowNumber routine.

The use of a tool bar or floating palettes introduces additional considerations because your application can now have *multiple* active windows (explained in the Window Layers model later in this chapter). The frontmost “standard” window (not a tool bar or floating palette) still remains the only active standard window. Additionally, the tool bar and all floating palettes are also active.

Activating the toolbar has no effect on the frontmost standard window or any of the floating palettes. Activating a floating palette simply brings it to the front of other floating palettes, without deactivating any windows. Activating an inactive standard window deactivates the frontmost standard window, and brings the newly activated standard window to the front of the standard windows.

Tools Plus automatically maintains the relationship between standard windows, the tool bar, and floating palette, and makes sure windows are activated and deactivated appropriately.

Work Window

With the introduction of the tool bar and floating palettes, your application can simultaneously have *multiple* active windows (the tool bar, all open floating palettes, and the frontmost standard window). This brings up the question: in which window is the user working at the moment? The concept of a *work window* is established solely to answer that question. Tools Plus automatically keeps track of the window that has most recently been the target of user activity.

Your application has only one work window, which can be determined by using the WorkWindowNumber routine. A window gains the “work window” status under any of the following conditions:

- the user clicks in a window, or any object in a window
- a window is opened as modal (because the next action *must* take place within that window)
- a standard window is opened (and therefore activated), and the previous work window was an active standard window
- the work window is closed or hidden, in which case the following window becomes the work window:
  frontmost standard window (if any are open), or
  frontmost floating palette (if any are open), or
  the tool bar (if it is open)
- a window is activated

Your application can treat a work window like an active window, in that it is an eligible target for the user’s activity. If your application does not use a tool bar or floating palettes, the work window is the same as the active window, and you only need to concern yourself with the concepts of an active window and a current window.

Current Window

The *current* window is the target of actions that occur within your application such as creating buttons or editing fields. Usually, the current window is the same as the active window, however there are times when it is desirable to affect a window without activating it. An example of this is when a window needs to be refreshed. This occurs when a window is obscured by another object on the screen. When the window is no longer obscured, the newly revealed section must be re-drawn or “refreshed.”

The CurrentWindow routine can be used to make any of your application’s windows the “current” window. When work on the window is completed, it’s good form to make the active window (or work window in the case where a tool bar and/or floating palettes are used) current. This is done with the CurrentWindowReset routine. Your application can also determine the current window’s number by using CurrentWindowNumber. In Macintosh toolbox terms, the current window’s WindowPtr is represented by the global variable thePort, which is the current grafPort, and can be determined with the GetPort routine.

Editing Field Window

The Editing Field Window is the *one* window in your application containing the active editing field (the field either has a flashing insertion point, or its selected text is highlighted). Tools Plus automatically keeps track of which window contains your application’s active editing field.

If your application does not use a tool bar or floating palettes, this window will likely be the active window (if it has an editing field). If your application uses a tool bar and/or floating palettes, potentially any active window (tool bar, any floating palette, or the active standard window) can contain the active editing field. See the Editing Fields chapter for details on editing fields and the Editing Field Window.

Modal Windows

When a window is modal, clicking the mouse outside the window results in a beep. This means that all interaction is restricted to the modal window until that window is closed. Pull-down menus can't be selected, nor can their ⌘-key equivalents. Modal windows are always opened in front of all other windows (including the tool bar and floating palettes).

When using MultiFinder or System 7 or higher, modal windows behave differently: they can be modal for the current application, or for all applications. The standard dialog box (of dBoxProc type) keeps you in the current application by preventing you from switching to the Finder or other applications, or by launching a new application. All other modal windows, however, allow you to switch to the Finder or other applications (by clicking on one of their objects), and to launch a new application (by double-clicking it or one of its files). They do not, however, let you select menu items or click other windows within the same application. This is consistent with System 7's movable modal dialog that is displayed while copying a file.

Your application's tool bar cannot be modal, nor can any of the floating palettes.

Window Layers

Normally, the Macintosh's Window Manager maintains your application's windows in such a way that the frontmost window is active when your application is active. When your application is suspended (under MultiFinder or System 7), all windows are inactive. Tools Plus preserves the Window Manager's handling of windows (providing your application does not have a tool bar or floating palettes). When a tool bar or floating palette is introduced in your application, Tools Plus automatically ensures that all windows behave as described in the model below.

In System 5 and 6's Finder, only one application can be running at a time. Additionally, desk accessories can be running, and their windows can intermingle with your application's windows. If your application has a tool bar and/or floating palettes, Tools Plus automatically creates a "Desk Accessory Layer" in which all desk accessory windows are kept together to prevent their intermingling with your application's windows. This is done to prevent the confusing condition that arises when the frontmost window is a floating palette or tool bar (belonging to your application), behind which is a desk accessory, followed by another window belonging to your application. To the user, it may (erroneously) appear that the palette's operations apply to the desk accessory.

The following model describes how Tools Plus maintains the order of windows:

Front
(nearest to user)

Layer	Behavior
Modal Windows	• Modal windows open at the front of this layer • Multiple modal windows can be open simultaneously • Frontmost modal window is only accessible window overriding all others
Tool Bar	• Always active (only one tool bar can be open) • Inaccessible if a modal window is open • Automatically hidden when your application is suspended
Floating Palettes	• Floating palettes open at the front of this layer • Always active • Multiple floating palettes can be open simultaneously • Inaccessible if a modal window is open • Automatically hidden when your application is suspended
Standard Windows	• Standard (modeless, non-tool bar non-floating palette) windows open at the front of this layer • Only the frontmost window in this layer is active • Multiple standard windows can be open simultaneously • When running under System 5 or 6's Finder, desk accessory windows may appear in this layer, providing a tool bar or floating palette is not open • Inaccessible if a modal window is open

Window Layer Model

Global and Local Co-ordinates

There are two co-ordinate systems that are used when creating Tools Plus user interface elements: *global* and *local*. Global co-ordinates are relative to the main monitor where the upper left corner is point 0,0. Local co-ordinates are relative to the current window where the window's upper left corner is point 0,0. The co-ordinate numbers increase when moving right or down, and decrease when moving left or up.

The Macintosh toolbox has a global variable called screenBits.bounds that is a rectangle defining the boundary of the Macintosh's main monitor in global co-ordinates. You may want to use this variable to help you define a window's maximum size, and zooming co-ordinates.

Details regarding global and local co-ordinates and screenBits.bounds can be found in Inside Macintosh.

Objects in Windows

Tools Plus automatically maintains the relationships between windows, user interface elements you place on those windows, and the rest of the Macintosh environment. Simply put, Tools Plus makes your user interface work. The user interface elements that are directly maintained by Tools Plus are:

- windows (modal and modeless), tool bar, floating palettes and dynamic alerts
- standard Macintosh buttons
- custom CDEFs that behave like a push button, check box or radio button
- scroll bars
- custom CDEFs that behave like scroll bars
- picture buttons
- list boxes
- editing fields
- pop-up menus
- cursor and cursor zones

Additionally, Tools Plus provides routines that facilitate drawing in windows, such as various picture and text drawing routines. User interface elements placed in windows can automatically move and/or resize as their window's size changes (see AutoMoveSize for details).

The ‘dftb’ Resource - Font and Color Settings

The ‘dftb’ (dialog font table) resource was first introduced with the Appearance Manager in Mac OS 8, but Tools Plus supports it on all systems with or without the Appearance Manager. The ‘dftb’ contains one record for each user interface element in your dialog item list (‘DITL’ resource). Each record can set font and color information for any user interface element. In Tools Plus, the ‘dctb’ resource works with or without an ‘ictb’ resource. If you have both resources in a dialog, Tools Plus merges the setting in both the ‘dctb’ resource and the ‘ictb’ resource and applies them to the dialog's elements. In situations where both resources set an item (i.e., the ‘ictb’ resource indicates that Geneva 9 should be used while the ‘dftb’ resource indicates Monaco 10), individual settings in the ‘dftb’ resource override equivalent settings in the ‘ictb’ resource.

The format of the ‘dftb’ resource contains one or more records, each of which corresponds to a single item in your dialog item list. Each record in the ‘dftb’ resource has one of two formats: (1) a “skip” command indicating that there are not settings (and no data) for a specific user interface element, and (2) a record that contains all the possible settings for any user interface element. If your resource editor can not automatically create and maintain the ‘dftb’ resource and to keep it synchronized with your ‘DITL’ resource, Tools Plus includes a resource template for the ‘dftb’ resource. See the “Optional Resources” folder for the ‘dftb’ resources and copy it into your resource editor application. The ‘dftb’ resource's data is as follows:

Byte Offset	Length (Bytes)	Bit Num	Description
0	2		**Version** (always 0)
2	2		**Font Styles**: Number of records in this resource (should be one record for each item in the related 'DITL' resource)
The follow items represent a single variable length record for a single item in the 'DITL' resource…			
4	2		**Entry Type**: 0 = no data available (skip), 1 = data available Fields following this record exist only when Entry Type =1.
6	2	15-10	*(reserved)*
		9	**Use font name**: Set to 1 if item's font is set using a font name instead of a font number.
		8	**Add font size**: Set to 1 if item's font size is calculated using the window's font size plus a specified point value.
		7	*(reserved)*
		6	**Use justification**: Set to 1 if item's text justification is set (any editing field only)
		5	**Use mode**: Set to 1 if item's text transfer mode is set (static text items only)
		4	**Use background color**: Set to 1 if item's background color is set (edit text items, static text items, editing fields, static text field, and some controls)
6	2	3	**Use foreground color**: Set to 1 if item's foreground color is set (edit text items, static text items, editing fields, static text field, and some controls)
		2	**Use size**: Set to 1 if item's font size if set to an exact value, such as 10pt.
		1	**Use face**: Set to 1 if item's font style is set
		0	**Use font**: Set to 1 if item's font is set using a font number
8	2		**Font number**: Font number set for item
10	2		**Font size**: Font size in points set for item. If "Add Font Size" bit is set, this value indicates the number of points that are added to the window's font size to determine item's font size.
12	2	15-7	*(reserved)*
		6	**Extended**: Set to 1 if item's font's style includes the "extended" attribute
		5	**Condensed**: Set to 1 if item's font's style includes the "condensed" attribute
		4	**Shadow**: Set to 1 if item's font's style includes the "shadow" attribute
		3	**Outline**: Set to 1 if item's font's style includes the "outline" attribute
		2	**Underline**: Set to 1 if item's font's style includes the "underline" attribute
		1	**Italic**: Set to 1 if item's font's style includes the "italic" attribute
		0	**Bold**: Set to 1 if item's font's style includes the "bold" attribute
14	2		**Text mode**: Item's text transfer mode (static text items only) srcCopy = 0 (static text item is implemented as a static text field) srcOr = 1 srcXor = 2 srcBic = 3 notSrcCopy = 4 notSrcOr = 5 notSrcXor = 6 notSrcBic = 7
16	2		**Justification**: Item's text justification (editing fields only) teJustLeft = 0 teJustCenter = 1 teJustRight = -1
18	6		**Foreground color**: Item's foreground color (edit text items, static text items, editing fields, static text field, and some controls)
24	6		**Background color**: Item's background color (edit text items, static text items, editing fields, static text field, and some controls)
30	1-256		**Font name**: Font name used to set item's font. This Pascal string is truncated to the number of valid characters plus a length byte. The record length varies because of this field.

Substituting Window ProcIDs

Certain system resources may or may not be available to your application depending on the system version of the Macintosh that is running your application. A perfect example of this is the “utility window,” normally called a floating palette, which is part of the Appearance Manager in Mac OS 8 or later, and the unattractive floating palette which most developers choose to avoid that is available in System 7.5 or later. With Tools Plus, you can design and write your application to use a custom window definition (WDEF resource) for a floating palette. Then at the beginning of your application it can determine the Mac’s capabilities, specifically if the Appearance Manager is running to make the “utility window” available to your application. If this is the case, your application can easily substitute the use of the custom floating palette WDEF with the Appearance Manager’s utility window throughout your application.

Two routines in the “Miscellaneous Routines” chapter of this manual help facilitate determining the capabilities of the Macintosh that is running your application: HasAppearanceManager and UsingAppearanceManager. You can also use the toolbox’s Gestalt routines to determine whether other features are available or not. Tools Plus's ReplaceWindowProcID routine is used to replace a specific window procID with another procID throughout your application, thereby substituting the use of one type of window with another.

Live Window Dragging and Resizing

The default Mac OS behavior for dragging and resizing windows displays a dotted outline that tracks the window’s eventual position or size for as long as the user holds the mouse button down. When the user releases the mouse button, the window snaps to its new position or size. This convention was originally adopted solely because of the limited processing capabilities of early Macintosh computers. Since mid 1998, Apple’s entry-level computers have been more than capable of overcoming these limitations.

Tools Plus optionally lets the user move and resize windows in real time. With this option turned on, the change in the window’s location or size is seen immediately and continuously as the user drags a window’s title bar or resizes the window by dragging the grow box. This option looks best on faster Macintoshes, like those equipped with a G3 processor or better, because the target window and those behind it may need to be refreshed frequently and rapidly. Faster Macintoshes, those equipped with a G3 processor or better, will provide fluid motion like that experienced on Windows NT workstations.

The InitToolsPlus routine has options to turn on real time window dragging and resizing unconditionally, or only if a specific processor is used. For greater flexibility, the SetLiveWindowDragging routine lets your application turn this feature on or off under its own conditions.

Special Considerations

In Mac OS 8.5 and later, certain themes may crash your system if you open a window that is too small, or too far off screen for the theme’s liking. If opening a window crashes your system or application, try making the window larger, and it you are creating the window off screen, change its co-ordinates so that they are not as far off the screen. The same applies for resizing or repositioning a window.

Handling Windows

Much of the control exercised over windows is performed by your application. By default, your application responds to window events in its main event handler routine. You can optionally have a separate event handler routine for each window. Tools Plus constantly inquires about any requests the system may have. Some of these requests must be acted upon, such as refreshing a window. Other requests may be interpreted by your application, such as the user clicking in a window’s close box, or clicking in another window. These facets are all detailed in the Event Management chapter. Tools Plus automatically handles such chores as sizing, dragging, and zooming windows.

When working with windows, it is important to remember that many Tools Plus routines (such as creating buttons and editing fields) apply to the *current* window.

The Macintosh toolbox's FrontWindow routine becomes less useful now that the frontmost window can be a tool bar or any of the floating palettes (instead of the frontmost standard window). Fortunately, Tools Plus provides routines to determine the frontmost floating palette, and frontmost standard window.

If your application is drawing in color, or if it needs to know about the details of a monitor such as the number of colors displayed or its size, see the Color Drawing & Multiple Monitors chapter later in this manual.

WindowOpen

Open a new window and make it the active and current window.

C
```
pascal void WindowOpen (short Window, short left, short top, short right,
             short bottom, const Str255 Title, long Spec, Boolean goAwayFlag,
             Boolean modalFlag);
```

Pascal
```
procedure WindowOpen (Window, left, top, right, bottom: INTEGER;
             Title: STRING; Spec: LONGINT; goAwayFlag, modalFlag: BOOLEAN);
```

Windows are opened as per the Window Layer model described earlier in this chapter. To summarize:

- a modal window opens in front of all other windows, and it is the sole active window
- a floating palette opens in front of the floating palette layer, and it is active
- a standard window opens in front of the standard window layer, and it is active if a modal window is not open.

The window adopts a backdrop color as set by the BackdropColor routine (default is white). The window's background color (as obtained by GetBackRGB) is initially set to the backdrop color when the window is opened.

Window specifies the window number that is opened. Once a window is opened, it is referenced by this window number. If a window using the same window number is already open, it is closed, then a new window is opened as specified by the parameters in the WindowOpen routine, thereby re-using the window number. The newly opened window becomes "active" (frontmost and highlighted as such) and "current" (action pertaining to graphics, text, buttons, scroll bars, etc. occurs in this window). Tools Plus allows up to 250 windows to be open simultaneously, however, you may choose to reduce this limit when using InitToolsPlus to initialize Tools Plus.

Left, *top*, *right*, and *bottom* define a rectangle in global co-ordinates that specifies the window's size and location. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). Some useful things to remember about sizing your window are:

- The co-ordinates you specify define the *usable* area. The window's title bar, outline and shadow are drawn *outside* these co-ordinates.
- If a document window has a right and/or bottom scroll bar (documentProc type), 15 pixels of the usable area are used up by the width of the scroll bar.
- Windows with a title bar use an additional 19 pixels above the window's top co-ordinate to draw the title bar.
- The menu bar takes up 20 pixels, so no window should have a top co-ordinate which is less than 20.

If the tool bar is open, and it was created with the tbOffsetNewWindows option, this window's co-ordinates are shifted downwards by an amount that is equal to the tool bar's height.

Title is the window's title. Note that some windows do not have a title. In this case, the title parameter may be passed as a null string ('').

Spec specifies a window's appearance and behavior. It is a combination of a window procID plus various Tools Plus options detailed later in this section.

The *goAwayFlag* indicates if a "close box" is available in the window's title bar. Close boxes are only drawn in the documentProc, noGrowDocProc, rDocProc, paletteProc, altPaletteProc, and ordPaletteProc windows. The two constants that can be used for this flag are GoAway and NoGoAway.

The *modalFlag* indicates if the window is "modal" or not. When a modal window is open, clicking outside the window results in a beep. This means that all interaction is restricted to the modal window until that window is closed. The two constants that can be used for this flag are Modal and NotModal. Floating palettes cannot be modal.

Appearance and Behavior Specification

Spec specifies a window's appearance and behavior. It is a combination of a window procID plus various Tools Plus options. The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result. For example, a document window with a zoom box would have a spec of documentProc + ZoomBox. The constants defining the available options are as follows:

Choose only one of the following procIDs…

`documentProc` Standard Apple document window with a grow box. Used for windows that need to be resized by the user.

`noGrowDocProc` Standard Apple document window without a grow box. Used for windows that are either fixed size or resized under the application's control.

`rDocProc` Standard Apple "desk accessory" window.

`movableBoxProc` Standard Apple "movable modal dialog" window available on System 7 or later. When running on System 6 or older, a noGrowDocProc window is used instead but it exhibits movable modal behavior.

`dBoxProc` Standard Apple modal dialog. Before System 7, this window physically prevents access to anything outside the window. In System 7 and later you can optionally have access to menus.

`plainDBox` Standard Apple plain dialog. Usually used for splash screens. If made modal, it still allows the user to switch to another application.

`altDBoxProc` Standard Apple drop-shadow dialog. If made modal, it still allows the user to switch to another application.

`paletteProc` Tools Plus floating palette with drag bar across the top. You must include the Tools Plus floating palette WDEF (or equivalent) in your application, and the resource ID must be 2000.

`ordPaletteProc` Window that looks like a floating palette but behaves like a standard window. Seldom used. You must include the Tools Plus floating palette WDEF (or equivalent) in your application, and the resource ID must be 2000.

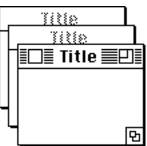
`altPaletteProc` Tools Plus floating palette with drag bar across the left side. You must include the Tools Plus floating palette WDEF (or equivalent) in your application, and the resource ID must be 2000.

(your own procID) `+ wPalette` You can use your own WDEF or those created by third parties to create standard windows that look different from Apple's, or to create floating palettes. See the note on custom WDEFs later in this section for calculation of the procID. Add the wPalette constant to your procID if you want your window to behave like a floating palette.

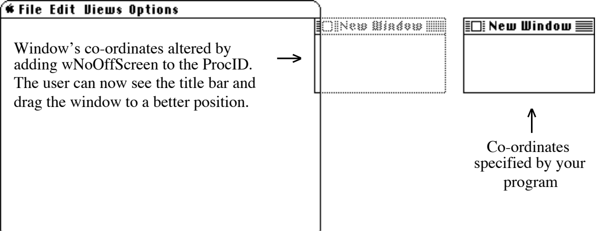
Optionally choose only one of the following auto-position options…

`wCenter` — Center the window on the main monitor. Useful for alerts and progress indicators like a thermometer.

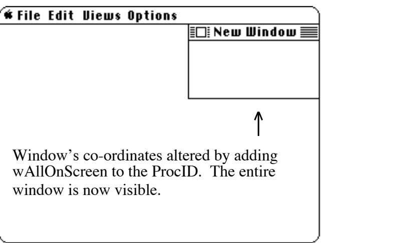
`wTile` — Tile the window in relation to the frontmost standard window. If no standard windows are open, the window opens in the top-left corner on the main monitor. This option is usually detrimental when opening a floating palette or modal window.

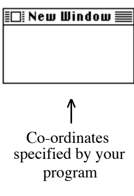


`wNoOffScreen` — Ensures that the window is at least partially visible when it is opened thereby allowing the user to drag it to an optimum position. If your application opens windows based on a stored location (likely kept in a preferences file or in the document), the window might be completely off the screen if the document is opened on a Mac with a monitor smaller than the document's creator. This option alleviates that problem. If you add this option to a window without a title bar (dBoxProc, plainDBox or altDBoxProc), it is positioned entirely on the screen because the user can't drag it to a better position.



`wAllOnScreen` — Tools Plus does the necessary calculations to make sure the entire window is visible. If the window is too large to fit on the screen, it is positioned such that the window's bottom and/or right side may extend beyond the screen's edges (the top left-hand corner will be visible). This is a variation of the wNoOffScreen option above. If you add this option to a window without a title bar (dBoxProc, plainDBox or altDBoxProc), it is positioned entirely on the screen because the user can't drag it to a better position.





Optionally choose only one of the following menu access options…

`wAllowEditMenu` — Allow a modal window to access the Edit menu. By default, when a modal window is opened, pull-down menus are automatically disabled and the user is prevented from accessing them (they hear a beep when they click on the menu bar). This option temporarily disables all menus except for Edit. In the Edit menu, the Undo, Cut, Copy, Paste and Clear items are enabled and disabled automatically as per the active editing field on the modal window. When the modal window is closed, the pull-down menus are restored to their original settings as set by your application. This option has no effect on modeless windows. If you are writing a plug-in, use this option only if the host's Edit menu follows the standards defined in the Menus chapter of this user manual.

`wAllowMenus` — Allow a modal window to access all menus. By default, when a modal window is opened, pull-down menus are automatically disabled and the user is prevented from accessing them (they hear a beep when they click on the menu bar). This option gives the user access to all pull-down menus as specified by your application (Tools Plus does not automatically enable or disable the menus). This option has no effect on modeless windows. It is safest not to use this option if you are writing a plug-in.

Note: In applications running under System 5 or System 6, the dBoxProc window is *truly* modal, and effectively prevents the user from accessing pull-down menus regardless of the menu accessing options described above.

Optionally choose any of the following options…

`wDimEditMenu` Disable the Edit menu's standard editing items (Undo, Cut, Copy, Paste, Clear and Select All) as the window is opened. This is useful when opening a window that has no editing fields, or one that has fields but none are active by default. If you are writing a plug-in, use this option only if the host's Edit menu follows the standards defined in the Menus chapter of this user manual.

`ZoomBox` Include a "zoom box" in either the documentProc or noGrowDocProc type windows. Floating palettes cannot have a zoom box.

`wRefresh` Generate a doRefresh event as the window is opened. DoRefresh events are not discarded if you open numerous windows before getting your next event. By default, windows do not generate doRefresh events as they are opened.

`wManualUpdate` You prefer to manually use the BeginUpdate and EndUpdate routines when this window needs to be refreshed. By default, Tools Plus automatically restricts drawing to only the part of the window that needs refreshing when your application gets a doPreRefresh of doRefresh event. See the doPreRefresh and doRefresh events for details.

`wUnprotectedRefresh`
By default, user interface elements are protected (cannot be overwritten) when your application draws to a window in response to a doRefresh event. This option turns off the protection to allow your application to draw anywhere on the window.

`wNoZoomLines` Suppress "zoom lines" when the user clicks a window's zoom box and zooms between a standard state and a user state. Use this option if you need the fastest possible speed for making a transition between the standard state and the user state.

`wBackgroundTheme`
Include the Appearance Manager's background theme in this window. You cannot set the window's content color when a background theme is used. Alternatively, you can use the SetBackgroundTheme routine which lets you set a brush that may not normally be associated with a specific type of window. This option is ignored if the Appearance Manager is not available.

`wHidden` Open the window as "hidden" (i.e., it is accessible to your application but invisible to the user).

Floating Palette, Custom WDEFs and Appearance Manager

If your application uses floating palettes or custom windows, you need to include a special window definition (WDEF resource) in your application's resource fork. Tools Plus provides the required WDEF for floating palettes, and you can find it in the "Palette WDEF" file in the "Optional Resources" folder on the Tools Plus disk. Add this WDEF resource to your project's resource file before you compile your application.

You can write your own WDEF or you can use third-party WDEFs. As per Macintosh standards, a window's procID is comprised from the following formula: WDEF resource ID x 16 + *variant code*. When you add the wPalette constant to the window's procID, Tools Plus makes the window behave like a floating palette. Appearance Manager

If your application is running on a Macintosh that has an Appearance Manager, then it automatically has access to a number of additional window types (WDEFs). Consult your Appearance Manager SDK (Software Developer Kit) for details. It is easiest to program your application using standard windows, then as your application starts up it can check to see if the Appearance Manager is available. If the Appearance Manager is available, use the ReplaceWindowProcID routine to replace standard window procIDs with those in the Appearance Manager. You can do this automatically in the InitToolsPlus routine.

Note: When using third party WDEFs (like the Infinity Windoid), make sure you carefully read the documentation that accompanies the WDEF. Your WDEF will likely have variant codes that differ from those expected by the Tools Plus, so you will likely not be able to use the Tools Plus constants ordPaletteProc, paletteProc or altPaletteProc.

```
CONST                                              {Window definition IDs (ProcIDs):         }
      documentProc   =  0;                         {Standard document window with size box   }
      dBoxProc       =  1;                         {Alert box or modal dialog box            }
      plainDBox      =  2;                         {Plain box (usually modal)                }
      altDBoxProc    =  3;                         {Plain box with shadow (usually modal)    }
      noGrowDocProc  =  4;                         {Document window without size box         }
      movableBoxProc =  5;                         {Movable dialog                           }
      rDocProc       = 16;                         {Round corner window (desk accessories)   }

      kWindowDocumentProc                 = 1024;  {ProcIDs for Appearance Manager's windows…  }
      kWindowGrowDocumentProc             = 1025;  {                                          }
      kWindowVertZoomDocumentProc         = 1026;  {                                          }
      kWindowVertZoomGrowDocumentProc     = 1027;  {                                          }
      kWindowHorizZoomDocumentProc        = 1028;  {                                          }
      kWindowHorizZoomGrowDocumentProc    = 1029;  {                                          }
      kWindowFullZoomDocumentProc         = 1030;  {                                          }
      kWindowFullZoomGrowDocumentProc     = 1031;  {                                          }

      kWindowPlainDialogProc              = 1040;  {ProcIDs for Appearance Manager's dialogs…  }
      kWindowShadowDialogProc             = 1041;  {                                          }
      kWindowModalDialogProc              = 1042;  {                                          }
      kWindowMovableModalDialogProc       = 1043;  {                                          }
      kWindowAlertProc                    = 1044;  {                                          }
      kWindowMovableAlertProc             = 1045;  {                                          }

      kWindowFloatProc                    = 1057;  {ProcIDs for Appearance Manager's top-title }
      kWindowFloatGrowProc                = 1059;  {  floating windows…                        }
      kWindowFloatVertZoomProc            = 1061;  {                                          }
      kWindowFloatVertZoomGrowProc        = 1063;  {                                          }
      kWindowFloatHorizZoomProc           = 1065;  {                                          }
      kWindowFloatHorizZoomGrowProc       = 1067;  {                                          }
      kWindowFloatFullZoomProc            = 1069;  {                                          }
      kWindowFloatFullZoomGrowProc        = 1071;  {                                          }

      kWindowFloatSideProcID              = 1073;  {ProcIDs for Appearance Manager's side-title }
      kWindowFloatSideGrowProcID          = 1075;  {  floating windows…                        }
      kWindowFloatSideVertZoomProcID      = 1077;  {                                          }
      kWindowFloatSideVertZoomGrowProcID  = 1079;  {                                          }
      kWindowFloatSideHorizZoomProcID     = 1081;  {                                          }
      kWindowFloatSideHorizZoomGrowProcID = 1083;  {                                          }
      kWindowFloatSideFullZoomProcID      = 1085;  {                                          }
      kWindowFloatSideFullZoomGrowProcID  = 1087;  {                                          }

                                                   {Add to the procID for these features:    }
      ZoomBox          =  8;                       {Zoom box                                 }
      wCenter          = $00010000;                {Auto-centering, or…                      }
      wTile            = $00020000;                {Auto-tiling                              }
      wRefresh         = $00040000;                {Generate a refresh event right away      }
      wNoOffScreen     = $00080000;                {Prevent from being off-screen (auto move) }
      wAllOnScreen     = $00100000;                {Entire window must be on screen (auto move) }
      wNoZoomLines     = $01000000;                {Suppress "zoom lines" when zooming       }
      wHidden          = $02000000;                {Window is opened hidden                  }
      wBackgroundTheme = $04000000;                {Use Appearance Manager's background theme }
      wPalette         = $80000000;                {Window behaves like palette              }

                                                   {Alternate (custom) procIDs:              }
      ordPaletteProc   = 32000;                    {window that looks like a floating palette }
      paletteProc      = 32000 + wPalette;         {Tools Plus's Floating Palette window     }
      altPaletteProc   = 32002 + wPalette;         {Tools Plus's palette with drag bar on left }

                                                   {Add to modal window procID for:          }
      wAllowEditMenu   = $40000000;                {Allow access to Edit menu only  or       }
      wAllowMenus      = $20000000;                {Allow access to all menus                }
      wDimEditMenu     = $10000000;                {Disable Edit menu's items                }

                                                   {"Close box availability" indicators:     }
      GoAway           = true;                     {Create "close" box                       }
      NoGoAway         = false;                    {Do not create "close" box                }

                                                   {"Modal window" indicators:               }
      Modal            = true;                     {Window is modal                          }
      NotModal         = false;                    {Window is modeless                       }
```

Warning: When you open a window, make sure that the co-ordinates are such that at least part of the window’s title bar is visible to let the user reposition the window. Alternatively, add either of the wNoOffScreen or wAllOnScreen constants to the window’s spec parameter. If you are using a tool bar, make sure you open your window *lower* to ensure that its title bar is not hidden by the tool bar, or let the ToolBarOpen routine do this for you automatically.

Note: Although it is programatically possible, you should not open or close windows when a modal window is active. The only exception to this is the use of another modal window.

Programming Tips:

1. Open a window with the wRefresh constant added to the window’s spec parameter, then populate it with Tools Plus objects (buttons, list boxes, etc.). This produces a doRefresh event, which in turn calls upon your window updating routine to draw the remaining lines and text. The advantage to this is that all of Tools Plus's objects are protected (and therefore can’t be overwritten) when your window drawing routine does its work.
2. If your application follows the first tip (above), your window refreshing routine can paint the window with a color or pattern, then draw text and/or lines on top.
3. You can make a window *appear* to open faster by creating a hidden window, creating the necessary Tools Plus objects, then displaying the window. When the window is displayed, a doPreRefresh event is generated giving your application an opportunity to draw a background then Tools Plus draws its objects. Lastly, a doRefresh event is generated letting your application perform any drawing after Tools Plus has drawn its objects. This process takes longer to complete than just opening a window and populating it, but from a user’s perspective, they experience a momentary pause then the window quickly materializes as opposed to being drawn piece by piece more slowly. Remember, it takes longer to create an object than to redraw it.
4. If your application stores a window’s co-ordinates (in a document or a preferences file), be aware that someone may try to open the document on a Mac with a monitor that is smaller than the one used by the document’s creator. By adding either the wNoOffScreen or wAllOnScreen constant to the window’s spec parameter, the window opens where the user can see it. Your application can then respond to a doMoveWindow (window was moved) event by storing the window’s *new* location. If the user does not move the window, your application retains the window’s original co-ordinates. Later, when the document is opened on the creator’s Mac (with a larger monitor), the window is in its original position.
5. Although the Tools Plus WDEF is quite memory efficient, you may want to use the “Infinity Windoid” instead (it’s a third party floating palette WDEF included with Tools Plus disks). The Infinity Windoid supports color to give you commercial quality palettes.

Also see: ToolBarOpen, WindowOpenRect, LoadWindow and BackdropColor.

WindowOpenRect

Open a new window and make it the “active” and “current” window.

C

```
pascal void WindowOpenRect (short Window, const Rect *Bounds,
            const Str255 Title, long Spec, Boolean goAwayFlag,
            Boolean modalFlag);
```

Pascal

```
procedure WindowOpenRect (Window: INTEGER; Bounds: RECT; Title: STRING;
            Spec: LONGINT; goAwayFlag, modalFlag: BOOLEAN);
```

WindowOpenRect is identical to the WindowOpen routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

WindowOpenRectBehind

Open a new window behind another window.

C
```
pascal void WindowOpenRectBehind (short Window, const Rect *Bounds,
            const Str255 Title, long Spec, Boolean goAwayFlag,
            Boolean modalFlag, short BehindWindow);
```

Pascal
```
procedure WindowOpenRectBehind (Window: INTEGER; Bounds: RECT; Title: STRING;
            Spec: LONGINT; goAwayFlag, modalFlag: BOOLEAN;
            BehindWindow: INTEGER);
```

WindowOpenRectBehind is identical to the WindowOpenRect routine, except that it places the new window behind a specified window, or at the back of a window layer.

BehindWindow specifies the number of an open and visible Tools Plus window behind which the new window is created. This window must be in the same window layer as the new window, meaning that a floating palette can only open behind another floating palette, and a standard window can only open behind another standard window. A value of -1 opens the new window at the front of its layer. This produces the same result as using WindowOpenRect. A value of 0 opens the new window at the back of its layer to make it the rear-most floating palette or standard window.

If the window specified by BehindWindow is not a floating palette or a standard modeless window, or if the window is hidden or closed, WindowOpenRectBehind creates the new window at the front of its layer as though you had used WindowOpenRect.

LoadWindow

Open a window using a ‘WIND’ resource.

C
```
pascal void LoadWindow (short Window, short ResID);
```

Pascal
```
procedure LoadWindow (Window, ResID: INTEGER);
```

LoadWindow opens a window by calling the NewWindow routine and supplying it with values from a ‘WIND’ resource, commonly called a window template. This is a good way to create a window that requires a custom color table.

Window specifies the window number that is opened (1 to 250, or the limit you specified in InitToolsPlus). Once a window is opened, it is referenced by this window number. If a window using the same window number is already open, it is closed, then a new window is opened as specified by the parameters in the ‘WIND’ resource, thereby re-using the window number.

ResID is the ‘WIND’ resource ID number that is used to create the window. If the window has a ‘wctb’ color table resource, it must use the same ID number. Any resource ID number can be used, but numbers 128 or higher are safest as stated in Inside Macintosh.

Since System 7, the Window Manager has had very specific needs for window color tables, and Inside Macintosh warns you against creating your own. Tools Plus protects you from color table inconsistencies by creating a window color table that matches the default color table, then updating or appending that table with entries you supply in the ‘wctb’ resource. This lets you supply a custom WDEF with an expanded color table, and it also ensures that you don’t accidentally damage the window color table as required by Mac OS.

When creating windows using ‘WIND’ resources, flag your ‘WIND’ and ‘wctb’ resources as purgeable to save memory. Tools Plus makes a copy of their data.

Auto Position Options

The 'WIND' resource can contain an optional 2-byte window positioning field. In ResEdit, you can access this field by selecting the required 'WIND' resource, then using the Resource menu's "Open Using Template…" command. Choose a 'WIND' template and you will see the "Auto Position" field. You specify how you want to position the window by entering a specific value in the 'WIND' resource's "Auto Position" field, detailed below. There are several terms used in the description of auto positioning options, and they are defined as follows:

center	Centered both vertically and horizontally relative to either a screen or another window. If the window being centered relative to another window is wider than the window that preceded it, it is pinned to the relative window's left edge.
stagger	This is similar to Tools Plus's *tiling*. Each staggered window is offset by 10 pixels horizontally and vertically.
alert position	Centered horizontally and placed in the "alert position" vertically, that is with one fifth of the window or screen above the new window and the rest below.
parent window	The front most, standard window. You can determine the parent window using the FirstStdWindowNumber routine.

Use one of the following values in the 'WIND' resource's "Auto Position" field to specify how the window is positioned:

$0000	Use co-ordinates specified in the 'WIND' resource
$280A	Center on main screen
$300A	Place in alert position on main screen
$380A	Stagger on main screen
$A80A	Center on parent window
$B00A	Place in alert position on parent window
$B80A	Stagger relative to parent window
$680A	Center on parent window's screen
$700A	Place in alert position on part window's screen
$780A	Stagger on parent window's screen

Also see: WindowOpen, LoadSpecWindow and LoadDialog.

LoadSpecWindow

Open a window using a 'WIND' resource.

C
```
pascal void LoadSpecWindow (short Window, long Spec, short ResID);
```

Pascal
```
procedure LoadSpecWindow (Window: INTEGER; Spec: LONGINT; ResID: INTEGER);
```

LoadSpecWindow is identical to the LoadWindow routine, except that it requires the additional *Spec* parameter to give you control over all the appearance and behavior options offered by Tools Plus. See the WindowOpen routine for details about the Spec parameter.

LoadSpecWindowBehind

Open a window using a ‘WIND’ resource behind another window.

C
```
pascal void LoadSpecWindowBehind (short Window, long Spec, short ResID,
            short BehindWindow);
```

Pascal
```
procedure LoadSpecWindowBehind (Window: INTEGER; Spec: LONGINT;
            ResID, BehindWindow: INTEGER);
```

LoadSpecWindowBehind is identical to the LoadSpecWindow routine, except that it places the new window behind a specified window, or at the back of a window layer.

BehindWindow specifies the number of an open and visible Tools Plus window behind which the new window is created. This window must be in the same window layer as the new window, meaning that a floating palette can only open behind another floating palette, and a standard window can only open behind another standard window. A value of -1 opens the new window at the front of its layer. This produces the same result as using LoadSpecWindow. A value of 0 opens the new window at the back of its layer to make it the rear-most floating palette or standard window.

If the window specified by BehindWindow is not a floating palette or a standard modeless window, or if the window is hidden or closed, LoadSpecWindowBehind creates the new window at the front of its layer as though you had used LoadSpecWindow.

LoadDialog

Open a dialog (window with user interface elements) using a ‘DLOG’ resource.

C
```
pascal void LoadDialog (short Window, short ResID);
```

Pascal
```
procedure LoadDialog (Window, ResID: INTEGER);
```

LoadDialog opens a window by calling the NewWindow routine using the parameters supplied by a ‘DLOG’ resource, commonly called a dialog template. The window is populated with user interface items as specified by a related ‘DITL’ (dialog item list) resource.

Window specifies the window number that is opened (1 to 250, or the limit you specify in InitToolsPlus). Once a window is opened, it is referenced by this window number. If a window using the same window number is already open, it is closed, then a new window is opened as specified by the parameters in the ‘DLOG’ resource, thereby re-using the window number.

ResID is the ‘DLOG’ resource ID number that is used to create the window. If the dialog has a ‘dctb’ color table resource, it must use the same ID number. Any resource ID number can be used, but numbers 128 or higher are safest as stated in Inside Macintosh.

Since System 7, the Window Manager has had very specific needs for window color tables, and Inside Macintosh warns you against creating your own. Tools Plus protects you from color table inconsistencies by creating a window color table that matches the default color table, then updating or appending that table with entries you supply in the ‘dctb’ resource. This lets you supply a custom WDEF with an expanded color table, and it also ensures that you don’t accidentally damage the window color table as required by Mac OS.

When creating windows using ‘DLOG’ and/or ‘DITL’ resources, the following tips will help you make the most of programming with resources:

- The dialog can be automatically position just like a ‘WIND’ resource. See LoadWindow for details on how to add auto-positioning options to your ‘DLOG’ resource.
- Flag your ‘DLOG’, ‘DITL’ and ‘dctb’ resources as purgeable to save memory. Tools Plus makes a copy of their data.

- Item numbers are assigned sequentially for dialog items starting at one (1). Tools Plus user interface elements share the same numbers. The following is an example of how your dialog's items are translated to Tools Plus user interface elements:

Item #	'DITL' Item	Tools Plus Item
1	Static Text (srcCopy)	Field #1 (static)
2	Static Text (other than srcCopy)	
3	Field	Field #3
4	Field	Field #4
5	Check Box	Button #5
6	Radio Button	Button #6
7	Radio Button	Button #7
8	Push Button	Button #8
9	User Item	
10	Field	Field #10

- If item number 1 is a push button, it becomes the default button and a default frame is drawn around it.
- When a 'DITL' resource (dialog item list) uses 'CNTL' resources (control templates), Tools Plus makes some assumptions about how to translate the information that is supplied by the 'CNTL' resource into a Tools Plus user interface element. Complete details on how to populate the 'CNTL' resource can be found in this user manual in the chapters that cover Buttons, Scroll Bars, Editing Fields, List Boxes and Pop-Up Menus.

  Note that in the table below, the rightmost column describes how the control is implemented in Tools Plus. If you see "Button: Tabs", this means that the control is implemented as a "button" in Tools Plus, that is, you use button related routines to access the control, and Tools Plus reports events related to this control as a doButton event. The control actually appears as the Appearance Manager's "Tab Control." Similarly, controls that are implemented in Tools Plus as "scroll bars" may actually be sliders or other "scroll bar like" controls.

CDEF ID	ProcIDs	Control is Implemented in Tools Plus as a…
0	0 to 15	Button
1	16 to 31	Scroll Bar
2	32 to 47	Button: Bevel Button (If "value" parameter is zero. Available only in Appearance Manager).
		Pop-Up Menu using a Bevel Button body (If "value" parameter specifies a non-zero menu resource ID. Available only in Appearance Manager).
3	48 to 63	Scroll Bar: Slider (available only in Appearance Manager)
4	64 to 79	Button: Disclosure Triangles (available only in Appearance Manager)
5	80 to 95	Scroll Bar: Progress Indicator (available only in Appearance Manager)
6	96, 98-111	Button: Little Arrows (available only in Appearance Manager) Used for *stepping* through values.
	97	Scroll Bar: Little Arrows (available only in Appearance Manager) Used for stepping through values when clicked, and *scrolling* through values when held.
7	112 to 127	Button: Chasing Arrows (available only in Appearance Manager)
8	128 to 143	Button: Tabs (available only in Appearance Manager 1.0.1 or later)
9	144 to 159	Button: Visual Separator (available only in Appearance Manager)
10	160 to 175	Button: Group Box (available only in Appearance Manager)
11	176 to 191	Button: Image Well (available only in Appearance Manager)
12	192 to 207	Button: Pop-Up Arrows (available only in Appearance Manager)
13	208 to 223	Button: *(reserved by Apple for future consideration)*
14	224 to 239	Button: Placard (available only in Appearance Manager)
15	240 to 255	Button: Clock Control (available only in Appearance Manager)
16	256 to 271	Button: User Pane (available only in Appearance Manager)
17	272 to 287	Editing field (available only in Appearance Manager)

CDEF ID	ProcIDs	Control is Implemented in Tools Plus as a…
18	288 to 303	Button: Static Text (if SetDialogCNTLStaticTextSpec is set to -1) (available only in Appearance Manager)
		Static Text field (if SetDialogCNTLStaticTextSpec is not set to -1) (available only in Appearance Manager)
19	304 to 319	Button: Picture Button (available only in Appearance Manager)
20	320 to 335	Button: Icon Control (available only in Appearance Manager)
21	336 to 351	Button: Window Header (available only in Appearance Manager 1.0.2 or later)
22	352 to 367	List Box (remember to include an 'ldes' resource in your application if you want to create an Appearance Manager list box control)
23	368 to 383	Button: 3D Button (available only in Appearance Manager)
24	384 to 399	Scroll Bar: 3D Scroll Bar (available only in Appearance Manager)
25 to 62	400 to 1007	Button: *(reserved by Apple for future consideration)*
63	1008 to 1023	Pop-Up Menu (if SetDialogCNTLPopUpSpec is not set to -1)
		Button: Pop-up menu (if SetDialogCNTLPopUpSpec is set to -1)
64-127	1024 to 2047	Button: *(reserved by Apple for future consideration)*
128-1023	2048 to 16383	Button (third party button CDEF IDs should be in this range)
1024-2047	16384 to 32767	Scroll Bar (if control does not have a title). Third party scroll bar or slider CDEF IDs should be in this range.
		Button (if control has a title)

- See the following routines later in this chapter for options on translating 'CNTL' resources to Tools Plus user interface elements: SetDialogCNTLEditTextSpec, SetDialogCNTLStaticTextSpec, SetDialogCNTLListBoxSpec, and SetDialogCNTLPopUpSpec.
- See the following routines later in this chapter for options on translating Edit Text items to Tools Plus editing fields, and for translating Static Text items to Tools Plus static fields: SetDialogEditTextSpec and SetDialogStaticTextSpec.
- When a dialog is opened, its font, font size and style settings are set to the values specified by the SetDialogFontInfo routine. By default, the font is Chicago 12pt.
- Edit Text items are translated into editing fields using the window's current font settings as defined by the SetDialogFontInfo routine. Tools Plus creates fields using the NewDialogField routine, and they remember their font settings even when the window's font settings are changed. These fields are very similar to plain Dialog Manager fields. The attributes that are automatically assigned to each field are as follows (see the NewField routine for details). You can override these settings using the SetDialogEditTextSpec routine.

  handle = nil — A text handle is automatically allocated to store the field's text.
  left aligned — The field's text is left aligned.
  not filtered — No text filter is applied to the field.
  teSystemBody — Use Appearance Manager's Edit Text control if available. Dim field when it is disabled or on an inactive window.
  teUseWFont — Use window's font settings.
  teBoxCR — A box is drawn around the field and line breaks are allowed unless the field is one line high.
  teCstring — The field's string is stored as a C string to allow up to 32767 characters.
  teTabSelectAll — Select all text in field when user tabs into the field.
  teBuffered — The field is buffered with its own TextEdit record to increase performance when dealing with large volumes of text.
- Static text items that use an unspecified text transfer mode, or any modes other than srcCopy (source copy), are drawn by Tools Plus when the window is populated and when the window needs to be refreshed. These items are inaccessible to your application.
- If you want to alter static text items programatically and your dialog has an 'ictb' resource or 'dctb' resource, the static text item must use the srcCopy (source copy) text transfer mode. When this is done, Tools Plus creates the static text item as a static text field thereby letting you use Tools Plus's field routines to easily manipulate the static text item. This also gives the static text field the ability to dim when it is on an inactive window. The attributes that are automatically assigned to the static text field are as follows (see the NewField routine for details). You can override these settings using the SetDialogStaticTextSpec routine.

handle = nil	A text handle is automatically allocated to store the field's text
left aligned	The field's text is left aligned
not filtered	No text filter is applied to the field
teSystemBody	Use Appearance Manager's Edit Text control if available. Dim field when it is disabled or on an inactive window.
teStaticText	Create a static text field.
teUseWFont	Use window's font settings.
teAllowCR	Allow line breaks in the text.
teBackdrop	Use window's backdrop color for text background.

- Icons are displayed using the DrawIcon routine. The advantage this provides is that the perfect icon is displayed regardless of the monitor's settings ('cicn', 'icl8', 'icl4', etc.), and a mask is used if available. Although ResEdit only displays 'ICON' icons when you are designing your dialog, Tools Plus displays *any* icon when your dialog is displayed, including 'cicn', and large and small color and black and white icons. Note that ResEdit has a bug where it changes your resource ID number for an icon in the dialog's item list if an 'ICON' resource does not exist with that ID, and if the item order is renumbered. If you initialize Tools Plus with the initPureAppearanceManager option, icons are translated into non-selectable icon controls that dim on inactive windows.
- Pictures are displayed using the DrawPict routine. The picture is scaled to the item's display rectangle. If you need more sophisticated picture drawing such as using multiple pictures depending on monitor settings, clipping an image or using the picture's rectangle, define the picture in your dialog as a user item and draw the picture using the DrawPict routine when the dialog is created and when it needs to be refreshed. If you initialize Tools Plus with the initPureAppearanceManager option, pictures are translated into non-selectable picture controls that dim on inactive windows.
- If you need to create a user interface element that is not a standard item in a dialog item list, such as a list box, pop-up menu or a panel, define the item's co-ordinates in the dialog item list as a user item. After you create the dialog, use the GetDialogItemRect routine to obtain the display rectangle for that item, then create the element using a Tools Plus routine such as NewListBoxRect.

Also see: LoadWindow and LoadSpecDialog.

LoadSpecDialog

Open a dialog (window with user interface elements) using a 'DLOG' resource.

C
```
pascal void LoadSpecDialog (short Window, long Spec, short ResID);
```

Pascal
```
procedure LoadSpecDialog (Window: INTEGER; Spec: LONGINT; ResID: INTEGER);
```

LoadSpecDialog is identical to the LoadDialog routine, except that it requires the additional *Spec* parameter to give you control over all the appearance and behavior options offered by Tools Plus. See the NewWindow routine for details about the Spec parameter.

LoadSpecDialogBehind

Open a dialog using a ‘DLOG’ resource behind another window.

C
```
pascal void LoadSpecDialogBehind (short Window, long Spec, short ResID,
            short BehindWindow);
```

Pascal
```
procedure LoadSpecDialogBehind (Window: INTEGER; Spec: LONGINT;
            ResID, BehindWindow: INTEGER);
```

LoadSpecDialogBehind is identical to the LoadSpecDialog routine, except that it places the new window behind a specified window, or at the back of a window layer.

BehindWindow specifies the number of an open and visible Tools Plus window behind which the new window is created. This window must be in the same window layer as the new window, meaning that a floating palette can only open behind another floating palette, and a standard window can only open behind another standard window. A value of -1 opens the new window at the front of its layer. This produces the same result as using LoadSpecDialog. A value of 0 opens the new window at the back of its layer to make it the rear-most floating palette or standard window.

If the window specified by BehindWindow is not a floating palette or a standard modeless window, or if the window is hidden or closed, LoadSpecDialogBehind creates the new window at the front of its layer as though you had used LoadSpecDialog.

LoadDialogList

Attach a dialog item list (‘DITL’ resource) to an open window.

C
```
pascal void LoadDialogList (short Window, short ResID);
```

Pascal
```
procedure LoadDialogList (Window: INTEGER; ResID: INTEGER);
```

LoadDialogList reads a ‘DITL’ (dialog item list) resource, attaches it to an open window, then populates the window with the items in the list. You use this routine if you want to open a window and prepare it in some way before populating it with the user interface elements defined in a dialog item list.

Window specifies the window number that will have the item list attached. If the specified window is not open, LoadDialogList does nothing. If the window already has an item list attached, the old list and the items in it are deleted before the new one is attached. If the window is not open, LoadDialogList does nothing.

ResID is the ‘DITL’ (dialog item list) resource ID number that is used to create the dialog’s items. If the ‘DITL’ resource does not exist, LoadDialogList does nothing. Flag your ‘DITL’ resource as purgeable since Tools Plus makes a copy of its data.

Also see: LoadDialog and AppendDialogList.

AppendDialogList

Append a dialog item list (‘DITL’ resource) to a dialog.

C
```
pascal void AppendDialogList (short Window, short ResID);
```

Pascal
```
procedure AppendDialogList (Window: INTEGER; ResID: INTEGER);
```

AppendDialogList is similar to LoadDialogList in that it reads a ‘DITL’ (dialog item list) resource, and attaches it to the current window. Unlike the LoadDialogList routine, this routine appends the new items to the end of an existing dialog list in the current window instead of deleting it like LoadDialogList.

Window specifies the window number that will have the item list appended. If the specified window is not open, AppendDialogList does nothing. If the window already has an item list, the new list is appended to the end of the existing list. If the window is not open, AppendDialogList does nothing.

ResID is the 'DITL' (dialog item list) resource ID number that is used to create the new dialog items. If the 'DITL' resource does not exist, LoadDialogList does nothing. Note that the item numbers will not be the same as those you see in ResEdit because they are being added to the end of an existing item list. The item number of the first item in the new 'DITL' list is the current number of items plus one. Flag your 'DITL' resource as purgeable since Tools Plus makes a copy of its data.

Also see: LoadDialog and LoadDialogList.

SetDialogEditTextSpec

Set appearance and behavior specifications for editing fields that are created by dialogs as Edit Text items.

C
```
pascal void SetDialogEditTextSpec (long Spec);
```

Pascal
```
procedure SetDialogEditTextSpec (Spec: LONGINT);
```

Spec is the appearance and behavior specification that is used to create an editing field from an Edit Text item in a dialog. A list of possible values can be found in the NewField description.

When a window is opened, it takes a copy of the *Spec* parameter and uses it for all edit text items created in that window. This excluded editing fields that are created by 'CNTL' resources.

SetDialogStaticTextSpec

Set appearance and behavior specifications for static text fields that are created by dialogs as Static Text items.

C
```
pascal void SetDialogStaticTextSpec (long Spec);
```

Pascal
```
procedure SetDialogStaticTextSpec (Spec: LONGINT);
```

Spec is the appearance and behavior specification that is used to create a static text field from a Static Text item in a dialog. A list of possible values can be found in the NewField description.

When a window is opened, it takes a copy of the *Spec* parameter and uses it for all static text items created in that window. This excluded static text fields that are created by 'CNTL' resources.

SetDialogCNTLEditTextSpec

Set the appearance and behavior specifications for editing fields that are created in dialogs using 'CNTL' resources.

C
```
pascal void SetDialogCNTLEditTextSpec (long Spec);
```

Pascal
```
procedure SetDialogCNTLEditTextSpec (Spec: LONGINT);
```

Spec is the appearance and behavior specification that is used to create an editing field from a 'CNTL' resource in a dialog. See the Fields chapter for details about creating editing fields using 'CNTL' resources. A list of possible values can be found in the NewField description.

When a window is opened, it takes a copy of the *Spec* parameter and uses it for all editing fields created in that window from 'CNTL' resources. The *Spec* value can be overridden for a single item by its 'CNTL' resource.

SetDialogCNTLStaticTextSpec

Set the appearance and behavior specifications for static text fields that are created in dialogs using ‘CNTL’ resources.

C
```
pascal void SetDialogCNTLStaticTextSpec (long Spec);
```

Pascal
```
procedure SetDialogCNTLStaticTextSpec (Spec: LONGINT);
```

Spec is the appearance and behavior specification that is used to create a static text field from a ‘CNTL’ resource in a dialog. See the Fields chapter for details about creating static text fields using ‘CNTL’ resources. A list of possible values can be found in the NewField description. A value of -1 indicates that ‘CNTL’ resources referencing CDEF ID 18 (the static text control) are implemented as “buttons” instead of static text fields when the Appearance Manager is available. This may offer greater control to the programmer, but with less ease of use.

When a window is opened, it takes a copy of the *Spec* parameter and uses it for all static text items created in that window from ‘CNTL’ resources. The *Spec* value can be overridden for a single static text item by its ‘CNTL’ resource.

SetDialogCNTLListBoxSpec

Set the appearance and behavior specifications for list boxes that are created in dialogs using ‘CNTL’ resources.

C
```
pascal void SetDialogCNTLListBoxSpec (long Spec);
```

Pascal
```
procedure SetDialogCNTLListBoxSpec (Spec: LONGINT);
```

Spec is the appearance and behavior specification that is used to create a list box from a ‘CNTL’ resource in a dialog. See the List Boxes chapter for details about creating list boxes using ‘CNTL’ resources. A list of possible values can be found in the NewListBox description.

When a window is opened, it takes a copy of the *Spec* parameter and uses it for all list boxes created in that window from ‘CNTL’ resources. The *Spec* value can be overridden for a single list box by its ‘CNTL’ resource.

SetDialogCNTLPopUpSpec

Set the appearance and behavior specifications for pop-up menus that are created in dialogs using ‘CNTL’ resources.

C
```
pascal void SetDialogCNTLPopUpSpec (long Spec);
```

Pascal
```
procedure SetDialogCNTLPopUpSpec (Spec: LONGINT);
```

Spec is the appearance and behavior specification that is used to create a pop-up menu from a ‘CNTL’ resource in a dialog. See the Pop-Up Menus chapter for details about creating pop-up menus using ‘CNTL’ resources. A list of possible values can be found in the NewPopUp description. A value of -1 indicates that ‘CNTL’ resources referencing CDEF ID 63 (the pop-up menu) are implemented as “buttons” instead of pop-up menus. This offers greater control to the programmer, but considerably less ease of use.

When a window is opened, it takes a copy of the *Spec* parameter and uses it for all pop-up menus created in that window from ‘CNTL’ resources. The *Spec* value can be overridden for a single pop-up menu by its ‘CNTL’ resource.

ToolBarOpen

Open a tool bar and make it the “current” window.

C
```
pascal void ToolBarOpen (short Window, short Height, long Spec);
```

Pascal
```
procedure ToolBarOpen (Window: INTEGER; Height: INTEGER; Spec: LONGINT);
```

ToolBarOpen is used to open your application’s tool bar beneath the menu bar on your main monitor. Your application can have only one tool bar.

Window specifies the tool bar’s window number (just like an ordinary window). Once the tool bar is opened, it is referenced by this window number. If a window using the same window number is already open, it is closed, then a new tool bar is opened as specified by the parameters in the ToolBarOpen routine, thereby re-using the window number. The newly opened tool bar is always “active,” and it becomes “current” (action pertaining to graphics, text, buttons, scroll bars, etc. occurs in this window). Tools Plus allows up to 250 windows to be open simultaneously, however, you may choose to reduce this limit when using InitToolsPlus to initialize Tools Plus. If a tool bar is open and you try to open another toolbar using a different window number, ToolBarOpen does nothing.

Height specifies the tool bar’s height. A 1-pixel window frame is drawn just below the tool bar. The tool bar’s height can be up to 70 pixels, but application’s typically have tool bars that seldom exceed 30 pixels.

The tool bar’s *Spec* influences the behavior of other windows in your application. Two constants are available to assist in the implementation of a tool bar, either or both of which can be used to specify a tool bar spec parameter. If you decide not to use either of the available options, specify a Spec of 0.

`tbShiftWindows`	Shift all open windows downward by an amount that is equal to the menu bar’s height to prevent windows from being obscured by the tool bar. When the tool bar is closed, the windows are shifted up by the identical amount.
`tbOffsetNewWindows`	If new windows are opened while the tool bar is open, offset their co-ordinates downward by an amount that is equal to the menu bar’s height to prevent windows from being obscured by the tool bar. This option lets you use a standard set of window locations and have them automatically offset depending on whether the tool bar is open or not.

Tool bars are typically colored a medium gray on color or gray-scale monitors, so you may want to declare a global variable of type RGBColor (appropriately named ToolBarGray) that has the red, green and blue components set to 52,428. You can set a tool bar window’s backdrop color to ToolBarGray.

Tool bar inside a window

If you want to include a tool bar inside a window, do the following steps when the window is first opened and in response to a doPreRefresh event:

```
procedure DrawToolBar;
  var
    ToolBarRect: rect;                         {Tool bar's rectangle inside a window          }
    ToolBarColor: RGBColor;                    {Tool bar's color                              }
  begin
    SetRect(ToolBarRect, -1, -1, 10000, 40);   {Tool bar is 40 pixels high. Left, top and     }
                                               {  right side are out of view (no border seen) }
    SetRGB(ToolBarColor, 52428, 52428, 52428); {Set tool bar's color (best as a global var)   }
    PenColorNormal;                            {Pen: 1x1, black on white                      }
    SetBackRGB(ToolBarColor);                  {If Color QuickDraw used, set background to the }
                                               {  tool bar's color.  Maps to white on a       }
                                               {  monochrome monitor.                         }
    EraseRect(ToolBarRect);                    {Erase tool bar using background color         }
    FrameRect(ToolBarRect);                    {Frame tool bar using foreground color (black) }
  end;
```

```
CONST                                {Tool Bar options:                   }
          tbShiftWindows     =$01;   {Shift windows down when tool bar opens  }
          tbOffsetNewWindows =$02;   {Offset future windows when they open    }
```

GetFreeWindowNum

Get the first unused window number.

C
```
pascal short GetFreeWindowNum (void);
```

Pascal
```
function GetFreeWindowNum: INTEGER;
```

Some developers may prefer to write code that more closely resembles a traditional Macintosh application, in that creating an object returns a reference to it such as a handle or pointer. Instead of having to assign your own window number, GetFreeWindowNum returns the first unused (free) window number. Using this routine, you can assign an unused window number to a variable, then use that variable throughout your application without concern for the true window number.

If the maximum number of windows has already been opened (no new ones can be created), GetFreeWindowNum returns a value of zero (0).

BackdropColor

Set the backdrop color for new windows as they are opened.

C
```
pascal void BackdropColor (const RGBColor *Color);
```

Pascal
```
procedure BackdropColor (Color: RGBColor);
```

By default, windows adopt a white backdrop when they are opened, however, each window can have its own unique backdrop color. When you use the BackdropColor routine, new windows will adopt the specified backdrop color as they are opened.

Color is the color that is adopted as a backdrop by windows that are opened after using this routine.

If you want to reset this color to the default white, you can use the NoBackdropColor routine.

Also see: SetBackdropColor and NoBackdropColor.

Programming Tips:

1. Use pale, neutral colors for the best looking windows. A light gray is often the best.
2. A tool bar or floating palette appears to come up faster if you give it a backdrop that is medium gray, which is typically the color of its buttons.

NoBackdropColor

Reset the backdrop color to white for new windows as they are opened.

C
```
pascal void NoBackdropColor (void);
```

Pascal
```
procedure NoBackdropColor;
```

By default, windows adopt a white backdrop when they are opened, however, each window can have its own unique backdrop color as set by the BackdropColor routine. NoBackdropColor causes new windows to adopt the default white backdrop as they are opened. This is the equivalent to using BackdropColor(White).

Also see: BackdropColor.

SetBackdropColor

Set the backdrop color for an open window.

C
```
pascal void SetBackdropColor (short Window, const RGBColor *Color);
```

Pascal
```
procedure SetBackdropColor (Window: INTEGER; Color: RGBColor);
```

This routine is similar to the BackdropColor routine in that it sets a window's backdrop color, however SetBackdropColor can be used to change a window's backdrop color at any time. The window's content is erased with the new backdrop color and the window is invalidated to force all objects to be refreshed on the new backdrop.

Window specifies the window number that is affected. If the specified window is not open or if Color QuickDraw is not used, SetBackdropColor does nothing.

Color is the window's new backdrop color.

Also see: BackdropColor.

SetBackgroundTheme

Set the background theme for an open window.

C
```
pascal void SetBackgroundTheme (short Window, short ActiveBrush,
            short InactiveBrush);
```

Pascal
```
procedure SetBackgroundTheme (Window, ActiveBrush, InactiveBrush: INTEGER);
```

This routine should be used to set a window's background brush (the color or pattern of the theme that is being used by the Appearance Manager) to a brush that is different from the window's default brush. Such is the case in a document window because the Appearance Manager fills the window with white and you may want to use a brush that is normally associated with a modeless dialog. Do not change brushes after a window is open and displayed (not hidden) because this may look confusing to the user, and an unsightly flash may be seen as the window changes brushes. You can easily set a window to use its default brush when the window opens by adding the appropriate option in the spec parameter of the routine that opens the window. SetBackgroundTheme does nothing if the Appearance Manager is not available.

Once a theme is applied to a window's background, you cannot remove it nor set the window's backdrop color. You can only change to another background brush using the SetBackgroundTheme routine.

Window specifies the window number that is affected. If the specified window is not open or if the Appearance Manager is not available, SetBackgroundTheme does nothing.

ActiveBrush is the background brush that is used when the window is active. If an invalid brush is specified, SetBackgroundTheme does nothing.

InactiveBrush is the background brush that is used when the window is inactive. If an invalid brush is specified, SetBackgroundTheme does nothing.

Also see: SetNextWindowBackgroundTheme

```
CONST                                                          {Appearance Manager's brushes:  }
               kThemeActiveDialogBackgroundBrush          = 1;   {                               }
               kThemeInactiveDialogBackgroundBrush        = 2;   {                               }
               kThemeActiveAlertBackgroundBrush           = 3;   {                               }
               kThemeInactiveAlertBackgroundBrush         = 4;   {                               }
               kThemeActiveModelessDialogBackgroundBrush  = 5;   {                               }
               kThemeInactiveModelessDialogBackgroundBrush = 6;  {                               }
               kThemeActiveUtilityWindowBackgroundBrush   = 7;   {                               }
               kThemeInactiveUtilityWindowBackgroundBrush = 8;   {                               }
               kThemeListViewSortColumnBackgroundBrush    = 9;   {                               }
```

```
kThemeListViewBackgroundBrush          = 10;  {                                }
kThemeIconLabelBackgroundBrush         = 11;  {                                }
kThemeListViewSeparatorBrush           = 12;  {                                }
kThemeChasingArrowsBrush               = 13;  {                                }
kThemeDragHiliteBrush                  = 14;  {                                }
kThemeDocumentWindowBackgroundBrush    = 15;  {                                }
kThemeFinderWindowBackgroundBrush      = 16;  {                                }
```

SetNextWindowBackgroundTheme

Set the background theme for the next window that is opened.

C
```
pascal void SetNextWindowBackgroundTheme (short ActiveBrush,
            short InactiveBrush);
```

Pascal
```
procedure SetNextWindowBackgroundTheme (ActiveBrush, InactiveBrush: INTEGER);
```

This routine is similar to the SetBackgroundTheme, except that it sets the background theme only for the next window that is opened. Use this routine just before you open a window if you want to create a window that has a non-standard background theme. A good example of this is if you want to create a non-growing document window (noGrowDocProc), but you want it to have a medium gray backdrop in Apple's Platinum theme, or an equivalent tone in other themes. The kThemeActiveModelessDialogBackgroundBrush (brush number 5) has this characteristic, so you would call SetNextWindowBackgroundTheme(5,5) just before you open the new window.

Also see: SetBackgroundTheme

WindowClose

Close an open window, tool bar or floating palette.

C
```
pascal void WindowClose (short Window);
```

Pascal
```
procedure WindowClose (Window: INTEGER);
```

The WindowClose routine closes a window that was opened by WindowOpen, or a tool bar that was opened with ToolBarOpen. If a standard window is being closed, the window immediately behind the newly closed window (if one exists) becomes active and current.

Window specifies the window number that is closed. If the specified window is not open, WindowClose does nothing.

When a window is closed, it automatically deletes the user interface elements on that window and in doing so, releases the memory consumed by them. Specifically, those elements are buttons, picture buttons, pop-up menus, scroll bars, editing fields, list boxes, panels and custom controls. Fields that have automatically allocated a string handle deallocate that handle as they are deleted when the window is closed. Cursor tables are not deleted because they can be shared by multiple windows. Field filters are also not deleted since any filter can be used by numerous fields spread across multiple windows.

If the affected window contains an active editing field, that field is automatically deactivated before the window is closed. The impact to your application is that you must save the field's edited text (with the SaveFieldString routine) before closing the window. If you want to validate the field's edited text before saving it, see the GetEditString routine.

When working with windows that are opened and closed frequently, you can take advantage of the WindowDisplay routine which hides and displays a window. This is particularly useful when used on a tool bar or a floating palette because a hidden window remembers the settings of all the objects on the window (picture buttons, check boxes and radio buttons, editing fields, etc.) as well as the window's location. When the window is displayed again, it is identical in position and appearance as when it was hidden.

WindowSize

Change a window's size or a tool bar's height.

C
```
pascal void WindowSize (short Window, short Width, short Height,
            Boolean Update);
```

Pascal
```
procedure WindowSize (Window, Width, Height: INTEGER; Update: BOOLEAN);
```

The WindowSize routine is used to change a window's width and/or height without changing its position on the screen. In most situations, windows that need resizing are best accommodated by the documentProc procID which provides a grow box in the bottom right corner of the window, and optionally a zoom box in the title bar. Some applications, however, need a window that presents an "expanded" view. An example of this is the Macintosh's Alarm Clock desk accessory which expands to let the user change the time, calendar and alarm timer. Your application should change a window's size only in response to some action taken by the user. This routine does not resize windows that are collapsed with System 7's WindowShade or in Mac OS 8 due to an OS bug that redraws the window structure improperly.

Window specifies the window number that is resized. If the specified window is not open, WindowSize does nothing.

Width and *Height* specify the window's new dimensions in pixels. These dimensions relate to those specified by the WindowOpen routine, that is, they represent the content region or *usable* area of the window (the window's frame, shadow, and title bar are all created outside of these co-ordinates). The tool bar's width cannot be changed, and its height cannot exceed 70 pixels. All other windows' new dimensions are automatically adjusted to keep them within the window's size limits which are set with SetWindowSizeLimits. If you specify zero (0) for either of these dimensions, it specifies that the dimension (height or width) should not be changed (i.e., WindowSize(1, 80, 0, true) changes window 1's width to 80 pixels and leaves the height unchanged).

Update specifies if the newly exposed area is added to the window's update region (thereby producing a doRefresh event). If you specify *true*, the newly exposed area is added to the window's update region. If you specify *false*, your application will handle the newly exposed area.

WindowMove

Move a window to another location on the screen.

C
```
pascal void WindowMove (short Window, short hGlobal, short vGlobal,
            long Spec);
```

Pascal
```
procedure WindowMove (Window: INTEGER; hGlobal: INTEGER; vGlobal: INTEGER;
            Spec: LONGINT);
```

WindowMove repositions a window on the screen without changing its size. Do not use WindowMove in place of having the user move a window to a new location by dragging the title bar. Windows should only be moved in response to a user's action, such as selecting an "Arrange Windows" menu item that arranges all open windows in a specified manner (grid or tile, for example).

Window specifies the window number that is moved. If the specified window is not open, WindowMove does nothing.

HGlobal and *vGlobal* specify the window's new location in global co-ordinates. These dimensions relate to the top left corner specified by the WindowOpen routine, that is, they represent the top left corner of the content region or *usable* area of the window (the window's frame, shadow, and title bar are all created outside of these co-ordinates). The tool bar cannot be moved.

Spec specifies optional behavior that can take place while moving the window. The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result. The options are:

`wAnimateMove`	Show “zoom lines” that move from the window’s original position to the window’s new position (see the ZoomLines routine for details).
`wOffsetForToolBar`	Offset the specified vertical co-ordinate downward by the tool bar’s height (if a tool bar is open).

```
CONST                                {Window moving options:                }
              wAnimateMove      = $01;   {Animate with Zoom Lines               }
              wOffsetForToolBar = $02;   {Offset co-ords by tool bar's height    }
```

WindowDisplay

Hide or show a window, floating palette, or tool bar.

C
```
pascal void WindowDisplay (short Window, Boolean Show);
```

Pascal
```
procedure WindowDisplay (Window: INTEGER; Show: BOOLEAN);
```

From a user’s perspective, a window that is “hidden” by your application is actually being closed. If a standard window is hidden, the standard window behind it is activated. When a window is displayed (unhidden), it appears as though the window was opened, in that it appears at the front of its layer and becomes “active” and “current.” Your application should hide and show windows in response to a user’s action, such as selecting a “Hide Tool Bar” menu item. Tools Plus automatically hides the tool bar and all floating palettes when your application is suspended under MultiFinder or System 7 or higher, and displays them when your application is activated.

Window specifies the window number that is hidden or shown.

Show indicates if the window is being hidden or displayed. The two constants that can be used for this flag are *wShow* and *wHide*.

When working with windows that are opened and closed frequently, you can take advantage of WindowDisplay instead of closing the window and opening it and having to recreate its contents. This is particularly useful when used on a tool bar or a floating palette because a hidden window remembers the settings of all the objects on the window (picture buttons, check boxes and radio buttons, editing fields, etc.) as well as the window’s location. When the window is displayed again, it is identical in position and appearance as when it was hidden.

When a window is hidden, it does not release any memory consumed by its associated user interface elements such as buttons (including radio buttons and check boxes), picture buttons, pop-up menus, scroll bars, editing fields, list boxes, and custom controls. If a window is being hidden and it contains an active editing field, that field is automatically deactivated before the window is hidden. The impact to your application is that you must save the field’s edited text (with the SaveFieldString routine) before hiding the window. If you want to validate the field’s edited text before saving it, see the GetEditString routine.

```
CONST                         {Window displaying options:       }
              wShow = true;      {Display (unhide)                 }
              wHide = false;     {Hide window                      }
```

Programming Tips:

1 Before you hide a window, realize that the user thinks the window is being *closed* (even though it may only be temporary). Don’t leave the window in an “unsettled” state when your are hiding it. Make sure that all editing fields are validated and processed.

WindowDisplayBehind

Hide or show a window, floating palette, or tool bar. When showing a window, show it behind another window.

C
```
pascal void WindowDisplayBehind (short Window, Boolean Show,
            short BehindWindow);
```

Pascal
```
procedure WindowDisplayBehind (Window: INTEGER; Show: BOOLEAN;
            BehindWindow: INTEGER);
```

WindowDisplayBehind is identical to the WindowDisplay routine, except when showing a window, this routine shows it behind a specified window, or at the back of a window layer.

BehindWindow specifies the number of an open and visible Tools Plus window behind which the new window is shown. This parameter is ignored when hiding a window. The BehindWindow must be in the same window layer as the window that is being displayed, meaning that a floating palette can only display behind another floating palette, and a standard window can only display behind another standard window. A value of -1 displays the window at the front of its layer. This produces the same result as using WindowDisplay. A value of 0 displays the window at the back of its layer to make it the rear-most floating palette or standard window.

If the window specified by BehindWindow is not a floating palette or a standard modeless window, or if the window is hidden or closed, WindowDisplayBehind displays the window at the front of its layer as though you had used WindowDisplay.

ActivateWindow

Activate a window.

C
```
pascal void ActivateWindow (short Window);
```

Pascal
```
procedure ActivateWindow (Window: INTEGER);
```

Window specifies the window number that is activated. The specified window is brought forward and becomes “active” and “current.” This window is also considered to be the “work” window. If the window is hidden or not open, ActivateWindow does nothing. You cannot activate a modal window.

If the tool bar is activated, it simply becomes the current window (because the tool bar is always active). If a floating palette is activated, it is brought to the front of the floating palette layer without deactivating any windows. When a standard window is activated, it is brought to the front of the standard window layer, and the previously active standard window is deactivated.

A window is normally activated only in response to a doChgWindow event that is reported to your event handler routine. Another possible use of ActivateWindow is if your application has a “Window” menu that lets the user activate a window from a menu. Don’t mysteriously activate an inactive window.

ClearFocus

Remove the keyboard focus from a window.

C
```
pascal void ClearFocus (short Window);
```

Pascal
```
procedure ClearFocus (Window: INTEGER);
```

Window specifies the window number whose keyboard focus is being removed. If the window is hidden or not open, ClearFocus does nothing. If the keyboard focus is an editing field, the field is automatically deactivated.

CurrentWindow

Make a window the current window without activating it.

C
```
pascal void CurrentWindow (short Window);
```

Pascal
```
procedure CurrentWindow (Window: INTEGER);
```

Subsequent window related operations such as drawing, and creating fields, buttons, scroll bars, etc., occur in the specified window without making it the "active" window. This routine is used to redirect your application's actions to a window other than the active one.

Window specifies the window number in which subsequent window related operations occur. If the specified window is not open, CurrentWindow does nothing.

The CurrentWindowReset routine resets window operations back to the "active" window making the active window current too. You should get into the habit of leaving the active window current. It makes debugging much simpler.

CurrentWindowReset

Reset the "current" window to be the same as the "active" window.

C
```
pascal void CurrentWindowReset (void);
```

Pascal
```
procedure CurrentWindowReset;
```

Subsequent window related operations such as drawing, and creating fields, buttons, scroll bars, etc., occur in the "active" window. This routine nullifies the effect of the CurrentWindow routine making the "active" window current also. If your application uses a tool bar and/or floating palettes, then the work window becomes current.

WindowTitle

Change a window's title.

C
```
pascal void WindowTitle (short Window, const Str255 Title);
```

Pascal
```
procedure WindowTitle (Window: INTEGER; Title: STRING);
```

The WindowTitle routine changes the title for an open window, regardless if it is active or not. You can only see the change on windows that have a title bar (documentProc, noGrowDocProc, rDocProc, paletteProc and ordPaletteProc). You won't see any change on windows that do not display titles (dBoxProc, plainDBox, altDBoxProc, altPaletteProc, and the tool bar).

Window specifies the window number in which the title is to be changed. The specified window does *not* have to be the "active" window, however, the window must be opened to display the title. Hidden windows display the new title once they become visible. If the window is not open, WindowTitle does nothing.

Title contains the window's new title.

✍ **Note**: When printing to a LaserWriter or any other printer that supports Print Monitor (or other spoolers), a temporary spool file is created. The Print Manager uses the active window's name to name the spool file (the file's name appears in the Print Monitor's queue to indicate the documents that are waiting to be printed). Before you do any printing, use WindowTitle to set the active window's title to the name you want your spool file to be. This applies even if you are using a modal dialog (which may not have a title bar) while printing.

SetWindowSizeLimits

Set a window's size limits that determine the minimum and maximum size allowable when using the "size box" or "zoom box."

C
```
pascal void SetWindowSizeLimits (short minHoriz, short minVert,
            short maxHoriz, short maxVert);
```

Pascal
```
procedure SetWindowSizeLimits (minHoriz, minVert, maxHoriz,
            maxVert: INTEGER);
```

MinHoriz specifies the minimum width (in pixels) the window may attain when being sized.

MinVert specifies the minimum height (in pixels) the window may attain when being sized.

MaxHoriz specifies the maximum width (in pixels) the window may attain when being sized.

MaxVert specifies the maximum height (in pixels) the window may attain when being sized.

SetWindowSizeLimits affects only the current window. If the current window is not a Tools Plus window, SetWindowSizeLimits does nothing. The minimum and maximum limits imposed on a window are automatically adjusted (if necessary) to ensure that the window's current size does not exceed the adjusted limits. For example, if the minHoriz limit is set to 100 pixels and the window is currently 90 pixels wide (10 pixels smaller than the specified minimum width), minHoriz is adjusted to 90 pixels. The same applies if the maximum limit is exceeded by the window's current dimensions.

By setting these limits, it is possible to allow a window to be sized horizontally or vertically only.

SetWindowZoom

Set a window's standard co-ordinates and user co-ordinates that are in effect when the window's "zoom box" is clicked.

C
```
pascal void SetWindowZoom (const Rect *userRect, const Rect *stdRect);
```

Pascal
```
procedure SetWindowZoom (userRect, stdRect: RECT);
```

A window containing a zoom box has two different states: [1] the standard state, and [2] the user state. The user can change the window's size and/or location, thereby defining the user state. When the zoom box is clicked, the window "zooms" back to the standard state (which, by default, is the window's co-ordinates when it was first opened). Clicking the zoom box again reverts to the user state.

Sometimes it is desirable to have the standard state and/or user state something other than the window's initial co-ordinates. SetWindowZoom sets either or both of these. The window's current co-ordinates become the user state. It is good form to call SetWindowZoom immediately after opening a window.

UserRect defines a rectangle in global co-ordinates that determines the window's user co-ordinates. If the current window is not a Tools Plus window, if the current window has no zoom box, or if an empty rectangle is specified, the user co-ordinates are not set.

StdRect defines a rectangle in global co-ordinates that determines the window's standard co-ordinates. If the current window is not a Tools Plus window, if the current window has no zoom box, or if an empty rectangle is specified, the standard co-ordinates are not set.

If the tool bar is open, and it was created with the tbOffsetNewWindows option, this window's co-ordinates for *userRect* and *stdRect* are shifted downwards by an amount that is equal to the tool bar's height.

☞ **Warning**: When you set the user and standard co-ordinates, make sure that they are such that at least part of the window's title bar is visible to allow the window to be dragging back into view (don't zoom to an "off-screen" window). Ideally, the zoom box should always be visible.

GetWindowZoom

Get a window's *standard state* and *user state* for zooming.

C
```
pascal void GetWindowZoom (Rect *userRect, Rect *stdRect);
```

Pascal
```
procedure GetWindowZoom (var userRect, stdRect: RECT);
```

A window containing a zoom box has two different states: [1] the standard state, and [2] the user state. The user can change the window's size and/or location, thereby defining the user state. When the zoom box is clicked, the window "zooms" back to the standard state (which, by default, is the window's co-ordinates when it was first opened). Clicking the zoom box again reverts to the user state.

UserRect defines the window's user state in the screen's global co-ordinates.

StdRect defines the window's standard state in the screen's global co-ordinates.

If the tool bar is open, and it was created with the tbOffsetNewWindows option, this window's *userRect* and *stdRect* are shifted upwards by an amount that is equal to the tool bar's height (i.e., they are shifted up as though there was no tool bar).

GetWindowZoom gets the values for the current window. If the current window is not a Tools Plus window, or if the current window has no zoom box, the userRect and stdRect rectangles are undefined. This routine is useful if you want to save both states as part of the document. When the document is opened, a window could be created using the userRect co-ordinates, and the user and standard state can be set by using the SetWindowZoom routine.

SetDialogItemRect

Set a dialog item's display rectangle.

C
```
pascal void SetDialogItemRect (short Item, const Rect *ItemRect);
```

Pascal
```
procedure SetDialogItemRect (Item: INTEGER; ItemRect: RECT);
```

SetDialogItemRect sets the display rectangle for a dialog item in the current window. The current window must be a "dialog," that is, a window that has been opened using LoadDialog or LoadSpecDialog. You can also populate a window with a dialog item list by using the LoadDialogList routine. See the LoadDialog routine for details about creating a dialog and its elements.

Item specifies the item number whose display rectangle is being changed. This number relates to the item numbers you see displayed while editing a 'DLOG' (dialog) resource or 'DITL' (dialog item list) resource in ResEdit.

ItemRect is the specified item's new display rectangle. If the current window does not have a dialog list, or if the item you specify does not exist in the item list, SetDialogItemRect does nothing.

This routine is useful only for changing the co-ordinates of static text items, icons, or pictures in a dialog, usually to "hide" them by moving their co-ordinates out of the visible part of the window. The change is visible next time the dialog is refreshed because the item is drawn at its new co-ordinates. SetDialogItemRect changes the item's co-ordinates and does nothing else, so you may want to erase the item at its old co-ordinates using the toolbox's EraseRect routine, and invalidate the area using the toolbox's InvalRect routine to force other objects within the area to be redrawn. After the item's co-ordinates are changed, use InvalRect at the new co-ordinates to force the item to be redrawn in its new position.

Tools Plus provides routines that let you move and/or resize any user interface element.

GetDialogItemRect

Get a dialog item's display rectangle.

C
```
pascal void GetDialogItemRect (short Item, Rect *ItemRect);
```

Pascal
```
procedure GetDialogItemRect (Item: INTEGER; var ItemRect: RECT);
```

GetDialogItemRect obtains the display rectangle for a dialog item in the current window. The current window must be a "dialog," that is a window that has been opened using LoadDialog or LoadSpecDialog. You can also populate a window with a dialog item list by using the LoadDialogList routine. See the LoadDialog routine for details about creating a dialog and its elements.

Item specifies the item number whose display rectangle is being retrieved. This number relates to the item numbers you see displayed while editing a 'DLOG' (dialog) resource or 'DITL' (dialog item list) resource in ResEdit.

ItemRect is the specified item's display rectangle. ItemRect returns as an empty rectangle (with all co-ordinates set to zero) if the current window was not created by Tools Plus, if the current window does not have a dialog list, or if the item you specify does not exist in the item list.

SetDialogFontInfo

Set the font settings for new dialogs as they are created.

C
```
pascal void SetDialogFontInfo (short theFont, short theSize, Style theStyle);
```

Pascal
```
procedure SetDialogFontInfo (theFont, theSize: INTEGER; theStyle: STYLE);
```

When new dialogs are created, by default they use the system's font (Chicago 12pt plain). SetDialogFontInfo lets you specify new default font settings that are adopted by dialogs as they are opened. GetDialogFontInfo lets you retrieve these settings.

TheFont specifies the font that is used by new dialogs. The default is Chicago, which is represented by the systemFont constant.

TheSize specifies the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle specifies the style(s) in which the font is displayed. Special character constants defined by the Font Manager are bold, italic, underline and shadow. C programmers use the font manager's constants to specify a composite style, such as SetDialogFontInfo(geneva, 9, bold + outline) for bold and outlined, or SetDialogFontInfo(geneva, 9, 0) for plain text. Pascal programmers use the font manager's constants to specify a set, such as SetDialogFontInfo(geneva, 9, [bold, outline]) for bold and outlined, or SetDialogFontInfo(geneva, 9, [ ]) for plain text.

GetDialogFontInfo

Get the font settings used by new dialogs as they are created.

C
```
pascal void GetDialogFontInfo (short *theFont, short *theSize,
            Style *theStyle);
```

Pascal
```
procedure GetDialogFontInfo (var theFont: INTEGER; var theSize: INTEGER;
            var theStyle: STYLE);
```

When new dialogs are created, by default they use the system's font (Chicago 12pt plain). GetDialogFontInfo lets you obtain the default font settings that are adopted by dialogs as they are opened.

TheFont specifies the font that is used by new dialogs.

TheSize specifies the font's size.

TheStyle specifies the style(s) in which the font is displayed. Special character constants defined by the Font Manager are bold, italic, underline and shadow. C programmers use the font manager's constants to specify a composite style, such as "bold + outline" for bold and outlined, or "0" for plain text. Pascal programmers use the font manager's constants to specify a set, such as [bold, outline] for bold and outlined, [ ] for plain text.

WindowStatus

Get a window's status information.

C
```
pascal void WindowStatus (short Window, TPWindowStatus *Status);
```

Pascal
```
procedure WindowStatus (Window: INTEGER; var Status: TPWindowStatus);
```

The WindowStatus routine returns the status of any Tools Plus window, whether it is open or closed, displayed or hidden.

Window is the window number of a Tools Plus window. Window must be less than or equal to *MaxWindows* as defined by InitToolsPlus. If it is not, the Status record is initialized to "false" and 0 values. *MaxWindows* defines the maximum number of Tools Plus windows that may be open at any time.

The *Status* record contains information about the Tools Plus window indicated by the *Window* value. The record is defined as such:

C
```
struct TPWindowStatus {
   short Kind;              /*Window kind: Tool Bar, Palette, or Standard       */
   Boolean Open;            /*Is the window open?                               */
   Boolean Visible;         /*Is the window visible (not hidden)?               */
   Boolean Active;          /*Is the window active?                             */
   Boolean Collapsed;       /*Is the window collapsed (by WindowShade)          */
   Boolean Front;           /*Is the frontmost Tools Plus window?               */
   Boolean Current;         /*Is the current window?                            */
   Boolean WorkWindow;      /*Is the work window?                               */
   Boolean EditFieldWindow; /*Does the window have app's active field?          */
   short ActiveField;       /*Window's active field number                      */
   Rect StrucRect;          /*Structure rect (global). Incl border & title bar  */
   Rect ContRect;           /*Content rect (global).  Working area only.        */
   };
typedef struct TPWindowStatus TPWindowStatus;
```

Pascal
```
TPWindowStatus = record      {                                                   }
   Kind: integer;            {Window kind: Tool Bar, Palette, or Standard        }
   Open: boolean;            {Is the window open?                                }
   Visible: boolean;         {Is the window visible (not hidden)?                }
   Active: boolean;          {Is the window active?                              }
   Collapsed: boolean;       {Is the window collapsed (by WindowShade)           }
   Front: boolean;           {Is the frontmost Tools Plus window?                }
   Current: boolean;         {Is the current window?                             }
   WorkWindow: boolean;      {Is the work window?                                }
   EditFieldWindow: boolean; {Does the window have the app's active field?       }
   ActiveField: integer;     {Window's active field number                       }
   StrucRect: rect;          {Structure rect (global).  Includes border & title bar}
   ContRect: rect;           {Content rect (global).  Working area only.         }
end;
```

Kind indicates the kind of window being referenced. The various kinds of windows are:

wNoKind	= 0	Window is not open
wToolBarKind	= 1	Tool Bar
wFloatingKind	= 2	Floating palette
wStandardKind	= 3	Standard window

Open indicates if the referenced window is open.

Visible indicates if the referenced window is visible or not. The term "visible" refers to being programatically unhidden. It does not mean "obscured by other windows or objects."

Active indicates if the referenced window is active. A Tools Plus window will not be active under any of the following conditions:
- the window is not open
- the referenced window is a standard window, but not the frontmost standard window
- the active window is a desk accessory (System 5/6's Finder only)

Collapsed indicates if the referenced window has been collapsed such that only its title bar is visible. Collapsing a window is available as a control panel called "Window Shade" in System 7, and as part of Mac OS 8. A collapsed window is visible to the user as a title bar only, even though the objects in the window still exist and are completely functional. For example, typing the Enter key will invoke the default button in an active collapsed window. A collapsed window cannot be resized using the WindowSize routine due to a bug in System software that does not redraw the window correctly.

Front indicates if the referenced window is the frontmost window in your application. If your application is not using a tool bar or floating palettes, the frontmost window is active unless:
- a desk accessory is active
- another application or the Finder is active (under MultiFinder or System 7 or higher)

Current indicates if the referenced window is the current window. If your application does not use a tool bar or floating palettes, *current* and *active* will be the same unless you used the CurrentWindow routine to change the current window number.

WorkWindow indicates if the referenced window is the work window. This is the same as *active* if your application does not use a tool bar or floating palettes.

EditFieldWindow indicates if the referenced window contains your application's active editing field (see the Editing Fields chapter for details).

ActiveField specifies the active editing field number for the referenced window. For standard windows, this field becomes active when the window is active. For the tool bar and floating palettes, this field is active while the application is active.

StrucRect is the window's structure rectangle in global co-ordinates. This includes the window's frame, shadow, and title bar.

ContRect is the window's content rectangle in global co-ordinates. This is the window's usable area excluding the window's frame, shadow, and title bar.

```
CONST                              {Kinds of windows:                }
            wNoKind       = 0;     {Not open                         }
            wToolBarKind  = 1;     {Tool Bar                         }
            wFloatingKind = 2;     {Floating Palette                 }
            wStandardKind = 3;     {Standard Window                  }
```

RefreshToolsPlusInWindow

Refresh Tools Plus user interface elements in a window.

C
```
pascal void RefreshToolsPlusInWindow (short Window);
```

Pascal
```
procedure RefreshToolsPlusInWindow (Window: INTEGER);
```

Window specifies the affected window number.

This routine refreshes Tools Plus user interface elements within the specified window's update region. Your application will typically never need to use this routine because it is automatically executed if it's required when your event handler routine is told to refresh a window. You may decide to use this routine in the following example:
- Display an alert indicating an error

- User dismisses the alert but your application must do some processing before it leaves the event handler routine.
- If your application draws anything in the window, draw those items from within a BeginUpdate/EndUpdate block.

When RefreshToolsPlusInWindow returns control to your application, the window's update region excludes the areas occupied by Tools Plus's user interface elements. This lets your application do additional drawing inside a BeginUpdate/EndUpdate block without worrying about overwriting any Tools Plus items. The interior of a panel is not affected in this way to permit your application to draw inside the panel if required.

RefreshDrawingInWindow

Refresh elements that are drawn by your application in a window.

C
```
pascal void RefreshDrawingInWindow (short Window);
```

Pascal
```
procedure RefreshDrawingInWindow (Window: INTEGER);
```

Window specifies the affected window number.

This routine refreshes the application-drawn elements in a window by issuing a doPreRefresh event followed by a doRefresh event to a window's event handler. Your application will likely call RefreshDrawingInWindow if it creates the window's interface elements dynamically . A typical sequence is as follows:

- Open window
- Create user interface elements such as buttons, sliders, list boxes, etc.
- Call RefreshDrawingInWindow

GetWindowInOrder

Determine the *N*th window from the front.

C
```
pascal short GetWindowInOrder (short Position);
```

Pascal
```
function GetWindowInOrder (Position: INTEGER): INTEGER;
```

The GetWindowInOrder routine can be used to determine the front to back order of Tools Plus windows. This is useful if you ever want to place a new window behind a specific window.

Position specifies the relative front-to-back position of the window you want to query. For example, 1 indicates the frontmost window, 2 indicates the second window from the font. In all cases, windows belonging to other applications or processes and desk accessories are ignored, as are hidden or closed windows. Only open and non-hidden windows that were opened with Tools Plus routines are counted, even if their co-ordinates are off-screen and they cannot be seen by the user. The relative position includes a tool bar, floating palettes and modal windows too.

The routine's value returns with a Tools Plus window number. If the specified position is less than one, or if it exceeds the total number of windows that are currently open in your application, GetWindowInOrder returns with a value of zero.

ActiveWindowNumber

Get the window number of the active window (or work window number if a tool bar and/or floating palettes are used).

C
```
pascal short ActiveWindowNumber (void);
```

Pascal
```
function ActiveWindowNumber: INTEGER;
```

This routine returns the window number of the active window when your application is the active application. If your application does not have a tool bar or floating palettes, this is the frontmost window. When a tool bar and/or floating palettes are used, ActiveWindowNumber returns the work window number. A value of zero (0) is returned if any of the following conditions occurs:

- no windows are open
- the active window is a desk accessory
- another application or the Finder is active (under MultiFinder or System 7 or higher)

Note that ActiveWindowNumber returns the same value regardless if your application is active or not. You can use the combination of ActiveWindowNumber and ApplicationSuspended to determine if the user sees this window as active or not.

Also see: CurrentWindowNumber, FirstWindowNumber, FirstStdWindowNumber, FirstPaletteNumber and WorkWindowNumber.

CurrentWindowNumber

Get the window number of the current window.

C
```
pascal short CurrentWindowNumber (void);
```

Pascal
```
function CurrentWindowNumber: INTEGER;
```

This routine returns the window number of the current window. If your application does not have a tool bar or floating palettes, this window is the same as the active window unless you used the CurrentWindow routine to change the current window. A value of zero (0) is returned if any of the following conditions occurs:

- no windows are open
- the current window is a desk accessory
- another application or the Finder is current (under MultiFinder or System 7 or higher)

Also see: ActiveWindowNumber and FirstWindowNumber.

FirstWindowNumber

Get the window number of your application’s frontmost window.

C
```
pascal short FirstWindowNumber (void);
```

Pascal
```
function FirstWindowNumber: INTEGER;
```

This routine is typically used to determine the frontmost window in order to close it or apply some equally universal operation to that window. If your application does not have a tool bar or floating palettes, this is your application’s frontmost window. If your application has a tool bar or floating palettes, FirstWindowNumber returns the number of the window that satisfies any of the following conditions (in ascending order of priority):

- the frontmost modal window (it is a standard window)
- the frontmost floating palette (if one is open and visible)
- the frontmost modeless standard window (if one is open and visible)

FirstWindowNumber always ignores the tool bar.

A value of zero (0) is returned if no windows are open. Note that the frontmost window in your application will not be the active window under any of the following conditions:

- a desk accessory is active
- another application or the Finder is active (under MultiFinder or System 7 or higher)

Also see: CurrentWindowNumber, FirstWindowNumber, FirstStdWindowNumber, FirstPaletteNumber and WorkWindowNumber.

ToolBarNumber

Get the window number of your application’s tool bar.

C
```
pascal short ToolBarNumber (void);
```

Pascal
```
function ToolBarNumber: INTEGER;
```

This routine returns the window number of your application’s tool bar. If your application does not have a tool bar, or if the tool bar has been hidden, ToolBarNumber returns a value of zero (0). You can use this routine in place of a global variable to determine if an event pertains to the tool bar.

FirstPaletteNumber

Get the window number of your application’s frontmost floating palette.

C
```
pascal short FirstPaletteNumber (void);
```

Pascal
```
function FirstPaletteNumber: INTEGER;
```

This routine returns the window number of your application’s frontmost visible floating palette. If your application does not have floating palettes, or if they are all hidden, FirstPaletteNumber returns a value of zero (0).

FirstStdWindowNumber

Get the window number of your application’s frontmost standard window.

C
```
pascal short FirstStdWindowNumber (void);
```

Pascal
```
function FirstStdWindowNumber: INTEGER;
```

This routine returns the window number of your application’s frontmost visible standard window. If your application does not have standard windows, or if they are all hidden, FirstStdWindowNumber returns a value of zero (0). Note that this window may be modal.

WorkWindowNumber

Get the window number of your application's work window.

C
```
pascal short WorkWindowNumber (void);
```

Pascal
```
function WorkWindowNumber: INTEGER;
```

This routine returns the window number of your application's work window. Your application has only one such window which gains its "work window" status under any of the following conditions:

- the user clicks in a window, or any object in a window
- a window is opened as modal (because the next action *must* take place within that window)
- a standard window is opened (and therefore activated), and the previous work window was an active standard window
- the work window is closed or hidden, in which case the following will become the work window:
  frontmost standard window (if any are open), or
  frontmost floating palette (if any are open), or
  the tool bar (if it is open)
- a window is activated

Your application can treat a work window like an active window, in that it is an eligible target for the user's activity. If your application does not use a tool bar or floating palettes, the work window is the same as the active window.

EditFldWindowNumber

Get the window number of the window containing your application's active editing field.

C
```
pascal short EditFldWindowNumber (void);
```

Pascal
```
function EditFldWindowNumber: INTEGER;
```

This routine returns the window number of the window containing the active editing field in your application. If your application does not have a tool bar or floating palettes, this window will either be the active window (frontmost), or it will be zero (0) when there is no active field. When a tool bar and/or floating palettes are used, this window can potentially be any of the active windows (tool bar, any floating palette, or the active standard window). See the Editing Fields chapter for details.

FocusWindowNumber

Get the window number of the window containing your application's keyboard focus.

C
```
pascal short FocusWindowNumber (void);
```

Pascal
```
function FocusWindowNumber: INTEGER;
```

This routine returns the window number of the window containing the your application's keyboard focus. If your application does not have a tool bar or floating palettes, this window will either be the active window (frontmost), or it will be zero (0) when there is no control bearing the keyboard focus. When a tool bar and/or floating palettes are used, this window can potentially be any of the active windows (tool bar, any floating palette, or the active standard window). See the Editing Fields chapter for details.

WindowIsOpen

Determine if a window is open.

C
```
pascal Boolean WindowIsOpen (short Window);
```

Pascal
```
function WindowIsOpen (Window: INTEGER): BOOLEAN;
```

Window is the window number of a Tools Plus window. Window must be less than or equal to *MaxWindows* as defined by InitToolsPlus.

The routine's value returns *true* if the window is open, and *false* if the window is not open. Note that an open window may have been hidden by your application, and therefore not be visible.

WindowIsVisible

Determine if a window is visible (not hidden).

C
```
pascal Boolean WindowIsVisible (short Window);
```

Pascal
```
function WindowIsVisible (Window: INTEGER): BOOLEAN;
```

Window is the window number of a Tools Plus window. Window must be less than or equal to *MaxWindows* as defined by InitToolsPlus.

The routine's value returns *true* if the window is open and visible, and *false* if the window is not open or not visible. The term "visible" refers to being programatically unhidden. It does not mean "obscured by other windows or objects."

WindowIsActive

Determine if a window is active.

C
```
pascal Boolean WindowIsActive (short Window);
```

Pascal
```
function WindowIsActive (Window: INTEGER): BOOLEAN;
```

Window is the window number of a Tools Plus window. Window must be less than or equal to *MaxWindows* as defined by InitToolsPlus.

The routine's value returns *true* if the window is open (not hidden) and active. Only the frontmost standard (not a tool bar or floating palette) window is active, and only when your application is active. The tool bar and floating palettes are always active when they are open and not hidden. The only exception to this is when a modal window is open, in which case the tool bar and floating palettes are temporarily inactive until the modal window is closed.

WindowKind

Determine a window's type.

C
```
pascal short WindowKind (short Window);
```

Pascal
```
function WindowKind (Window: INTEGER): INTEGER;
```

Window is the window number of a Tools Plus window. Window must be less than or equal to *MaxWindows* as defined by InitToolsPlus. The window may be hidden.

The routine returns with a value that corresponds to the type of window being referenced. The four constants that can be used to evaluate a window's type are *wNoKind* (window is not open), *wToolBarKind*, *wFloatingKind*, and *wStandardKind*.

```
CONST                           {Kinds of windows:                }
            wNoKind       = 0;  {Not open                         }
            wToolBarKind  = 1;  {Tool Bar                         }
            wFloatingKind = 2;  {Floating Palette                 }
            wStandardKind = 3;  {Standard Window                  }
```

GetFocusInfo

Determine the object with the keyboard focus in a window.

C
```
pascal void GetFocusInfo (short Window, short *ObjectNum, short *ObjectKind);
```

Pascal
```
procedure GetFocusInfo (Window: INTEGER; var ObjectNum: INTEGER;
            var ObjectKind: INTEGER);
```

Window is the window number of a Tools Plus window. Window must be less than or equal to *MaxWindows* as defined by InitToolsPlus. The window may be hidden.

ObjectNum returns with the number of the object that has the keyboard focus in the specified window, such as editing field number 22 or button number 5. If the specified window is not open or if it does not have an object with the keyboard focus, *ObjectNum* returns with a value of zero (0).

ObjectKind returns with a value that tell your application the kind of object that has the keyboard focus. If the specified window is not open or if it does not have an object with the keyboard focus, *ObjectKind* returns with a value of zero (0). The five constants that can be used to evaluate the kind of keyboard focus object are listed below.

```
CONST                                {Kinds of objects with keyboard focus:           }
            kNoFocusKind        = 0; {No object with keyboard focus                   }
            kButtonFocusKind    = 1; {Button (or item implemented as button)          }
            kScrollBarFocusKind = 2; {Scroll bar (or item implemented as scroll bar)  }
            kListBoxFocusKind   = 3; {List box                                        }
            kFieldFocusKind     = 4; {Editing field                                   }
```

WindowPointer

Get the pointer to a Tools Plus window.

C
```
pascal WindowPtr WindowPointer (short Window);
```

Pascal
```
function WindowPointer (Window: INTEGER): WindowPtr;
```

This routine returns a window pointer to a standard toolbox WindowRecord that is used by a Tools Plus window, regardless if that window is open or not.

Window is the window number of a Tools Plus window. Window must be less than or equal to *MaxWindows* as defined by InitToolsPlus. If it is not, nil is returned.

AutoMoveSize

Automatically move and/or resize subsequently created objects as their window's size changes.

C
```
pascal void AutoMoveSize (Boolean left Boolean top, Boolean right,
            Boolean bottom);
```

Pascal
```
procedure AutoMoveSize (left, top, right, bottom: BOOLEAN);
```

AutoMoveSize sets four global parameters that can optionally be adopted by objects as they are created. The four parameters indicate if a subsequently created object's *left*, *top*, *right* and/or *bottom* are automatically adjusted when their parent window's size changes. These settings optionally apply to objects created on any window.

left Does the object's left side track the window's right edge?
top Does the object's top track the window's bottom edge?
right Does the object's right side track the window's right edge?
bottom Does the object's bottom track the window's bottom edge?

As each object is created, an optional constant (like the button's bAutoMoveSize) is added to the object's spec to make it assume AutoMoveSize's settings. For example, AutoMoveSize lets you specify that the right and bottom edge of subsequently created objects are automatically resized without having to do so for each object.

If you prefer, you can specify the automatic resizing setting for an individual object instead of doing so for a group of subsequently created objects. For example, AutoMoveSizeButton lets you set these parameters for a specific button. Equivalent routines are available for all user interface elements that appear on windows.

☞ **Warning**: Make sure that you resize objects in a way that makes sense. Don't allow a window to shrink down to a size where objects become unusable or disappear altogether.

FinderDisplay

Hide or show the Finder and other applications.

C
```
pascal void FinderDisplay (Boolean Show);
```

Pascal
```
procedure FinderDisplay (Show: BOOLEAN);
```

Some applications, typically installers, hide the Finder (desk top) and other applications while they are doing their work. This effect shows the user an empty desk top, one that is void of all items including disk drives while the active application is doing its work. FinderDisplay performs this function but in doing so, provides only the desired visual effect. It does not prompt the user to quit other applications that may be running nor does it instruct those applications to quit. As a fail-safe precaution, the Finder and other applications are redisplayed when your application is suspended.

They return to their state set by your application when your application is activated.

Show indicates if the Finder and other applications are being hidden or displayed. The two constants that can be used for this flag are *on* and *off*.

☞ **Warning**: Some development environments may act up if you try stepping through your program while the Finder and other applications are hidden.

SetLiveWindowDragging

Turn the live window dragging/resizing option on or off.

C
```
pascal void SetLiveWindowDragging (Boolean LiveDrag);
```

Pascal
```
procedure SetLiveWindowDragging (LiveDrag: BOOLEAN);
```

This routine lets your application globally enable or disable the live window dragging/resizing option. The InitToolsPlus routine can optionally enable this feature unconditionally, or when your application is running on a specific processor. For additional flexibility, your application can turn this feature on or off using this routine based on its own criteria. You may consider letting your application's user set this option in a Preferences dialog.

LiveDrag specifies if the option is turned on or off. The constants *on* and *off* can be used for this purpose.

ReplaceWindowProcID

Replace a window type throughout the application. The use of one procID is replaced with another.

C
```
pascal void ReplaceWindowProcID (short OriginalProcID,
           short ReplacementProcID);
```

Pascal
```
procedure ReplaceWindowProcID (OriginalProcID, ReplacementProcID: INTEGER);
```

This routine lets your application globally replace the use of one procID with another. The replacement takes effect in windows that are opened after this routine is used.

OriginalProcID is the procID that is specified in your application's source code and in various resources such as 'DLOG' and 'WIND.'

ReplacementProcID is the procID that replaces OriginalProcID when the window is opened. When a window is opened in which the procID has a value that matches OriginalProcID, the procID is replaced with the value specified by ReplacementProcID.

As an example, you can program your application to make use of the "utility window" (floating palette) that is available when Mac OS 8's Appearance Manager is running. Early in your application following InitToolsPlus, your application can determine if the Appearance Manager is running by using the UsingAppearanceManager routine. If it is not, then your application can call ReplaceWindowProcID to replace the utility window's procID with a procID for a custom floating palette WDEF, such as the Infinity Windoid. This allows you to use the system's standard floating palette if it is available, otherwise you can use a custom floating palette.

ReplaceWindowProcID can be used to specify numerous window procID substitutions for your application. Tools Plus accumulates all the substitutions in a dynamic list and uses that list whenever a window is opened. You can remove an entry from the list by specifying a ReplacementProcID with the same value as OriginalProcID.

The Infinity Windoid

Tools Plus includes an efficient, versatile floating palette with the Tools Plus disk (in the “Optional Resources” folder). We also include a highly refined third-party floating palette window definition from Infinity Systems. The Infinity Windoid (WDEF) features a color drag bar and zoom box options, just like other commercial applications.

Water’s Edge Software is merely furnishing a third-party add-on for your benefit (at no cost), and we are in no way related to Infinity Systems. We can say, however, that Infinity has come up with a great looking palette WDEF! So good, that we feel it compliments Tools Plus.

Please read the related documentation for full details on warranty, copyright, and support questions. To contact the creators of the Infinity Windoid, please send all enquiries to:

Troy Gaul
Infinity Systems
19850 Portal Plaza
Cupertino, CA 95014
USA

America Online: TGaul
Internet: TGaul@aol.com

6 Buttons

Tools Plus supports the use of buttons on any Tools Plus window. Buttons are created on the current window by the NewButton routine. Each button is referenced by a unique button number, which can be from 1 to 511. This number is specified when the button is created, and refers to the specific button until that button is deleted. Note that the button number is related to its associated window. This means that two different windows can each have a button numbered "1" without interfering with each other. Whenever a button is clicked by the user, Tools Plus calls your event handler routine and reports the button number as well as its window number. You can also create a button from a 'CNTL' resource by using the LoadButton routine.

Buttons can be moved to a new location with MoveButton and have their width and/or height changed with SizeButton. MoveSizeButton combines both tasks by letting you specify new co-ordinates for the button.

When a button is no longer required, it is deleted by the DeleteButton routine, which releases the memory used by that button. This is done automatically if a window is closed. Buttons can be renamed by using the ButtonTitle routine, and hidden or displayed with the ButtonDisplay routine.

Tools Plus also supports the use of custom CDEFs as buttons, as well as the extended set of controls that are part of the Appearance Manager which first appeared in Mac OS 8. Many of these controls are implemented as buttons and are detailed in this section. See the chapter on Scroll Bars for details on the remaining Appearance Manager controls.

Button Types

All three standard Macintosh button types are supported by Tools Plus. The push-button is always used to "do something now," such as confirming or canceling a process. Check boxes and radio buttons are variations on a similar theme: they can be either selected or deselected by clicking on them. The check box contains an "x" when checked, whereas the radio button contains a dot. The difference between these two buttons is that radio buttons are logically grouped by your application such that only one button is selected within the group. When the user selects a radio button, your application de-selects the other buttons in the group. Radio buttons can be automatically deselected by being placed in a panel.

Push Button   Check Box   Radio Button

You can also use custom control (CDEF) resources in your application and Tools Plus will make them behave like a push-button, check box or radio button. Other controls that are available only in the Appearance Manager are details later in this section.

Button States

All three button types can be either enabled or disabled by using the EnableButton routine. When a button is disabled, it becomes dim and cannot be selected by the user. Check boxes and radio buttons can be either selected or de-selected by using the SelectButton routine.

When a window is inactive, all the associated buttons are automatically disabled and cannot be selected. When the window is activated, the buttons are automatically returned to their normal state as set by your application.

Push Button	Check Box	Radio Button
disabled	disabled	disabled
enabled & checked (no change)	enabled & checked	enabled & checked

Button Titles

A button's title can be changed by the ButtonTitle routine, however this should be done judiciously since this can be confusing to the user. A button's title is centered in a push button, and left aligned in a check box or radio button.

Fonts

All buttons default to using the Chicago 12pt font. When a button is created, it can optionally adopt and remember the window's current font, size and style settings (as set by the TextFont, TextSize, and TextFace routines) by including the bUseWFont option. The window's settings can then be changed without affecting the button. Unlike regular buttons, Tools Plus buttons can each have a different font. You can use the GetButtonFontSettings and SetButtonFontSettings routines to get and set the button's font, size and style settings.

Colors

By default, new buttons have a black frame and text, and a background that matches their parent window's backdrop color (which is white by default). Optionally, each button can adopt unique color settings as it is created. The colors for the various button parts are defined by the ButtonColors routine, and are optionally adopted by buttons as they are created. Buttons' colors can be changed afterwards using the SetButtonColors routine. Conversely, the GetButtonColors routine retrieves a button's color settings.

When designing applications, always design them in black and white then apply color (if required) to add value to your application. Don't add color just because you can. In the case of color buttons, test your color selection thoroughly on a monitor set to 8, 4, and 2-bit color and gray scale, and black and white to ensure that your colors and window backdrop color map to usable colors. Note that some controls ignore color settings.

Default Button

One push button on each window can be designated to be the "default" button by the SetDefaultButton routine. This routine draws an outline around the button as shown below. If the Return or Enter key is pressed, Tools Plus responds as if the default button had been clicked. The button's default status can be cleared by the NoDefaultButton routine. A default button cannot be created on a tool bar or floating palette.

Default

When the user is working in an editing field, only the Enter key invokes the default push button. This is done to avoid confusion between fields that can and cannot accept the Return key as a carriage return in the field.

Selecting Buttons and Command Keys

Normally, a button is selected when the user clicks on it. You can optionally make a button selectable by using a command key. When you add bCmdKey to the button's spec, the button can be selected by typing the command key in conjunction with the first character of the button's title. Additionally, ⌘-. (command-period) and the Escape key select a button whose title is "Cancel" (or a language-dependent equivalent).

Substituting Button ProcIDs

Certain system resources may or may not be available to your application depending on the system version of the Macintosh that is running your application. A good example of this is the 3D buttons that are part of the Appearance Manager in Mac OS 8 or later. With Tools Plus, you can design and write your application to use a custom button definition (CDEF resource) to provide 3D buttons in your application, such as those in SuperCDEFs. Then at the

beginning of your application it can determine the Mac's capabilities, specifically if the Appearance Manager is running to make the system's 3D buttons available to your application. If this is the case, your application can easily substitute the use of the custom 3D button CDEF with the Appearance Manager's 3D button throughout your application.

Two routines in the Miscellaneous Routines chapter of this manual help facilitate determining the capabilities of the Macintosh that is running your application: HasAppearanceManager and UsingAppearanceManager. You can also use the toolbox's Gestalt routines to determine whether other features are available or not. Tools Plus's ReplaceControlProcID is used to replace a specific button procID with another procID throughout your application, thereby substituting the use of one type of button with another.

Handling Buttons

Your application specifies if check boxes or radio buttons are selected or not. It also specifies if a button is enabled or disabled. When a window in inactive, Tools Plus disables all buttons on that window. When the window is activated again, all the buttons regain their correct status as specified by your application.

Tools Plus constantly inquires about any events that have occurred, including clicking on buttons. If a button is selected (i.e., the user presses the mouse button down and releases it within the button's region), Tools Plus reports it by calling your event handler routine. This also applies if the user presses the Enter or Return key when a window has a default button. In the case of check boxes or radio buttons, your application must then select or de-select the button appropriately.

If you place radio buttons in a panel, you can optionally have them behave as a radio button group so that when a button is selected, the other buttons in the group are automatically deselected. Otherwise Tools Plus doesn't know how your buttons are grouped and your application must select/deselect related buttons appropriately.

Warning: If you have obtained a handle to a button, do not change any of the fields in the button's record.

Appearance Manager Controls

The Appearance Manager, first introduced in mid 1997 with Mac OS 8, gives your application a number of 3D controls in addition to the ordinary push button, check box, radio button, and scroll bar that were originally supplied by Apple when Macintosh debuted in 1984. All the new Appearance Manager controls are implemented as CDEFs, but unlike third party CDEF resources that must be installed in your application when it is built, the Appearance Manager's controls are available to your application without having to install them. They are available from the system, just like regular system controls, if the Macintosh running your application has an Appearance Manager.

Your application can access the Appearance Manager's 3D push buttons, check boxes, radio buttons and scroll bars without any special programming. In fact, you can replace the standard controls throughout your application with the equivalent Appearance Manager controls as a default behavior when you initialize Tools Plus libraries with the InitToolsPlus routine. However, if you want to make use of other Appearance Manager controls and features, you need to make your application "Appearance Manager aware." 680x0 applications are automatically Appearance Manager aware. To make your PowerPC application Appearance Manager aware, see the Designing Your Application chapter of this manual for details in the "Using the Appearance Manager" section.

Many of the Appearance Manager's controls are considerably more complex than the standard controls, and understandably so because they offer considerably more features. Many controls place special significance on their initial values when they are created, specifically the control's minimum limit, maximum limit and current value (these items equate to the contrlMin, contrlMax and contrlValue fields of the Control Manager's ControlRecord record). Constants for these controls and all their options appear in the Appearance.h (C/C++ header) and Appearance.p (Pascal interface) files, as well as in Controls.h and Controls.p files.

See the chapters on Scroll Bars, Editing Fields, List Boxes and Pop-Up Menus in this user manual for additional Appearance Manager controls.

Note: For complete information on Appearance Manager concepts, the Appearance Manager's features, and how to best use the Appearance Manager's new controls, please read the documentation pertaining to the Appearance

Manager. It is available from Apple or in the latest issue of Inside Macintosh. This manual does not duplicate that material.

Push Button (CDEF 23)

This push button works identically to a standard pushButProc push button.

```
CONST
   kControlPushButtonProc       = 368;  {Push button ProcID                 }
   kControlPushButLeftIconProc  = 374;  {Push-button with left-side icon    }
   kControlPushButRightIconProc = 375;  {Push-button with right-side icon   }
```

When either of the icon push buttons are created, the control's maximum limit is used to specify the 'cicn' resource ID that is drawn in the push button.

Enabled

Pressed

With Icon

Check Box (CDEF 23)

This check box works similarly to a standard checkBoxProc check box, except that it also has a "mixed" mode in addition to being selected or unselected.

```
CONST
   kControlCheckBoxProc         = 369;  {Check box ProcID                   }
   kControlCheckBoxMixedValue   = 2;    {Button's value for "mixed" mode    }
```

Off

On

Mixed

Radio Button (CDEF 23)

This radio button works similarly to a standard radioButProc radio button, except that it also has a "mixed" mode in addition to being selected or unselected.

```
CONST
   kControlRadioButtonProc      = 370;  {Radio button ProcID                }
   kControlCheckBoxMixedValue   = 2;    {Button's value for "mixed" mode    }
```

Off

On

Mixed

Bevel Button (CDEF 2)

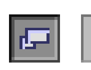
The bevel button is the most versatile control offered by the Appearance Manager. It allows you to specify the button's appearance, its content (picture, icon, etc.), and its behavior (push button, toggle, or sticky). See the Pop-Up Menus chapter for information about implementing the bevel button control as a pop-up menu.

All these capabilities are invoked by correctly setting the control's variant code, minimum limit, maximum limit, and value.

Parameter	Parameter's value is used for…
Variant Code	Bit 3 = Use window's font Bit 2 = Pop-up arrow's direction Bits 0-1 = Bevel size
Min Limit	High byte = Behavior Low byte = Type of content
Value	Always 0 (zero)
Max Limit	Resource ID for resource-based content types

If you use an icon suite, remember to include a mask icon (ICN# or ics#).

Small, Medium and Large Bevel Buttons



On

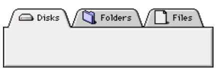
Off

```
CONST
                                           {Bevel Button ProcIDs:        }
   kControlBevelButtonSmallBevelProc  = 32;    {Small bevel                  }
   kControlBevelButtonNormalBevelProc = 33;    {Standard size bevel          }
   kControlBevelButtonLargeBevelProc  = 34;    {Large bevel                  }

                                           {Behaviors (in min. limit):   }
   kControlBehaviorPushbutton         = $0000; {Push button                  }
   kControlBehaviorToggles            = $0100; {Click on/off                 }
   kControlBehaviorSticky             = $0200; {Instant on                   }
   kControlBehaviorOffsetContents     = $8000; {Contents offset 1 pixel down }
                                           {  and right when clicked.    }

                                           {Content (in min. limit):     }
   kControlContentTextOnly     = 0;            {Button contains only text    }
   kControlContentIconSuiteRes = 1;            {Image us an icon suite       }
   kControlContentCIconRes     = 2;            {Image is a 'cicn' icon       }
   kControlContentPictRes      = 3;            {Image is a 'PICT'            }
   kControlContentIconRef      = 132;          {Image is an 'ICON'           }
```

Tabs (CDEF 8)



Tab Control

A tab control is a single control that has a number of parts (the tabs) that can be individually selected by the user. From a user's perspective, each tab part is used in a similar way to a radio button in a group: click one choice to select it and to deselect the others. The titles and icons for each of the tab parts can be set in two ways: by setting the control's title, or if you are creating the tab control using a 'CNTL' resource, by including a 'tab#' resource with the same resource ID as the 'CNTL' resource.

The title you provide for the control is broken into separate parts for each tab by using a vertical bar (the "|" key, or shift-\) between parts. If one or more numbers precede the title, then that number is used to specify the 'cicn' icon resource that appears in that tab part. The following example shows you a sample title for a tab control:

```
601Disks|602Folders|Files|605
```

In this example, four tab parts are created as follows:

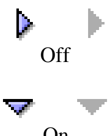
- #1 'cicn' icon ID 601, title = "Disks"
- #2 'cicn' icon ID 602, title = "Folders"
- #3 no icon, title = "Files"
- #4 'cicn' icon ID 605, no title

If you are creating a tab control using a 'CNTL' resource, you may find it easier to create a 'tab#' resource with the same ID as the 'CNTL' resource. The 'tab#' resource specifies the number of tab parts, each icon ID (0 = no icon) and title. If your resource editor does not support 'tab#' resources, check the "Optional Resources" folder in Tools Plus for a folder named "Optional Resource Templates". There, you will find a file named "Appearance Manager Templates" that contains the 'TMPL' resource that is needed to create a 'tab#' resource. Just copy this 'TMPL' resource into your resource editor application to give it the ability to create 'tab#' resources.

Note: You need Appearance Manager 1.0.1 or later to create tab controls in Tools Plus. Earlier versions have a bug that does not create the tab parts.

```
CONST
                                   {Tab ProcIDs:                      }
   kControlTabLargeProc = 128;         {Tab control with large tabs       }
   kControlTabSmallProc = 129;         {Tab control with small tabs       }
```

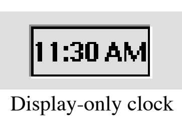
Disclosure Triangles (CDEF 4)


Off
On

Disclosure triangles work like check boxes: they can be *off* (related details are hidden, triangle points right or optionally to the left), or *on* (related details are displayed, triangle points down). You are responsible for coding the logic to hide and display the details relating to the triangle control, as this does not happen automatically. Disclosure triangles should always be created in a 12 x 12 pixel square.

```
CONST
                                               {Tab ProcIDs:            }
   kControlTriangleProc                  = 64;  {Triangle faces right    }
   kControlTriangleLeftFacingProc        = 65;  {Triangle faces left     }
   kControlTriangleAutoToggleProc        = 66;  {Faces right, auto-toggle}
   kControlTriangleLeftFacingAutoToggleProc = 67;  {Faces left, auto-toggle }
```

Clock (CDEF 15)



Display-only clock

Interactive clock

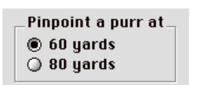
The clock control is used to display the time (and optionally the date) in a consistent manner, and to let the user set the time (and optionally the date). When created, this control defaults to displaying the current time (and optionally the date) as indicated by the Macintosh's internal clock. The clock control can be automatically updated once per minute or per second so that it always shows the current time. See the Appearance Manager's documentation for setting the time for this control and retrieving the setting.

The clock control updates automatically each time your event handler routine finishes executing. If you need to update the clock more frequently, see the Process1EventWhileBusy routine for details. When you create a clock control, it is best if you create it exactly 22 pixels high and use the system font.

```
CONST
                                    {Clock ProcIDs:                     }
   kControlClockTimeProc        = 240;  {Standard HH:MM time                }
   kControlClockTimeSecondsProc = 241;  {Time with seconds (HH:MM:SS)       }
   kControlClockDateProc        = 242;  {Date clock                         }
   kControlClockMonthYearProc   = 243;  {Date clock with month and year     }
                                    {Value settings for behavior:       }
   kControlClockNoFlags         = 0;    {User can change the time           }
   kControlClockIsDisplayOnly   = 1;    {User cannot change the time        }
   kControlClockIsLive          = 2;    {Auto-updated clock (combine this   }
                                    {  with kControlClockIsDisplayOnly.  }
```

Group Box (CDEF 10)

The group box control offers some of the visual cues that are found in Tools Plus's panels. Even though this control is not nearly as versatile as a Tools Plus Panel, you may decide to use it anyway because it presents a look that is consistent with the Appearance Manager. The user cannot interact with a group box control (clicking on it does not generate an event).

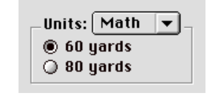


Primary titled group box

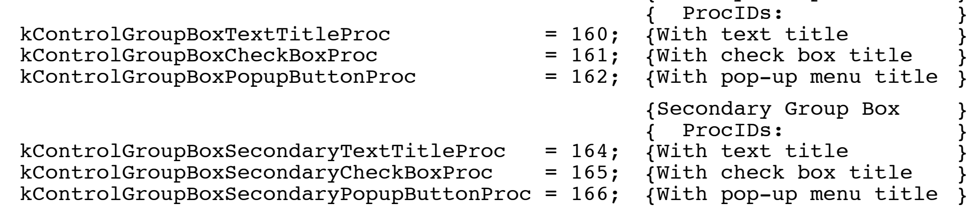
The Appearance Manager offers two additional services with the group box control:

- Setting the control's value to zero (0) deselects all radio buttons
- The control's value indicates the most recently selected radio button

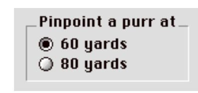
Group with check box



Group with pop-up menu



```
CONST
                                               {Primary Group Box       }
                                               {  ProcIDs:              }
   kControlGroupBoxTextTitleProc          = 160;  {With text title         }
   kControlGroupBoxCheckBoxProc           = 161;  {With check box title    }
   kControlGroupBoxPopupButtonProc        = 162;  {With pop-up menu title  }
                                               {Secondary Group Box     }
                                               {  ProcIDs:              }
   kControlGroupBoxSecondaryTextTitleProc   = 164;  {With text title         }
   kControlGroupBoxSecondaryCheckBoxProc    = 165;  {With check box title    }
   kControlGroupBoxSecondaryPopupButtonProc = 166;  {With pop-up menu title  }
```



Secondary Group Box

Chasing Arrows (CDEF 7)

Chasing Arrows

Chasing Arrows are used to indicate that a window, which is accessible to the user, is being updated by some process. This is seen in Mac OS 8's Finder when you open a folder that is set to icon view and that folder contains a lot of files -- while the Finder busies itself displaying the icons, the user sees the Chasing Arrows and can still interact with the window.

The Chasing Arrows control animates automatically each time your event handler routine finishes executing. If you need to animate the control more frequently, see the Process1EventWhileBusy routine for details. To stop animation, simply hide or delete this control. The user cannot interact with this control. Chasing Arrows should always be created in a 16 x 16 pixel square.

```
CONST
   kControlChasingArrowsProc = 112;     {Chasing Arrows ProcID              }
```

Little Arrows (CDEF 6)

Little Arrows

Little Arrows are used to increase or decreased a value, as seen in the Clock control. In Tools Plus, this control can be implemented either as a button to allow the user to step through a series of values one at a time with each click, or as a scroll bar to allow the user to also hold the up arrow or down arrow to continuously increase or decrease a value while the button is held down.

If you are using a 'CNTL' resource to create this control, add 1 to the procID to tell Tools Plus that you want to implement the Little Arrows control as a scroll bar, otherwise it is implemented as a button. Little Arrows should always be created in a rectangle that is 13 pixels wide by 23 pixels high.

```
CONST
   kControlLittleArrowsProc = 96;       {Little Arrows ProcID               }
```

Static Text (CDEF 18)

Static Text
Enabled

Static Text
Disabled

The Static Text control can be implemented as a non-selectable button or as a special kind of field called a static text field (see the Editing Fields chapter). You can use static text controls in place of standard static text items in dialogs. The advantage this provides is that the text looks disabled on an inactive window (it is dimmed) and you can easily manipulate the text as you would any other control, such as hiding and showing the control. The user cannot interact with this control.

```
CONST
   kControlStaticTextProc = 288;        {Static Text ProcID                 }
```

Placard (CDEF 14)

Placard

The placard is designed to be a frame or "context" for displaying small items, such as a page number to the left of a document's horizontal scroll bar. Avoid using a placard for a large background area, such as an area within a tab control that is used to hold several radio buttons. Using a placard in this manner may cause unsightly flickering because when the placard is refreshed, it overwrites the image of all embedded controls, then it refreshes those controls. The user cannot interact with this control.

```
CONST
   kControlPlacardProc = 224;           {Placard ProcID                     }
```

Visual Separator (CDEF 9)

The Visual Separator control is a dividing line between objects or groups of elements. Make sure you make this control 3 pixels wide. The user cannot interact with this control.

Visual Separator

```
CONST
   kControlSeparatorLineProc = 144;     {Visual Separator ProcID            }
```

Image Well (CDEF 11)

An image well is a display-only area where an image is shown. It provides an attractive recessed border and a consistent background for displaying pictures or icons. The control's Minimum Limit and Value fields are used to specify what is displayed in the control. The user cannot interact with this control.

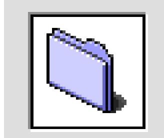


Image Well showing an Icon Suite

Parameter	Parameter's value is used for…
Min Limit	Type of content
Value	Resource ID for resource-based content types

If you use an icon suite, remember to include a mask icon (ICN# or ics#).

```
CONST
   kControlImageWellProc        = 176;          {Image Well ProcID            }

                                                {Content (in min. limit):     }
   kControlContentTextOnly      = 0;            {Button contains only text    }
   kControlContentIconSuiteRes = 1;             {Image us an icon suite       }
   kControlContentCIconRes      = 2;            {Image is a 'cicn' icon       }
   kControlContentPictRes       = 3;            {Image is a 'PICT'            }
   kControlContentIconRef       = 132;          {Image is an 'ICON'           }
```

Pop-Up Arrow (CDEF 12)

These minor controls are used typically in custom pop-up menus or in menu lists to indicate that more information can be shown. The user cannot interact with this control.

Pop-up Arrows

```
CONST
   kControlPopupArrowEastProc       = 192;  {Pop-up Arrow ProcIDs:             }
   kControlPopupArrowWestProc       = 193;  {                                  }
   kControlPopupArrowNorthProc      = 194;  {                                  }
   kControlPopupArrowSouthProc      = 195;  {                                  }
   kControlPopupArrowSmallEastProc  = 196;  {                                  }
   kControlPopupArrowSmallWestProc  = 197;  {                                  }
   kControlPopupArrowSmallNorthProc = 198;  {                                  }
   kControlPopupArrowSmallSouthProc = 199;  {                                  }
```

Picture Control (CDEF 19)

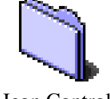
The Picture Control uses a 'PICT' resource to present the user with a click-sensing image. This control darkens the image when it is being tracked by the user, and dims the image when the control is disabled. Set the control's Value parameter to the 'PICT' resource ID that you want to display.

Picture Control

Warning: Flag the 'PICT' resource as preloaded, locked, and not purgeable to avoid an Appearance Manager bug.

```
CONST
                                       {Picture Control ProcIDs:          }
   kControlPictureProc        = 304;   {Standard, tracking picture control }
   kControlPictureNoTrackProc = 305;   {Instant-Event, non-tracking control }
```

Icon Control (CDEF 20)

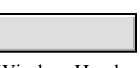

Icon Control

The Icon Control uses a ‘cicn’ resource or an icon suite to present the user with a click-sensing image. This control darkens the image when it is being tracked by the user, and dims the image when the control is disabled. Set the control’s Value parameter to the ‘cicn’ resource ID or to the icon suite ID that you want to display.

If you use an icon suite, remember to include a mask icon (ICN# or ics#).

```
CONST
                                 {Icon Control ProcIDs:              }
  kControlIconProc             = 320;  {Use a 'cicn', track control        }
  kControlIconNoTrackProc      = 321;  {Instant-Event, non-tracking control }
  kControlIconSuiteProc        = 322;  {Use an icon suite, track control    }
  kControlIconSuiteNoTrackProc = 323;  {Instant-Event, non-tracking control }
```

Window Header (CDEF 21)


Window Header

The window header control is similar to the placard control, except that it is used as a header area for a window. This control would typically contain column titles and perhaps a chasing arrows control. The user cannot interact with this control.

Note: You need Appearance Manager 1.0.2 or later to use window header controls in Tools Plus. Earlier versions have a bug that causes static text items placed on this control to display a pseudo random pattern instead of text.

```
CONST
                                       {Window Header ProcIDs:          }
  kControlWindowHeaderProc         = 336;  {Normal header                  }
  kControlWindowListViewHeaderProc = 337;  {List variant, no bottom line    }
```

User Pane (CDEF 16)

User Pane (invisible)

The User Pane control can be used in two very different ways. With knowledge of the Appearance Manager, you can write code that draws components of a custom pane in a style of your choosing to produce an interface element that is similar to a placard or to a window header control.

This control also provides a use in its naturally invisible state: you can create a user pane, then create a number of user interface elements on that pane and auto-embed them to the pane. This gives you the ability to hide or show all the user interface elements that belong to that pane just by hiding or showing the user pane control. This is a useful technique in hiding and showing “layers” of controls that are associated with a single tab part in a tab control. The user cannot interact directly with the user pane control.

```
CONST
  kControlUserPaneProc = 256;             {User Pane ProcID                 }
```

Appearance Manager and Keyboard Focus

Before the Appearance Manager's arrival, the only user interface element that could process keystrokes was an editing field. With the Appearance Manager present, a variety of user interface elements can process keystrokes, such as editing fields, list boxes and the clock control. Keystrokes are directed at only one user interface element at a time (and possibly at no element at all). When a user interface element processes keystrokes in such a way, it is said to have the "keyboard focus." Only one user interface element can have the keyboard focus at a time, and it is visually indicated with a highlighted "band" around the object. Tools Plus takes care of the focus highlight, and of applying keystrokes to the element that has the keyboard focus.

The process of moving the keyboard focus between objects, either by tabbing or clicking, is identical to that of navigating between editing fields. For details, see the "Clicking and Tabbing" section in the Editing Fields chapter.

NewButton

Create a new button.

C
```
pascal void NewButton (short Button, short left, short top, short right,
            short bottom, const Str255 Title, long Spec,
            Boolean EnabledFlag, Boolean SelectedFlag);
```

Pascal
```
procedure NewButton (Button, left, top, right, bottom: INTEGER;
            Title: STRING; Spec: LONGINT;
            EnabledFlag, SelectedFlag: BOOLEAN);
```

Button specifies the button number (from 1 to 511) that is created in the current window. Once a button is created, it is referenced by this button number. If a button has been previously created in the current window using the same number, it is replaced with a new button as specified by the parameters in the NewButton routine. If the current window doesn't belong to your application, or if no windows are open, NewButton does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in local co-ordinates that determines the button's size and location in the window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). See the chart below regarding the minimum height for buttons. These measurements are based on using the system font (Chicago 12pt) for the button's text.

	Minimum Height		Standard
	With Descenders	No Descenders	Height
Push Buttons	18	13	20
Radio/Check Box Buttons	14	12	16

The *Title* parameter is the button's title. Each button should have a unique title. Button titles can have multiple lines, each line being separated by the ASCII character code $0D (carriage return). It is your responsibility to ensure that the rectangle defining the button's co-ordinates is sufficient to contain the button's title.

Spec specifies a button's appearance and behavior. It is a combination of a button procID plus various Tools Plus options detailed later in this section.

EnabledFlag indicates if the newly created button is enabled or not. All three button types can be either enabled or disabled. When a button is disabled, it becomes dim and cannot be selected by the user. All buttons automatically become disabled when the window containing them is inactive. When the window is activated, the buttons assume their state as set by the NewButton routine and subsequent calls to the EnableButton routine. The two constants that can be used for this flag are *enabled* and *disabled.*

SelectedFlag indicates if the newly created button is selected or not. Only check boxes and radio buttons can be selected (this setting has no effect on push buttons). The two constants that can be used for this flag are *selected* and *notSelected.*

The following are examples of disabled and checked buttons:

Push Button	Check Box	Radio Button
pushButProc (disabled)	checkBoxProc (disabled, notSelected)	radioButProc (disabled, notSelected)
Push Button	Check Box	Radio Button
pushButProc (enabled)	checkBoxProc (enabled, selected)	radioButProc (enabled, selected)

Appearance and Behavior Specification

Spec specifies a button's appearance and behavior. It is a combination of a button procID (low 16 bits) plus various Tools Plus options (high 16 bits). The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result. For example, a push-button using the window's current font would have a spec of pushButProc + bUseWFont. The constants defining the available options are as follows:

Choose only one of the following procIDs (or use a custom CDEF's procID)…

`pushButProc` Standard Apple push button. Used to "do something now." [Push Button]

`checkBoxProc` Standard Apple check box. Used for "yes/no" types of selections. [Check Box]

`radioButProc` Standard Apple radio button. Used to select one of several options in a group where all options must be visible. This differs from the use of a pop-up menu where the only time all options need to be visible is when the user is making a selection, then only the selected item is displayed. [Radio Button]

Also see the section on Appearance Manager Controls earlier in this chapter for additional procIDs.

Optionally choose any of the following options…

`bUseWFont` Display the button using the window's current font, size and style settings (as set by the TextFont, TextSize, and TextFace routines). The button stores this information for future reference. By default, all buttons are drawn using the system font (Chicago, 12 pt).

`bColorButton` Adopt the color settings as defined by the ButtonColors routine. By default, buttons have black text and frame, and a background that matches their parent window's backdrop color (which is white by default). Note that some controls ignore color settings, particularly those in the Appearance Manager.

`bDefault` Make this button the window's default button. Can only be applied to one push button on a standard window. Pressing the "Return" or "Enter" key selects this button. [Default]

`bCmdKey` Allow the button to be selected by a command key (⌘-first letter). If the button's title is "Cancel" (or a language-dependent equivalent), ⌘-. (command-period) and the Escape key can be used to select the button.

`bAutoMoveSize` Automatically move and/or resize the button when the window's size changes. The AutoMoveSize routine lets you specify which sides are altered. You can use the AutoMoveSizeButton routine as an alternative to setting this option.

`bHidden` Create a hidden button. This kind of button is accessible to your application but not to the user. A default button loses its default status if you hide it.

Optionally choose only one of the following options if you are using non-standard custom CDEFs…

If you want to use a custom CDEF whose variant codes do not match up with those defined by Apple, you won't be able to use the pushButProc, checkBoxProc, radioButProc or bUseWFont constants because their values may map to completely different functions in the CDEF. Use the following options to inform Tools Plus of the CDEFs properties.

`bCDEFPushButton`	The CDEF is treated like a push button by Tools Plus. It can be the default for the window and its value always remains 1.
`bCDEFRadioButton`	The CDEF is treated like a radio button by Tools Plus. Its value can be 0 or 1 (unselected or selected). It can also be automatically deselected when placed in a panel as a radio button group.
`bCDEFCheckBox`	The CDEF is treated like a check box by Tools Plus. It can have any value.

Custom Control Definitions (CDEFs)

Your application can use custom control definitions (CDEFs) on a per-button basis. Tools Plus can make your custom button behave like a push button, radio button, or check box. When using a custom CDEF, you will need to include a special control definition (CDEF) resource in your application's resource fork. Tools Plus includes custom CDEFs in the "Optional Resources" folder.

You can write your own CDEFs or use those created by a third-party. A CDEF's procID is calculated as follows: CDEF's resource ID x 16 + variant code. As previously noted, if you are using a CDEF whose variant codes are different from those defined by Apple (pushButProc, checkBoxProc or radioButProc plus the optional bUseWFont), you can tell Tools Plus about the CDEF's properties by using the constants bCDEFPushButton, bCDEFRadioButton or bCDEFCheckBox. This lets you use the low 4 bits (variant code) as required by your CDEF.

Your CDEF's resource ID can be in the range of 2 to 2047. If you use 0 you will replace the use of Apple's standard buttons with your CDEF throughout your application. ID 1 is reserved by Apple's scroll bar and ID 63 is used by the pop-up menu CDEF in System 7 or higher. It is best to use resource IDs 128 or higher for your custom CDEFs.

✍ **Note**: When using third party CDEFs, make sure you carefully read the documentation that accompanies the CDEF. Your CDEF may not be able to make use of all the variant codes that are available to Apple's controls.

If your button is on a manually drawn background (other than a window's backdrop) such as a picture, that background must be refreshed in response to a doPreRefresh event. Tools Plus removes your button's rectangle from the update region when it generates the doRefresh event, thereby protecting it from being overwritten.

✍ **Note**: Tools Plus makes no attempt to control the placement of buttons or to protect them once they have been created. It is your responsibility to ensure that buttons are of sufficient size to contain their title, and that their placement within the window is reasonable and does not conflict with other objects. Furthermore, you should not allow your application's text and drawing processes to interfere with buttons, or with the "default button" frame. Windows with a "size box" should not allow buttons to be obscured or hidden by making the window too small.

Also see: SetAutoEmbed, NewButtonRect, NewDialogButton, ButtonColors and ReplaceControlProcID.

```
CONST                                       {Button appearance/behavior specifications:  }
               pushButProc   = 0;           {Push button                                 }
               checkBoxProc  = 1;           {Check box                                   }
               radioButProc  = 2;           {Radio button                                }
               bUseWFont     = $00000008;   {Use window's font settings                  }
               bDefault      = $00010000;   {Default push button (1 only per window)     }
               bCmdKey       = $00020000;   {Button is selectable via command key        }
               bColorButton  = $00080000;   {Color button                                }
               bHidden       = $00100000;   {Create hidden button                        }
               bAutoMoveSize = $00200000;   {Auto-resize as window's size changes        }
                                            {Button states:                              }
               enabled       = true;        {Enable button                               }
               disabled      = false;       {Disable button                              }
               selected      = true;        {Select (check) button                       }
               notSelected   = false;       {Deselect (un-check) button                  }
```

```
                                  {For custom (non-standard) CDEFs only:        }
bCDEFPushButton  = $80000000; {Control is a push button                    }
bCDEFRadioButton = $40000000; {Control is a radio button                   }
bCDEFCheckBox    = $20000000; {Control is a check box                      }
```

NewButtonRect

Create a new button.

C

```
pascal void NewButtonRect (short Button, const Rect *Bounds,
                const Str255 Title, long Spec, Boolean EnabledFlag,
                Boolean SelectedFlag);
```

Pascal

```
procedure NewButtonRect (Button: INTEGER; Bounds: RECT; Title: STRING;
                Spec: LONGINT; EnabledFlag, SelectedFlag: BOOLEAN);
```

NewButtonRect is identical to the NewButton routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

NewButtonControl

Create a new button.

C

```
pascal void NewButtonControl (short Button, short left, short top,
                short right, short bottom, const Str255 Title, long Spec,
                Boolean EnabledFlag, short ControlMinLimit, short ControlValue,
                short ControlMaxLimit);
```

Pascal

```
procedure NewButtonControl (Button, left, top, right, bottom: INTEGER;
                Title: STRING; Spec: LONGINT; EnabledFlag: BOOLEAN;
                ControlMinLimit, ControlValue, ControlMaxLimit: INTEGER);
```

NewButtonControl is identical to the NewButton routine, except that it allows you to specify the control's minimum limit, value, and maximum limit. You will only need to use this routine if you are using a custom CDEF with special requirements that necessitate setting these items to specific values.

NewButtonControlRect

Create a new button.

C

```
pascal void NewButtonControlRect (short Button, const Rect *Bounds,
                const Str255 Title, long Spec, Boolean EnabledFlag,
                short ControlMinLimit, short ControlValue, short ControlMaxLimit);
```

Pascal

```
procedure NewButtonControlRect (Button: INTEGER; Bounds: RECT;
                Title: STRING; Spec: LONGINT; EnabledFlag: BOOLEAN;
                ControlMinLimit, ControlValue, ControlMaxLimit: INTEGER);
```

NewButtonControlRect is identical to the NewButtonControl routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates. You will only need to use this routine if you are using a custom CDEF with special requirements that necessitate setting these items to specific values.

NewDialogButton

Create a new button in a dialog using a dialog item's co-ordinates.

C
```
pascal void NewDialogButton (short Button, const Str255 Title, long Spec,
            Boolean EnabledFlag, Boolean SelectedFlag);
```

Pascal
```
procedure NewDialogButton (Button: INTEGER; Title: STRING; Spec: LONGINT;
            EnabledFlag, SelectedFlag: BOOLEAN);
```

NewDialogButton is identical to the NewButton routine, except that the button is created in a dialog (a window opened with the LoadDialog routine, or one that had a dialog list attached with the LoadDialogList routine). The button's co-ordinates are obtained from the dialog item whose number matches the button number.

NewDialogButtonControl

Create a new button in a dialog using a dialog item's co-ordinates.

C
```
pascal void NewDialogButtonControl (short Button, const Str255 Title,
            long Spec, Boolean EnabledFlag,
            short ControlMinLimit, short ControlValue, short ControlMaxLimit);
```

Pascal
```
procedure NewDialogButtonControl (Button: INTEGER; Title: STRING;
            Spec: LONGINT; EnabledFlag: BOOLEAN;
            ControlMinLimit, ControlValue, ControlMaxLimit: INTEGER);
```

NewDialogButtonControl is identical to the NewDialogButton routine, except that it allows you to specify the control's minimum limit, value, and maximum limit. You will only need to use this routine if you are using a custom CDEF with special requirements that necessitate setting these items to specific values.

LoadButton

Create a new button using a 'CNTL' resource.

C
```
pascal void LoadButton (short Button, short ResID);
```

Pascal
```
procedure LoadButton (Button, ResID: INTEGER);
```

LoadButton creates a button by calling the NewButton routine and supplying it with values from a 'CNTL' resource, commonly called a control template. This is a good way to create a button or button-like control that requires a color table with more elements than those supported by the SetButtonColors routines. Note that some controls ignore color settings.

Button specifies the button number (from 1 to 511) that is created in the current window. Once a button is created, it is referenced by this button number. If a button has been previously created in the current window using the same number, it is replaced with a new button as specified by the parameters in the 'CNTL' resource. If the current window doesn't belong to your application, or if no windows are open, LoadButton does nothing.

ResID is the 'CNTL' resource ID number that is used to create the button. If the button has a 'cctb' color table resource, it must use the same ID number. Any resource ID number can be used, but numbers 128 or higher are safest as stated in Inside Macintosh.

When creating buttons using 'CNTL' resources, please note the following:

- Flag your 'CNTL' and 'cctb' resources as purgeable to save memory. Tools Plus makes a copy of their data.
- The RefCon field in the 'CNTL' resource is ignored since Tools Plus uses the control's RefCon field to store its own data.

Also see: NewButton and LoadSpecButton.

LoadSpecButton

Create a new button using a 'CNTL' resource.

C
```
pascal void LoadSpecButton (short Button, long Spec, short ResID);
```

Pascal
```
procedure LoadSpecButton (Button: INTEGER; Spec: LONGINT; ResID: INTEGER);
```

LoadSpecButton is identical to the LoadButton routine, except that it requires the additional *Spec* parameter to give you control over all the appearance and behavior options offered by Tools Plus. See the NewButton routine for details about the Spec parameter.

SetAutoEmbed

Automatically embed new controls (Appearance Manager only).

C
```
pascal void SetAutoEmbed (Boolean Embed);
```

Pascal
```
procedure SetAutoEmbed (Embed: BOOLEAN);
```

The Appearance Manager lets you embed a control into a parent control such that when the parent is hidden or disabled, all embedded controls are similarly affected. All Tools Plus routines that load a dialog item list (LoadDialog, LoadSpecDialog, LoadDialogList, etc.) automatically embed controls at all times. By default, when you create a control dynamically using a Tools Plus routine, that control is also embedded.

Embed indicates if subsequently created controls are automatically embedded, those controls being buttons and scroll bars, controls that are implemented as buttons and scroll bars (such as the Appearance Manager's Tab control or Progress Indicator control), Edit Text controls, Static Text controls, List Box controls, and Pop-Up Menu controls. This affects only controls that are dynamically created using Tools Plus routines. When *Embed* is set to true, each new control you create automatically calls the Appearance Manager's AutoEmbedControl routine.

You can safely call SetAutoEmbed even if the Appearance Manager is not available.

Also see: EmbedButtonInButton, EmbedButtonInScrollBar, EmbedFieldInButton, EmbedFieldInScrollBar, EmbedListBoxInButton, EmbedListBoxInScrollBar, EmbedScrollBarInButton, EmbedScrollBarInScrollBar, EmbedPopUpInButton, and EmbedPopUpInScrollBar.

EmbedButtonInButton

Embed a button into a button or into the window's root control (Appearance Manager only).

C
```
pascal void EmbedButtonInButton (short Button, short ContainerButton);
```

Pascal
```
procedure EmbedButtonInButton (Button, ContainerButton: INTEGER);
```

The Appearance Manager lets you embed a control into a parent control such that when the parent is hidden or disabled, all embedded controls are similarly affected. All Tools Plus routines that load a dialog item list (LoadDialog, LoadSpecDialog, LoadDialogList, etc.) automatically embed controls at all times. EmbedButtonInButton lets you manually embed a button into a button, or into the window's root control. Note that the term "button" does not literally mean a button control. It means any control that is implemented as a button in Tools Plus. The most likely candidate is a Group Box control. If the Appearance Manager is not available, EmbedButtonInButton does nothing.

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, EmbedButtonInButton does nothing.

ContainerButton specifies the button number (from 1 to 511) into which *Button* is embedded. This control must exist in the current window, and it must be a "container" type control such as the Appearance Manager's Group Box. The button must fit entirely within the container control or EmbedButtonInButton does nothing. If a value of 0 is provided for a container button, *Button* is embedded into the window's root control.

Also see: EmbedButtonInScrollBar and SetAutoEmbed.

EmbedButtonInScrollBar

Embed a button into a scroll bar or into the window's root control (Appearance Manager only).

C
```
pascal void EmbedButtonInScrollBar (short Button, short ContainerScrollBar);
```

Pascal
```
procedure EmbedButtonInScrollBar (Button, ContainerScrollBar: INTEGER);
```

The Appearance Manager lets you embed a control into a parent control such that when the parent is hidden or disabled, all embedded controls are similarly affected. All Tools Plus routines that load a dialog item list (LoadDialog, LoadSpecDialog, LoadDialogList, etc.) automatically embed controls at all times. EmbedButtonInScrollBar lets you manually embed a button into a scroll bar, or into the window's root control. Note that the term "button" does not literally mean a button control. It means any control that is implemented as a button in Tools Plus. The most likely candidate is a Group Box control. The same applies for the term "scroll bar." If the Appearance Manager is not available, EmbedButtonInScrollBar does nothing.

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, EmbedButtonInScrollBar does nothing.

ContainerScrollBar specifies the scroll bar number (from 1 to 511) into which *Button* is embedded. This control must exist in the current window, and it must be a "container" type control. The button must fit entirely within the container control or EmbedButtonInScrollBar does nothing. If a value of 0 is provided for a container scroll bar, *Button* is embedded into the window's root control.

Also see: EmbedButtonInButton and SetAutoEmbed.

GetFreeButtonNum

Get the first unused button number.

C
```
pascal short GetFreeButtonNum (void);
```

Pascal
```
function GetFreeButtonNum: INTEGER;
```

Some developers may prefer to write code that more closely resembles a traditional Macintosh application, in that creating an object returns a reference to it such as a handle or pointer. Instead of having to assign your own button number, GetFreeButtonNum returns the first unused (free) button number. Using this routine, you can assign an unused button number to a variable, then use that variable throughout your application without concern for the true button number.

GetFreeButtonNum returns the first free button number on the current window. If the current window doesn't belong to your application, if no windows are open, or if the maximum number of buttons has already been created on the current window (no new ones can be created), GetFreeButtonNum returns a value of zero (0).

ButtonColors

Set the colors for new buttons as they are created.

C
```
pascal void ButtonColors (const RGBColor *Frame, const RGBColor *Body,
            const RGBColor *Text, const RGBColor *Back);
```

Pascal
```
procedure ButtonColors (Frame, Body, Text, Back: RGBColor);
```

When new buttons are created, by default they have a black outline and text, and they adopt their parent window's backdrop as a background color. When you use the ButtonColors routine, new buttons adopt the colors specified in this routine (providing that the button is created with the bColorButton option in the button's spec). This is the most efficient way to color multiple buttons using the same colors. Note that some controls ignore color settings, particularly those in the Appearance Manager.

Frame is the button's frame color (seen in push buttons, check box's box, radio button's circle, and possibly custom CDEFs).

Body is the button's body color (seen in push buttons only, and possibly custom CDEFs).

Text is the button's text color (seen in all buttons, and usually in custom CDEFs).

Back is the button's background color (seen in check boxes and radio buttons, and possibly custom CDEFs).

Also see: NoButtonColors and SetButtonColors.

NoButtonColors

Reset the colors for new buttons to the default.

C
```
pascal void NoButtonColors (void);
```

Pascal
```
procedure NoButtonColors;
```

When new buttons are created, by default they have a black outline and text, and they adopt their parent window's backdrop as a background color. When you use the ButtonColors routine, new buttons adopt the colors specified by that routine (providing that the button is created with the bColorButton option in the button's spec).

This routine resets the settings of the ButtonColors routine to the default values (black frame and text, white body and background). It is seldom required since you can create default buttons by simply excluding the bColorButton constant from the button's spec parameter.

Also see: ButtonColors.

DeleteButton

Delete a button.

C
```
pascal void DeleteButton (short Button);
```

Pascal
```
procedure DeleteButton (Button: INTEGER);
```

Button specifies the button number (from 1 to 511) that is deleted from the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, DeleteButton does nothing. Use KillButton if you want to delete the button without removing its image from the window.

KillButton

Delete a button without affecting its image on the window.

C
```
pascal void KillButton (short Button);
```

Pascal
```
procedure KillButton (Button: INTEGER);
```

KillButton is identical to DeleteButton except that it does not remove the button's image from the window. This routine is useful for scrolling buttons in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetButton repositions the button's co-ordinates without affecting its image (since ScrollRect has already moved it). KillButton then deletes the buttons that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

ButtonDisplay

Hide or show a button.

C
```
pascal void ButtonDisplay (short Button, Boolean Show);
```

Pascal
```
procedure ButtonDisplay (Button: INTEGER; Show: BOOLEAN);
```

ButtonDisplay hides or shows a button on the current window. The result is seen immediately. Use discretion with this routine since buttons should be enabled and disabled to indicate if they are accessible by the user.

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, ButtonDisplay does nothing.

Show indicates if the button is being hidden or displayed. The two constants that can be used for this flag are *on* and *off*. A default button loses its default status if you hide it.

ButtonIsVisible

Determine if a button is visible.

C
```
pascal Boolean ButtonIsVisible (short Button);
```

Pascal
```
function ButtonIsVisible (Button: INTEGER): BOOLEAN;
```

ButtonIsVisible reports if a button (or a control that is implemented as a button) is visible on the current window, or if it is hidden.

Button specifies the button number (from 1 to 511) that is queried in the current window.

This routine's value returns *true* if the button is visible, and *false* if the button is hidden. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, ButtonIsVisible returns *false*. This routine takes control embedding into account, so it will return *false* if the target button is embedded and its container control is hidden.

ObscureButton

Hide a button without removing its image from the window.

C
```
pascal void ObscureButton (short Button);
```

Pascal
```
procedure ObscureButton (Button: INTEGER);
```

ObscureButton hides a button on the current window without removing its image from the window. This routine is useful for scrolling buttons in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetButton repositions the button's co-ordinates without affecting its image (since ScrollRect has already moved it). ObscureButton then hides the buttons that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

Button specifies the button number (from 1 to 511) that is hidden in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, ObscureButton does nothing.

ActivateButton

Activate a button to give it the keyboard focus.

C
```
pascal void ActivateButton (short Button, short PartCode);
```

Pascal
```
procedure ActivateButton (Button, PartCode: INTEGER);
```

Button specifies the button number (from 1 to 511) that acquires the keyboard focus in the current window. ActivateButton does nothing under any of these conditions: the current window doesn't belong to your application, no windows are open, the button does not exist in the current window, the button is disabled or hidden, the button cannot accept the keyboard focus, or the Appearance Manager is not available to your application.

PartCode is the control's part number that is being activated. The part number is available either in the Appearance Manager documentation, or from the author of the custom control you are using.

Activating a button allows the user to interact with the button by typing on the keyboard. On an active window, the button acquires the keyboard focus making it the item that automatically processes keystrokes. Visually, this is indicated by having the text highlighted or with a flashing caret. Additionally, the button is encompassed with a highlighting keyboard focus band to indicate that it has the focus. Using ActivateButton in an active window removes

the keyboard focus from any other object that may have the focus within the same window or any other active window such as a tool bar or floating palette. This action may deactivate an active editing field.

If the button being activated is in an active window that allows access to pull-down menus, the Edit menu's "Undo" item is changed to "Can't Undo" and is disabled. The "Cut", "Copy", "Paste", "Clear" and "Select All" items are also disabled.

Your application can activate virtually any editing field or Appearance Manager control that accepts the keyboard focus. This flexibility can lead to a confusing user interface by allowing the keyboard focus to jump between active windows. A good rule to observe is to activate a single item only on a standard window (not a tool bar or a floating palette) when the window first opens. This sets up the default keyboard focus item for that window. At all other times, activate a button only in response to a user's actions.

Also see: HaveTabInFocus, TabToFocus, the doClickToFocus event, and ClickToFocus for other activating services.

GetButtonRect

Get a button's co-ordinates.

C
```
pascal void GetButtonRect (short Button, Rect *Bounds);
```

Pascal
```
procedure GetButtonRect (Button: INTEGER; var Bounds: RECT);
```

Button specifies the button number (from 1 to 511) that is queried in the current window.

Bounds returns the button's bounding rectangle specified in the window's local co-ordinates. These co-ordinates match those used to create the button. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, Bounds returns with all co-ordinates set to zero (0).

EnableButton

Enable or disable a button.

C
```
pascal void EnableButton (short Button, Boolean EnabledFlag);
```

Pascal
```
procedure EnableButton (Button: INTEGER; EnabledFlag: BOOLEAN);
```

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, EnableButton does nothing.

The *EnabledFlag* indicates if the button is enabled or not. All three button types can be either enabled or disabled. When a button is disabled, it becomes dim and cannot be selected by the user. All buttons automatically become disabled when the window containing them is inactive. When the window is activated, the buttons assume their state as set by the NewButton routine, and subsequent calls to the EnableButton routine. The two constants that can be used for this flag are *enabled* and *disabled*.

```
CONST                              {Button state                     }
               enabled      = true;   {button is enabled                }
               disabled     = false;  {button is disabled               }
```

See the NewButton routine for additional information pertaining to the button's enabling, disabling, and selection (i.e., checked or not).

ButtonIsEnabled

Determine if a button is enabled or disabled.

C
```
pascal Boolean ButtonIsEnabled (short Button);
```

Pascal
```
function ButtonIsEnabled (Button: INTEGER): BOOLEAN;
```

Button specifies the button number (from 1 to 511) that is queried in the current window.

The routine's value returns *true* if the button is enabled, and *false* if the button is disabled. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, ButtonIsEnabled returns *false*. ButtonIsEnabled returns the button's enabled state as it is currently displayed, so if the button's window is inactive and has temporarily disabled the button, ButtonIsEnabled returns *false*.

SelectButton

Select or deselect (check or un-check) a button.

C
```
pascal void SelectButton (short Button, Boolean SelectedFlag);
```

Pascal
```
procedure SelectButton (Button: INTEGER; SelectedFlag: BOOLEAN);
```

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, SelectButton does nothing.

The *SelectedFlag* indicates if the button is selected (checked) or not. Only check boxes and radio buttons can be selected. This setting has no effect on push buttons. The two constants that can be used for this flag are *selected* and *notSelected*. If you are using a custom CDEF and you need to set the button to a specific value, use the SetButtonVal routine.

```
CONST                              {Button state                          }
            selected     = true;   {button is selected (checked)          }
            notSelected  = false;  {button is not selected (not checked)  }
```

See the NewButton routine for additional information pertaining to the button's enabling, disabling, and selection (i.e., checked or not).

ButtonIsSelected

Determine if a button is selected (i.e., checked.)

C
```
pascal Boolean ButtonIsSelected (short Button);
```

Pascal
```
function ButtonIsSelected (Button: INTEGER): BOOLEAN;
```

Button specifies the button number (from 1 to 511) that is queried in the current window.

The routine's value returns *true* if the button is selected (checked), and *false* if the button is not selected. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, ButtonIsSelected returns *false*.

See the NewButton routine for additional information pertaining to the button's enabling, disabling, and selection (i.e., checked or not).

GetButtonMin

Get a button's minimum value limit. This routine may be required for buttons that use a custom CDEF. Otherwise you will never need to use it.

C
```
pascal short GetButtonMin (short Button);
```

Pascal
```
function GetButtonMin (Button: INTEGER): INTEGER;
```

Button specifies the affected button number (from 1 to 511) in the current window.

GetButtonMin returns a button's minimum value limit. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, GetButtonMin returns a value of zero (0).

SetButtonMin

Set a button's minimum value limit. This routine may be required for buttons that use a custom CDEF. Otherwise you will never need to use it.

C
```
pascal void SetButtonMin (short Button, short minimum);
```

Pascal
```
procedure SetButtonMin (Button, minimum: INTEGER);
```

Button specifies the affected button number (from 1 to 511) in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, SetButtonMin does nothing.

Minimum specifies the button's new minimum value limit. The button's current value and maximum limit are automatically adjusted (if necessary) to be consistent with the new minimum limit.

GetButtonMax

Get a button's maximum value limit. This routine may be required for buttons that use a custom CDEF. Otherwise you will never need to use it.

C
```
pascal short GetButtonMax (short Button);
```

Pascal
```
function GetButtonMax (Button: INTEGER): INTEGER;
```

Button specifies the affected button number (from 1 to 511) in the current window.

GetButtonMax returns a button's maximum value limit. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, GetButtonMax returns a value of zero (0).

SetButtonMax

Set a button's maximum value limit. This routine may be required for buttons that use a custom CDEF. Otherwise you will never need to use it.

C
```
pascal void SetButtonMax (short Button, short maximum);
```

Pascal
```
procedure SetButtonMax (Button, maximum: INTEGER);
```

Button specifies the affected button number (from 1 to 511) in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, SetButtonMax does nothing.

Maximum specifies the button's new maximum value limit. The button's current value and minimum limit are automatically adjusted (if necessary) to be consistent with the new maximum limit.

GetButtonVal

Get a button's current value. This routine may be required for buttons that use a custom CDEF. Otherwise you will never need to use it.

C
```
pascal short GetButtonVal (short Button);
```

Pascal
```
function GetButtonVal (Button: INTEGER): INTEGER;
```

Button specifies the affected button number (from 1 to 511) in the current window.

GetButtonVal returns a button's current value. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, GetButtonVal returns a value of zero (0).

SetButtonVal

Set a button's current value. This routine may be required for buttons that use a custom CDEF. Otherwise you will never need to use it.

C
```
pascal void SetButtonVal (short Button, short value);
```

Pascal
```
procedure SetButtonVal (Button, Value: INTEGER);
```

Button specifies the affected button number (from 1 to 511) in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, SetButtonVal does nothing.

Value specifies the button's new current value. The value must fall within the limits defined by GetButtonMin and GetButtonMax or SetButtonVal does nothing.

ButtonTitle

Change a button's title.

C
```
pascal void ButtonTitle (short Button, const Str255 Title);
```

Pascal
```
procedure ButtonTitle (Button: INTEGER; Title: STRING);
```

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, ButtonTitle does nothing.

The *Title* parameter is the button's title. Each button should have a unique title. Button titles can have multiple lines, each line being separated by the ASCII character code $0D (carriage return). Note that a button's size does not change automatically to accommodate larger or smaller titles.

FlashButton

Flash a button as though it was clicked by the user.

C
```
pascal void FlashButton (short Button);
```

Pascal
```
procedure FlashButton (Button: INTEGER);
```

Button specifies the button number (from 1 to 511) that is affected in the active window. If the active window doesn't belong to your application, or if no windows are open, FlashButton does nothing.

FlashButton can be used in some specific instances. Advanced programmers may decide to display a modal window when the Macintosh is busy with a lengthy process. If a button (such as "Cancel") on this window is equivalent to typing ⌘-., your application should flash the button when a ⌘-. is reported to your event handler routine. This makes the user feel that the key triggered the button. Another example is double-clicking in a list box; this action can be interpreted as "select line and OK" in which case the OK button should be flashed. This also occurs if your application interprets double-clicking a radio button as "select button and OK."

MoveButton

Move a button to a new location on the window.

C
```
pascal void MoveButton (short Button, short toHoriz, short toVert);
```

Pascal
```
procedure MoveButton (Button, toHoriz, toVert: INTEGER);
```

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Button* specifies a button that does not exist, MoveButton does nothing. The change is seen immediately providing that the button is not hidden. The button's width and height are not changed.

ToHoriz is the new horizontal co-ordinate at which the left side of the button appears.

ToVert is the new vertical co-ordinate at which the top of the button appears.

Also see: SizeButton and MoveSizeButton.

OffsetButton

Change a button’s co-ordinates without affecting its image on the window.

C
```
pascal void OffsetButton (short Button, short distHoriz, short distVert);
```

Pascal
```
procedure OffsetButton (Button, distHoriz, distVert: INTEGER);
```

When you scroll an area that contains buttons, first use ScrollRect to scroll the pixel image containing the affected objects in the window. OffsetButton is used to offset a button’s co-ordinates without altering its image (since ScrollRect has already done so). At this point, the button’s co-ordinates match the scrolled image of the button. ObscureButton or KillButton can be used to hide or delete buttons that are scrolled out of view.

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn’t belong to your application, if no windows are open, or if *Button* specifies a button that does not exist, OffsetButton does nothing.

DistHoriz and *distVert* specify the horizontal and vertical amount by which the button’s co-ordinates are offset. Positive numbers are right and down. The button’s co-ordinates are updated but no change is seen.

SizeButton

Change a button’s size.

C
```
pascal void SizeButton (short Button, short width, short height);
```

Pascal
```
procedure SizeButton (Button, width, height: INTEGER);
```

SizeButton changes a button’s width and/or height without altering the button’s top or left co-ordinate. The change is seen immediately providing that the button is not hidden.

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn’t belong to your application, if no windows are open, or if *Button* specifies a button that does not exist, SizeButton does nothing.

Width and *height* specify the button’s new width and height in pixels. If either parameter is less than 1, SizeButton does nothing.

Also see: MoveButton and MoveSizeButton.

MoveSizeButton

Change a button’s co-ordinates.

C
```
pascal void MoveSizeButton (short Button,
            short left, short top, short right, short bottom);
```

Pascal
```
procedure MoveSizeButton (Button, left, top, right, bottom: INTEGER);
```

MoveSizeButton changes any of the button’s four co-ordinates. The change is seen immediately providing that the button is not hidden. This routine combines the functions of MoveButton and SizeButton.

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn’t belong to your application, if no windows are open, or if *Button* specifies a button that does not exist, MoveSizeButton does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in local co-ordinates that determines the button's size and location in the window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). If these parameters specify an empty rectangle, MoveSizeButton does nothing.

Also see: GetButtonRect.

MoveSizeButtonRect

Change a button's co-ordinates.

C
```
pascal void MoveSizeButtonRect (short Button, const Rect *Bounds);
```

Pascal
```
procedure MoveSizeButtonRect (Button: INTEGER; Bounds: RECT);
```

MoveSizeButtonRect is identical to the MoveSizeButton routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

AutoMoveSizeButton

Specify how a button is automatically moved and/or resized as its window's size is changed.

C
```
pascal void AutoMoveSizeButton (short Button,
            Boolean left, Boolean top, Boolean right, Boolean bottom);
```

Pascal
```
procedure AutoMoveSizeButton (Button: INTEGER;
            left, top, right, bottom: BOOLEAN);
```

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Button* specifies a button that does not exist, AutoMoveSizeButton does nothing.

The *left*, *top*, *right* and *bottom* parameters specify if that side of the button is automatically adjusted when the window's size changes. These setting are applied to the button and are used the next time the window's size changes:

left Does the button's left side track the window's right edge?
top Does the button's top track the window's bottom edge?
right Does the button's right side track the window's right edge?
bottom Does the button's bottom track the window's bottom edge?

You can think of each *false* value as locking that side of the button to a fixed co-ordinate regardless of the window's size (this is the default). Each *true* value establishes a fixed distance between that side of the button and the window's edge. For example, setting only *left* and *right* to *true* makes the button move horizontally as the window widens and narrows, but the button does not move vertically when the window's height changes.

If you are setting these values identically for a group of objects, use AutoMoveSize to define the settings then add the appropriate *x*AutoMoveSize constant (such as bAutoMoveSize for buttons) to the objects' spec as they are created. The objects will adopt the settings specified by the AutoMoveSize routine.

☞ **Warning**: Make sure that you resize objects in a way that makes sense. Don't allow a window to shrink down to a size where objects become unusable or disappear altogether.

SetButtonFontSettings

Set a button's font, size and style settings.

C
```
pascal void SetButtonFontSettings (short Button,
            short theFont, short theSize, Style theStyle);
```

Pascal
```
procedure SetButtonFontSettings (Button: INTEGER;
            theFont: INTEGER; theSize: INTEGER; theStyle: Style);
```

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if the button does not exist, SetButtonFontSettings does nothing. Otherwise, the change is seen immediately.

TheFont specifies the button's new font. The default is Chicago, which is represented by the systemFont constant.

TheSize specifies the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle specifies the button's new style. Special character constants defined by the Font Manager are bold, italic, underline and shadow. C programmers use the Font Manager's constants to specify a composite style, such as SetButtonFontSettings(1, geneva, 9, bold + outline) for bold and outlined, or SetButtonFontSettings(1, geneva, 9, 0) for plain text. Pascal programmers use the Font Manager's constants to specify a style set, such as SetButtonFontSettings(1, geneva, 9, [bold, outline]) for bold and outlined, or SetButtonFontSettings(1, geneva, 9, [ ]) for plain text.

A button's font settings are set when a button is created, so this routine is not normally used by many applications.

Note: This routine works on Appearance Manager savvy controls (ones that were written to take advantage of the Appearance Manager's extended features) that accept the "set font" command. This routine also works on classic controls (those that were not written to take advantage of the Appearance Manager, including Apple's controls in System 6 and System 7, and SuperCDEFs) as well as third party controls that observe two rules:

1. The high bit of the variant code (8) indicates that the control uses the window's font.
2. All parameters that are used to *create* the control, specifically the control's rectangle, title, visible state, initial value, minimum limit, maximum limit, and reference constant, all have no special significance.

You may experience issues with third-party CDEFs that place special significance on the initial settings that are used to create the control. For example, you may experience issues if you use a third-party icon-button CDEF that initially uses the "current value" setting to determine which icon it should display, then it later changes the control's "current value" setting to reflect if the button is selected or not. Your only solutions are: (1) create the control with the high bit of the variant code set on (+8 or bUseWFont), or (2) use another CDEF that does not place special significance on initial settings when the control is created, or (3) do not use the SetButtonFontSettings routine on that control.

GetButtonFontSettings

Get a button's font, size and style settings.

C
```
pascal void GetButtonFontSettings (short Button,
            short *theFont, short *theSize, Style *theStyle);
```

Pascal
```
procedure GetButtonFontSettings (Button: INTEGER;
            var theFont: INTEGER; var theSize: INTEGER; var theStyle: Style);
```

Button specifies the button number (from 1 to 511) in the current window whose font settings are being retrieved. If the current window doesn't belong to your application, if no windows are open, or if *Button* specifies a button that does not exist, GetButtonFontSettings returns default values.

TheFont is the button’s font number. The default is 0 which is represented by the systemFont constant.

TheSize is the font’s size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle is the field’s font style. The default is plain text, which is represented by 0 in C and [ ] in Pascal.

SetButtonColors

Set a button’s colors.

C
```
pascal void SetButtonColors (short Button, const RGBColor *Frame,
            const RGBColor *Body, const RGBColor *Text, const RGBColor *Back);
```

Pascal
```
procedure SetButtonColors (Button: INTEGER;
            Frame, Body, Text, Back: RGBColor);
```

Button specifies the button number (from 1 to 511) in the current window whose colors are being set. If the current window doesn’t belong to your application, or if no windows are open, SetButtonColors does nothing. Also, if *Button* specifies a button that does not exist, SetButtonColors does nothing. The change is seen immediately, regardless if the button was originally created with the bColorButton option or not. Note that some controls ignore color settings, particularly those in the Appearance Manager.

Frame is the button’s frame color (seen in push buttons, check box’s box, radio button’s circle, and possibly custom CDEFs).

Body is the button’s body color (seen in push buttons only, and possibly custom CDEFs).

Text is the button’s text color (seen in all buttons, and usually in custom CDEFs).

Back is the button’s background color (seen in check boxes and radio buttons, and possibly custom CDEFs).

Also see: ButtonColors and GetButtonColors.

GetButtonColors

Get a button’s colors.

C
```
pascal void GetButtonColors (short Button, RGBColor *Frame, RGBColor *Body,
            RGBColor *Text, RGBColor *Back);
```

Pascal
```
procedure GetButtonColors (Button: integer; var Frame: RGBColor;
            var Body: RGBColor; var Text: RGBColor; var Back: RGBColor);
```

Button specifies the button number (from 1 to 511) in the current window whose colors are being retrieved. If the current window doesn’t belong to your application, or if no windows are open, or if *Button* specifies a button that does not exist, GetButtonColors returns default color values.

Frame is the button’s frame color (seen in push buttons, check box’s box, radio button’s circle, and possibly custom CDEFs).

Body is the button’s body color (seen in push buttons only, and possibly custom CDEFs).

Text is the button’s text color (seen in all buttons, and usually in custom CDEFs).

Back is the button’s background color (seen in check boxes and radio buttons, and possibly custom CDEFs).

Also see: ButtonColors and SetButtonColors.

SetDefaultButton

Set a button to be a window's "default" button.

C
```
pascal void SetDefaultButton (short Button);
```

Pascal
```
procedure SetDefaultButton (Button: INTEGER);
```

Button specifies the button number (from 1 to 511) that will become the new default button in the current window. If the current window doesn't belong to your application, or if no windows are open, SetDefaultButton does nothing. Also, if *Button* specifies a button that is not a push button, or a button that does not exist, SetDefaultButton does nothing. A default button cannot be set on a tool bar or floating palette.

The default button is automatically selected if the user presses the "Return" key or "Enter" key. A black outline is automatically drawn around the default button when the window is active. If another default button already exists in the current window, it loses its "default" status. Note that only 1 button can be the default in each window.

NoDefaultButton

Remove "default button" status for a window.

C
```
pascal void NoDefaultButton (void);
```

Pascal
```
procedure NoDefaultButton;
```

This routine removes the "default button" status from the current window (i.e., a specific button will not be automatically selected when the user presses the "Return" key or "Enter" key). The black outline that is automatically drawn around the default button is removed. Buttons themselves, however, are not altered. If the current window doesn't belong to your application, or if no windows are open, NoDefaultButton does nothing.

GetButtonHandle

Get a handle to a button's control record.

C
```
pascal ControlHandle GetButtonHandle (short Button);
```

Pascal
```
function GetButtonHandle (Button: INTEGER): ControlHandle;
```

This routine returns a standard ControlHandle to a button that was created by a Tools Plus routine. You should never need to use this routine. It is provided for advanced programmers who may have specialized needs. Always use Tools Plus routines to create and manipulate buttons.

Button specifies the button number (from 1 to 511) in the current window whose handle is being retrieved. If the current window doesn't belong to your application, or if no windows are open, or if *Button* specifies a button that does not exist, GetButtonHandle returns nil.

☞ **Warning**: If you need to lock the handle or change its attributes, do so temporarily then restore the original settings before using any Tools Plus routines. If you alter this handle or any data that is made accessible by this handle, you do so at your own risk. The only exception is the control's reference constant (contrlRfCon field) which can safely be set using the toolbox's SetControlReference routine, and retrieved using the toolbox GetControlReference routine.

ReplaceControlProcID

Replace a button type throughout the application. The use of one procID is replaced with another.

C
```
pascal void ReplaceControlProcID (short OriginalProcID,
            short ReplacementProcID);
```

Pascal
```
procedure ReplaceControlProcID (OriginalProcID, ReplacementProcID: INTEGER);
```

This routine lets your application globally replace the use of one procID with another. The replacement takes effect in controls that are created after this routine is used.

OriginalProcID is the procID that is specified in your application's source code and in various resources such as dialogs, 'DITL' and 'CNTL'.

ReplacementProcID is the procID that replaces OriginalProcID when the button (or any other control type) is created. When a button is created in which the procID has a value that matches OriginalProcID, the procID is replaced with the value specified by ReplacementProcID.

As an example, you can program your application to use a custom 3D button CDEF for buttons, such as those found in SuperCDEFs. Early in your application following InitToolsPlus, your application can determine if the Appearance Manager is running by using the UsingAppearanceManager routine. If it is, then your application can call ReplaceControlProcID to replace the custom buttons' procIDs with standard Apple procIDs. This allows you to use the system's standard 3D buttons when they are available, otherwise you can use custom 3D buttons.

ReplaceControlProcID can be used to specify numerous button and scroll bar procID substitutions for your application. Tools Plus accumulates all the substitutions in a dynamic list and uses that list whenever a button or scroll bar is created. You can remove an entry from the list by specifying a ReplacementProcID with the same value as OriginalProcID.

7 Picture Buttons

Tools Plus supports the use of picture buttons on any Tools Plus window. Picture buttons allow an icon (of any type) or picture (PICT resource) to be used as a button. Picture buttons also allow the use of multiple images (icons or PICTs) to produce buttons whose appearance changes depending on whether the button is selected, disabled, or if its value changes. (Please note that within this chapter, the term *button* is used to refer to a picture button unless otherwise stated.)

The designing of attractive buttons is simplified with the use of Tools Plus's 3D picture buttons, which let you design a black and white (1-bit) image, then Tools Plus takes care of transforming it into an elevated 3D color button, much like the kind seen in the tool bars of many commercial software packages. When selected, these buttons appear to be pushed into the window with appropriate shading and highlighting.

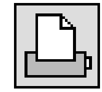
Picture buttons are created on the current window by the NewPictButton routine. Each picture button is referenced by a unique picture button number, which can be from 1 to 511. This number is specified when the picture button is created, and refers to the specific picture button until that button is deleted. Note that the button's number is related to its associated window. This means that two different windows can each have a picture button numbered "1" without interfering with each other. Whenever a picture button is clicked by the user, Tools Plus calls your event handler routine and reports the picture button number as well as its window number.

The support of color and multiple monitors is automatic, so Tools Plus's picture buttons will always be drawn using the most appropriate image, even if the button straddles the boundary of multiple monitors.

Picture buttons can be moved to a new location with MovePictButton. When a picture button is no longer required, it is deleted by the DeletePictButton routine, which releases the memory used by that button. This is done automatically if a window is closed. Picture buttons can be hidden or displayed with the PictButtonDisplay routine.

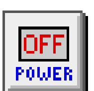
Button Types

Various types of picture buttons can be created, incorporating different behavioral characteristics and appearances. In the simplest implementation, a picture button can be used merely as a "click-sensitive" icon that reports an event when it is selected by the user. Picture buttons can also be used to functionally replace the standard push buttons, check boxes, or radio button groups. More importantly, picture buttons are more than a cosmetic user interface enhancement; in many cases they provide a stronger metaphor for tasks at hand (such as a "power on/power off" switch). The following are just a few examples of the types of picture buttons that can be created:



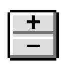
Click-Sensitive Icon

Push Button



Multistage Button (Click on, click off)

Radio Button Set (Exclusive Selection)



Polarized Button

Analog Simulation

Button Behavior

When you create a picture button, you must specify its behavioral characteristics. These characteristics define how the button operates when it is clicked by the user, as well as other properties the button has. Although the *behavior specification* is detailed by the NewPictButton routine, some of the choices you have are as follows:

- Does the button report an event when the mouse is first pressed down in the button, or does it wait for the mouse button to be released?
- Does the button lock in the "selected" position?
- Does the button produce repeated events when it is held down?
- How quickly does the button's value change?

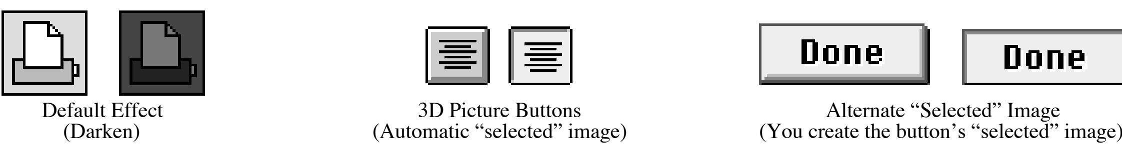
- Is the button polarized? (One side increases its value, the other decreases)
- Are PICTs used, or icons?

Please see the NewPictButton routine for details describing all the behavioral options at your disposal.

Selection Effects

An enabled button can be selected by the user by clicking on the button. Your application can also select or deselect any button by using the SelectPictButton routine. By default, Tools Plus darkens the button's image to make it appear selected, but you can override this effect individually for each button. Instead of darkening the image, you can provide an alternate image of the selected button. This is particularly effective if you are trying to produce the illusion of three dimensional controls. A complete description of selection effects is provided within the NewPictButton documentation.

Tools Plus's 3D picture buttons require only a single black and white icon to produce all the necessary three dimensional effects in color. They also provide you with several selection effects, all of which include the button being pushed into the window when it is selected.

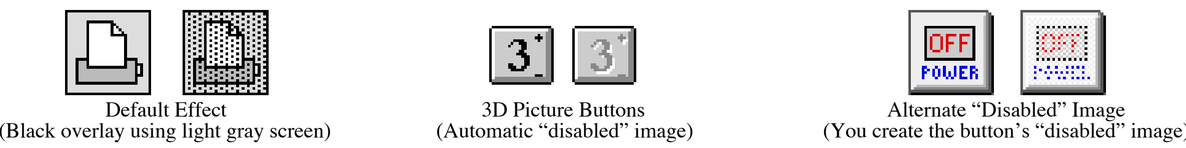


Default Effect (Darken) — 3D Picture Buttons (Automatic "selected" image) — Alternate "Selected" Image (You create the button's "selected" image)

Disabling Effects

All picture buttons can be enabled or disabled by using the EnablePictButton routine. When a button is disabled, it cannot be selected by the user. By default, Tools Plus overlays the image with a black color using a light gray pattern (25%) to make the button appear disabled (as does the Finder), but you can override this effect individually for each button. See the DefaultIconLook routine for details on setting disabled images' default appearance. Instead of applying the default disabling effect, you can provide an alternate image of the disabled button. This option provides you with the ultimate control over a button's appearance. A complete description of disabling effects is provided within the NewPictButton documentation.

Tools Plus's 3D picture buttons require only a single black and white icon to produce all the necessary three dimensional effects in color. When disabled, these buttons are automatically dimmed and "embossed" to provide a functional and attractive depiction of a disabled button.



Default Effect (Black overlay using light gray screen) — 3D Picture Buttons (Automatic "disabled" image) — Alternate "Disabled" Image (You create the button's "disabled" image)

Button's Value and Stages

Each picture button has an associated value, and range defined by an upper and lower limit of that range (a minimum value and maximum value). Buttons that are only concerned with being selected or disabled do not have to set or report button values. Other buttons, such as a picture button that controls the rotation of an object through 360°, need a *current value* that will fall within the button's *value range*. Your application can obtain the button's value at any time, or in response to an event.

Each button that has a range of values, even simple on/off buttons that have a small range (off = 0 and on = 1), can *optionally* be defined as *multistage* picture buttons. A multistage button has a different image for each stage. For example, a "Cutting Tool" button may have three stages, each of which has a different image for the different kinds of cutting tools available:

- Stage 1: Scissors image
- Stage 2: Knife image
- Stage 3: Ax image

In a multistage button, there is a “stage” for each value in the button’s range. Within each stage, the button potentially has four states: enabled/deselected, enabled/selected, disabled/deselected, and disabled/selected.

Multistage buttons should be used with discretion, since the buttons’ values are serial and the user cannot skip directly to a value. You can imagine the difficulty that a user would experience if they were forced to step through a dozen stages. Generally, three to five stages are the practical limits for a user interface. If you need more stages, consider using a more appropriate control, like a pop-up menu.

Single stage picture buttons can have any value that lies within their upper and lower limit. Multistage buttons use the button’s value to determine the current stage, therefore their value must also lie within their upper and lower limit.

Handling Picture Buttons

Your application specifies if picture buttons are selected or deselected, and if they are enabled or disabled. When a window in inactive, Tools Plus disables all of its picture buttons. When the window is activated again, all the buttons regain their correct status as specified by your application.

Tools Plus constantly inquires about any events that have occurred, including clicking on picture buttons. If a button is selected (i.e., the user presses the mouse button down and releases it within the button’s region), Tools Plus reports it by calling your event handler routine. If the button is configured to report an “instant” event, your application will be informed as soon as the mouse-down occurs in the picture button. Picture buttons that are configured to produce repeating events will produce events while the picture button is held down and the cursor is in the button.

When working with picture buttons that function like radio buttons, you can place these buttons in a panel and optionally have them behave as a radio button group so that when a picture button is selected, the other buttons in the group are automatically deselected. Otherwise Tools Plus doesn’t know how your picture buttons are grouped and your application must select/deselect related buttons appropriately.

NewPictButton

Create a new picture button.

C
```
pascal void NewPictButton (short Button, short left, short top,
            short BaseID, long Spec,
            Boolean EnabledFlag, Boolean SelectedFlag,
            short minimum, short value, short maximum);
```

Pascal
```
procedure NewPictButton (Button, left, top, BaseID: INTEGER;
            Spec: LONGINT; EnabledFlag, SelectedFlag: BOOLEAN;
            minimum, value, maximum: INTEGER);
```

Button specifies the picture button number (from 1 to 511) that is created in the current window. Once a picture button is created, it is referenced by this picture button number. If a picture button has been previously created in the current window using the same number, it is replaced with a new picture button as specified by the parameters in the NewPictButton routine. If the current window doesn’t belong to your application, or if no windows are open, NewPictButton does nothing.

Left and *top* define the top left-hand corner of the picture button in window’s local co-ordinates. The button’s images define the size of the button.

BaseID specifies the base resource ID number of the icon (any type) or PICT used by the picture button. All other image resources used by this button will be numbered higher than this one. See “Resource IDs” later in this section for a detailed description of image resource numbering.

Spec is the picture button's behavior and appearance specification. It is used by the button to determine its behavioral characteristics, and how it looks when selected or deselected, enabled or disabled, and when the button's value is changed. See "Behavior and Appearance Specification" later in this section for a detailed description of how the value for this item is determined.

The *EnabledFlag* indicates if the newly created picture button is enabled or not. When a picture button is disabled, it cannot be selected by the user. All picture buttons automatically become disabled when the window containing them is inactive. When the window is activated, the picture buttons assume their state as set by the NewPictButton routine and subsequent calls to the EnablePictButton routine. The two constants that can be used for this flag are *enabled* and *disabled*.

The *SelectedFlag* indicates if the newly created picture button is selected or not. The two constants that can be used for this flag are *selected* and *notSelected*.

Minimum declares the picture button's minimum value limit. In multistage buttons, the first stage starts at this minimum limit. Use zero (0) if your button's value does not change.

Value defines the picture button's current value. The current value must be greater than or equal to the minimum limit, and less than or equal to the maximum limit. Use zero (0) if your button's value does not change.

Maximum declares the picture button's maximum value limit. The maximum limit must be greater than the minimum limit. In multistage buttons, use this upper limit to define the total number of available stages (i.e., Maximum = Minimum + Total Stages - 1). Use zero (0) if your button's value does not change.

Note: Tools Plus makes no attempt to control the placement of picture buttons or to protect them once they have been created. It is your responsibility to ensure that picture buttons are placed within the window as to not conflict with other objects. Furthermore, you should not allow your application's text and drawing processes to interfere with picture buttons. Windows with a "size box" should not allow picture buttons to be obscured or hidden by making the window too small.

Warning: If you are using a 'cicn' (variable size color) icon that may be displayed on a Macintosh that doesn't have Color QuickDraw, make sure the icon's size is set to at least 9 pixels wide (although the actual image can be smaller). A bug in the Macintosh's ROMs causes a crash when CopyBits tries to work on a BitMap that is 8 pixels wide or less. Tools Plus circumvents this bug by not displaying the 'cicn'.

Resource IDs

Picture buttons can use PICTs or icons of any kind for a button's image. Multiple buttons can also share the same resources. When you are designing images for your buttons, make sure you adhere to the resource numbering schemes detailed in this section. The numbering system is based on "stages" with each stage having a block of resource IDs. The relative resource ID in each stage (i.e., the third resource in each stage) performs the same routine, the only difference being the stage number (which correlates to the button's value). For example, the first image in each stage is the button's image as an enabled, deselected button. The first image in stage 1 is for an enabled, deselected button with the button's *minimum* value. The first image in stage 2 is for an enabled, deselected button with the button's *minimum* value + 1.

Picture buttons can also incorporate a mask, which is useful if the button uses images that are not rectangular, or if it uses a set of images that vary in size. If a mask is provided, the button's image is limited to the mask's region, as will the user's mouse clicks. See the DrawIcon routine for a detailed description of how a mask works.

Icon Resource IDs

If you are going to use icons for picture button images, you must first have an understanding of an *icon family*. An icon family is a set of icons (of any type) that share the same resource ID number. From a picture button's point of view, all the images in an icon family are all the same image with the only difference being the suitability of a particular icon for the target monitor's settings.

Tools Plus's picture buttons will always select the best possible image for the button, depending on the monitor's settings. This is true even if the button straddles multiple monitors. See the DrawIcon routine for a detailed description of each icon, and the sequence in which they are accessed by Tools Plus.

The following chart describes the icon resource numbering sequence that must be followed when creating images for picture buttons:

	Icon ID	Selected	Enabled
Stage #1 (min. limit)	Base ID	N	Y
	Base ID + 1	Y	Y
	Base ID + 2	N	N
	Base ID + 3	Y	N
Stage #2	Base ID + 4	N	Y

Note: Multistage 3D buttons using the SICN resource require only 1 resource for all possible combinations and stages

When you are creating icons for your picture button, be aware that you do not have to create an icon for each possible combination of being selected/deselected and enabled/disabled. If your button uses an alternate image as a selection effect, you will need to create an image for the selected button (Base ID + 1). If your button uses an alternate image as a disabling effect, you will have to create an image for the disabled button (Base ID + 2). And if your button uses an alternate image for both selecting *and* disabling, you will have to create an image for the selected, disabled button (Base ID + 3). If you are creating a multistage button, you will have to create similar images for each stage (note that your button can have a range of values and still be a single stage button).

In the table above, the "Stage" represents multiple stages in a multi-stage button. Each stage corresponds to a single button value. For example, the first stage (Stage #1) represents the button's minimum limit. The second stage represents the button's minimum limit plus one. Using this pattern, if we have an icon Base ID of 128, then the selected and enabled icon for Stage #1 has a resource ID of 129 (Base ID of 128 +1), and the same button for Stage #2 has a resource ID of 133 (Base ID of 128 + 5).

3D SICN Buttons

A unique feature is available in Tools Plus's picture buttons that allows you to create 3D color buttons by simply designing a black and white icon. This is accomplished by using an SICN resource. A single SICN resource is capable of storing multiple icon images, and is therefore particularly well suited for multistage buttons.

If you are using an SICN resource to create a 3D button, you only need to create *one* resource for the button. Tools Plus formulates all the necessary selected and disabled images. If the button has multiple stages, create one icon within the that SICN resource for each stage.

Note: To avoid icon and picture conflicts while you are developing your application, avoid resource numbers that are used by your development environment (THINK C or THINK Pascal). THINK C and THINK Pascal sometimes supply their *own* resources in place of those in your resource file whenever resources numbers coincide. You can create and edit resources with a resource editor such as Apple's ResEdit. Remember to use ID numbers 128 or higher. The rest are reserved numbers.

PICT Resource IDs

The numbering scheme used for PICTs is similar to the one used for icons, except that it is not possible to create several PICTs (to account for different monitor settings) with the same resource ID number. Therefore, a button's PICTs are numbered in such a way as to preserve the trend established by the icon's numbering scheme, and to account for multiple PICTs that are required for different screen depths.

Tools Plus's picture buttons will always select the best possible PICT image for the button, depending on the monitor's settings. This is true even if the button straddles multiple monitors.

The following chart describes the PICT resource numbering sequence that must be followed when creating images for picture buttons:

	Scr Depth	PICT ID	Selected	Enabled
Stage #1 (min. limit)	B&W 4-bit 8-bit Mask	Base ID Base ID + 1 Base ID + 2 Base ID + 3	N	Y
	B&W 4-bit 8-bit Mask	Base ID + 4 Base ID + 5 Base ID + 6 Base ID + 7	Y	Y
	B&W 4-bit 8-bit Mask	Base ID + 8 Base ID + 9 Base ID + 10 Base ID + 11	N	N
	B&W 4-bit 8-bit Mask	Base ID + 12 Base ID + 13 Base ID + 14 Base ID + 15	Y	N
Stage #2	B&W	Base ID + 16	N	Y

Note that in the table above, the pixel depth of your picture does not need to match that of the screen depth in the second column. You can, for example, use 16-bit, 24-bit, or even 32-bit pictures for your picture buttons.

When you are creating PICTs for your button, be aware that you do not have to create a PICT for each possible combination of being selected/deselected and enabled/disabled, or color. If your button uses an alternate image as a selection effect, you will need to create an image for the selected button (Base ID + 4 through 7). If your button uses an alternate image as a disabling effect, you will have to create an image for the disabled button (Base ID + 8 through 11). And if your button uses an alternate image for both selecting *and* disabling, you will have to create an image for the selected, disabled button (Base ID + 12 through 15). If you are creating a multistage button, you will have to create similar images for each stage (note that your button can have a range of values and still be a single stage button).

Always create a PICT for the black and white image. Depending on your requirements, you may choose not to have color PICTs for 4-bit or 8-bit monitor settings.

The mask is optional in all cases, but is recommended in cases when the button is irregularly shaped, or if all the PICTs used by a button are not the same size. Tools Plus also recognizes that in many cases, masks will be identical between stages, and possibly between selected/deselected and enabled/disabled buttons. A picture button will try to find a mask that corresponds to its current selection and enabled state. If the appropriate mask can't be found (because the correctly number PICT resource does not exist), it will try to use substitute masks based on the following rules:

- If the button is disabled, try to find the equivalent "enabled" mask in the same stage
- If the button is selected, try to find the equivalent "deselected" mask in the same stage

If a multistage button is not at its minimum stage…

- If the button is disabled, try to find the equivalent "enabled" mask in the minimum stage
- If the button is selected, try to find the equivalent "deselected" mask in the minimum stage

Behavior and Appearance Specification

Spec specifies the picture button's appearance and behavior characteristics. It is used by the button to determine its behavioral characteristics, and how it looks when selected or deselected, enabled or disabled, and when the button's value is changed. The value for this 4-byte long integer can be specified either by adding a set of constants to obtain the desired result, or using a specially defined variant record, as illustrated below:

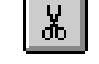
Optionally choose any of the following options…

`picbutMultiStage`	The button has a different image for each value in a range of values (i.e., 0 through 3). Note that even with this option turned off, you can still use an alternate image for a selected button and/or a disabled button.

`picbutAutoMoveSize`	Automatically move and size the picture button when the window's size changes. The AutoMoveSize routine lets you specify which sides are altered. You can use the AutoMoveSizePictButton routine as an alternative to setting this option.
`picbutHidden`	Create a hidden button. This kind of button is accessible to your application but not to the user.

Optionally choose only one of the following "image type" options…

`picbutUsePICTS`	Use PICTs instead of icons for the button's image(s). By default, a suite of icons is used for the button's image. Note that PICTs and icons use different resource numbering schemes. See the relevant details earlier in this chapter.
`picbutGray4use8`	If your 8-bit PICTs look good on 4-bit gray scale monitors, turn this option on to allow them to be used in such a way. By default, a 4-bit PICT is required when a monitor is set to 4-bits. This option can only be used in conjunction with the picbutUsePICTS (use PICT resources) option.
`picbutBigSICN3D`	If you are using an SICN icon to produce a 3D button, you can create a slightly larger button (24 x 22 pixels) with more pronounced shading. By default, a 3D SICN icon produces a slightly smaller button (24 x 20 pixels).


Big SICN 3D

Standard SICN 3D

Optionally choose only one of the following tracking options…

`picbutInstantEvent`	Report a picture button event as soon as the mouse-down occurs in the button. By default, the picture button generates an event when the mouse button is released. This option is best utilized with "click sensitive" icons, such as the ones seen in the Chooser. Instant events are automatically turned on when you turn on the picbutRepeatEvents (repeating events) option.
`picbutTrackWithHilite`	Draw a tracking highlight (a bold outline around the button, similar to the one used by the toolbox's radio buttons) when the mouse button is down and the cursor is inside the button.

Optionally choose only one of the following mouse-down options…

`picbutLockSelected`	When the user selects the button, lock it in the selected state thereby preventing the user from deselecting it. This option is usually used to produce the functionality of radio buttons, where a button can be turned on by clicking on it, but the user has to click another button to turn this one off.
`picbutSwitchSelected`	When the user clicks the button, reverse the "selection" state (i.e., if currently selected, switch to deselected; if currently deselected, switch to selected). This option produces a simple click-on/click-off type of button using a single stage.
`picbutRepeatEvents`	Repeated doPictButton events are generated as long as the mouse button is down and the cursor is in the button. These events can optionally increment/decrement the button's value at a specified rate. This option is useful for a button that controls object movement or situations where the button's value may change by more than one stage.

Optionally choose any of the following value changing options…

`picbutAutoValueChg`	Automatically increment/decrement the button's value when it is selected by the user. By default, your application must change the button's value as required. Automatic value changing is only useful if your button has a *range* of values through which is can progress.
`picbutValueWrap`	When the button's value reaches either the high or low limit, start at the opposite end of the range. By default, the button's value stops changing when it reaches the minimum or maximum limit. This option can only be used in conjunction with the picbutAutoValueChg (automatic value change) option.

Optionally choose only one of the following "rate of value change" options if the picbutAutoValueChg option is on…

`picbutScaleLinear`	After an initial pause, the button's value increments/decrements at a fixed rate. This is the default speed for automatic value changes and does not need to be explicitly stated.
`picbutScaleSlowAccel`	Same as above, but the rate slowly accelerates.
`picbutScaleMedAccel`	Same as above, but the rate accelerates at a moderate rate.
`picbutScaleFastAccel`	Same as above, but the rate accelerates rapidly.

Choose only one of the following "button splitting" options if required…

`picbutLeftRightSplit`	Clicking on the left half of the button decrements the button's value, while clicking on the right half increments it. By default, clicking anywhere on the button increments the value. Note that if the picbutAutoValueChg (automatic value change) option is off, the button's value is not automatically changed: an event is generated telling your application to change the value.
`picbutTopBottomSplit`	Same as above except that clicking on the bottom half of the button decrements the button's value, while clicking on the top half increments it.

Choose only one of the following "selection effects" options…

`picbutSelectDarken`	Darken the button's image when it is selected.
`picbutSelectDarkenSICN3D`	Darken and "push in" a 3D button when it is selected. An SICN resource is used for the button's image. Tools Plus converts the SICN to a 3D color button and formulates all the necessary selected and disabled images.
`picbutSelectLightenSICN3D`	Lighten and "push in" a 3D button when it is selected. An SICN resource is used for the button's image. Tools Plus converts the SICN to a 3D color button and formulates all the necessary selected and disabled images.
`picbutSelectPushedSICN3D`	"Push in" a 3D button when it is selected, but don't darken or lighten the button. An SICN resource is used for the button's image. Tools Plus converts the SICN to a 3D color button and formulates all the necessary selected and disabled images. This option is suitable for momentary push buttons (like the Macintosh's standard push button), because they provide minimal visual feedback that the button is selected.
`picbutSelectAltImage`	Use an alternate image when the button is selected. A selected image is required for each stage if the picbutMultiStage (multiple stage) option is on.

Choose only one of the following “disabling effects” options.
If a disabling effect is not specified, the global default is used as defined by the DefaultIconLook routine…

Constant	Description			
`picbutDimUsingBlackLite`	When the button is disabled, overlay the image with a black color using a “light gray” (25%) pattern.	Enabled (not selected)	Disabled (selected)	Disabled (not selected)
`picbutDimUsingWhiteLite`	When the button is disabled, overlay the image with a white color using a “light gray” (25%) pattern.	Enabled (not selected)	Disabled (selected)	Disabled (not selected)
`picbutDimUsingWhite`	When the button is disabled, overlay the image with a white color using a “medium gray” (50%) pattern.	Enabled (not selected)	Disabled (selected)	Disabled (not selected)
`picbutDimLeaveBorder`	When the button is disabled, do not apply the disabling effect to the image’s border. This option can be used in conjunction with any of the disabling effects listed above.	Enabled	Disabled	Disabled border preserved
`picbutDimAltImage`	Use an alternate image when the button is disabled. If the button is multistage, a selected image will likely be required for each stage. You may have to create a disabled image for both the selected and deselected state if the button can be seen in such a way.			
`picbutDimNoChange`	The button appears unchanged when it is disabled. The user is beeped if they click on a disabled button.			

So, if you want to create a large SICN 3D picture button that locks into the selected state, lightens when selected, and is disabled by overlaying the image using a white color with a 50% gray pattern, you should use the combined constants picbutLockSelected + picbutBigSICN3D + picbutSelectLightenSICN3D + picbutDimUsingWhite. Alternatively, a C structure and a Pascal variant record are available to help you define the spec parameter in a more intuitive way, as follows:

C

```
union TPPictButtonSpec {                    /*Picture Button's appearance and            */
                                            /*  behavior specs in 2 formats…             */
   struct{                                  /* · Parsed into components:                 */
      unsigned short InstantEvent: 1;       /*     Report event on mouse-down            */
      unsigned short TrackWithHilite: 1;    /*     Track w/highlight (like radio button) */
      unsigned short LockSelected: 1;       /*     Lock if selected (mouse can't deselect) */
      unsigned short SwitchSelected: 1;     /*     Switch 'select' state if clicked      */
      unsigned short RepeatEvents: 1;       /*     Repeat event when button is held      */
      unsigned short AutoValueChg: 1;       /*     Automatically change button's value   */
      unsigned short AutoValueScaling: 3;   /*     Rate of change for button's value     */
      unsigned short ValueWrap: 1;          /*     Button's range of values 'wrap' around */
      unsigned short LeftRightSplit: 1;     /*     Left side reduces value, right increase */
      unsigned short TopBottomSplit: 1;     /*     Top increases value, bottom reduces   */
      unsigned short MultiStage: 1;         /*     Button has multiple stages            */
      unsigned short BigSICN3D: 1;          /*     Create a larger SICN 3D button        */
      unsigned short UsePICTS: 1;           /*     Use PICTs instead of icons            */
      unsigned short Gray4use8: 1;          /*     Use 8-bit color pict on 4-bit gray    */
                                            /*       scale monitor.                      */
                                            /*   Selection Effects…                      */
      unsigned short SelectDarken: 1;       /*     Darken image                          */
      unsigned short SelectDarkenSICN3D: 1; /*     Darken (+push in) a 3D SICN icon      */
      unsigned short SelectLightenSICN3D: 1; /*    Lighten (+push in) 3D SICN icon       */
      unsigned short SelectPushedSICN3D: 1; /*     Same color (+push) 3D SICN icon       */
      unsigned short bit12: 1;              /*      (reserved bit)                       */
      unsigned short bit11: 1;              /*      (reserved bit)                       */
      unsigned short SelectAltImage: 1;     /*     Use an alternate image                */
      unsigned short Hidden: 1;             /*   Create a hidden button                  */
                                            /*   Disabling Effects…                      */
      unsigned short DimUsingBlackLite: 1;  /*     Overlay Black color, Lt Gray pat      */
      unsigned short DimUsingWhiteLite: 1;  /*     Overlay White color, Lt Gray pat      */
      unsigned short DimUsingWhite: 1;      /*     Overlay White color, Gray pat.        */
      unsigned short bit5: 1;               /*      (reserved bit)                       */
      unsigned short bit4: 1;               /*      (reserved bit)                       */
      unsigned short DimLeaveBorder: 1;     /*     Leave border when effect applied      */
      unsigned short DimAltImage: 1;        /*     Use an alternate image                */
      unsigned short DimNoChange: 1;        /*     Button looks same when disabled       */
      unsigned short AutoMoveSize: 1;       /*   Auto-resize as window's size changes    */
   } Bits;                                  /*                                           */
   long Num;                                /* · Long equivalent                         */
};                                          /*                                           */
typedef union TPPictButtonSpec TPPictButtonSpec;
```

Pascal

```
TPPictButtonSpec = packed record          {Picture Button's appearance and behavior        }
                                          {  specifications in 2 formats…                  }
    case integer of                       {                                                }
     0: (                                 {· Parsed into components:                       }
       InstantEvent: boolean;             {     Report event on mouse-down                 }
       TrackWithHilite: boolean;          {     Track w/highlight, like a radio button     }
       LockSelected: boolean;             {     Lock if selected (mouse can't deselect)    }
       SwitchSelected: boolean;           {     Switch 'selected' state if clicked         }
       RepeatEvents: boolean;             {     Repeat event when button is held down      }
       AutoValueChg: boolean;             {     Automatically change button's value        }
       AutoValueScaling: 0..3;            {     Rate of change for button's value          }
       ValueWrap: boolean;                {     Button's range of values 'wrap' around     }
       LeftRightSplit: boolean;           {     Left side reduces value, right increases   }
       TopBottomSplit: boolean;           {     Top increases value, bottom reduces        }
       MultiStage: boolean;               {     Button has multiple stages                 }
       BigSICN3D: boolean;                {     Create a larger SICN 3D button             }
       UsePICTS: boolean;                 {     Use PICTs instead of icons                 }
       Gray4use8: boolean;                {     Use 8-bit color pict on 4-bit gray monitor }
                                          {   Selection Effects…                           }
       SelectDarken: boolean;             {     Darken image                               }
       SelectDarkenSICN3D: boolean;       {     Darken (and push in) a 3D SICN icon        }
       SelectLightenSICN3D: boolean;      {     Lighten (and push in) a 3D SICN icon       }
       SelectPushedSICN3D: boolean;       {     Same color (and push in) a 3D SICN icon    }
       bit12, bit11: boolean;             {      (reserved bits)                           }
       SelectAltImage: boolean;           {     Use an alternate image                     }
       Hidden: boolean;                   {   Create a hidden button                       }
                                          {   Disabling Effects…                           }
       DimUsingBlackLite: boolean;        {     Overlay Black color using Lt Gray pat.     }
       DimUsingWhiteLite: boolean;        {     Overlay White color using Lt Gray pat.     }
       DimUsingWhite: boolean;            {     Overlay White color using Gray pat.        }
       bit5, bit4: boolean;               {      (reserved bits)                           }
       DimLeaveBorder: boolean;           {     Leave border when applying effect          }
       DimAltImage: boolean;              {     Use an alternate image                     }
       DimNoChange: boolean;              {     Button looks the same when disabled        }
       AutoMoveSize: boolean;             {   Auto-resize as window's size changes         }
     );                                   {                                                }
     1: (                                 {· Longint equivalent:                           }
       Num: longint;                      {     Specification longint                      }
     );                                   {                                                }
   end;
```

As an example, lets create a picture button that repeats events, uses an alternate image when selected, and looks the same when disabled. The following code sample illustrates how this is done:

```
procedure DoItNow;
  var
    Spec: TPPictButtonSpec;              {Define the variable used for the Spec            }
  begin
    Spec.Num := 0;                       {Initialize all the bits to zero values           }
    Spec.RepeatEvents := true;           {Button will produce repeating events             }
    Spec.SelectAltImage := true;         {Alternate image is used when button is selected  }
    Spec.DimNoChange := true;            {Button looks the same when disabled              }
                                         {Create the picture button using the long integer }
                                         {  part of the Spec…                              }
    NewPictButton(1, 441, 5, PlusIcon, Spec.Num, enabled, notSelected, 0, 0, 0); {         }
```

You can use whatever you like best as the Spec, a single constant, several constants added together, a variable, or the long integer component of a structure or variant record.

Rate of Repeating Events

Picture buttons have the ability to produce repeating events when they are held down. Four predefined rates are available to control the speed at which a picture button's value changes:

- *Linear*: The button's value changes when the button is selected. After a brief pause, the value continues to change at a slow and consistent rate.
- *Slow Acceleration*: The button's value changes when the button is selected. After a brief pause, the value continues to change at a rate that slowly accelerates.
- *Medium Acceleration*: The button's value changes when the button is selected. After a brief pause, the value continues to change at a moderately accelerating rate.
- *Fast Acceleration*: The button's value changes when the button is selected. After a brief pause, the value continues to change at a rate that rapidly accelerates.

There is yet another way to control a button's speed, and that is by using the SetPictButtonSpeed routine which lets you specify an exact rate (change in value per second). When you use SetPictButtonSpeed, the specified rate takes effect immediately when the user presses the picture button. Unlike the four standard Tools Plus acceleration rates, there is no pause between the time when the user selects the picture button and when the repeating events begin.

Note: Your event handler routine may receive doNothing events (no event) between repeating doPictButton events.

Picture Buttons on Color Backgrounds

If you are creating a picture button on a color surface, set the window's background color (by using SetBackRGB) to the color on which the picture button is being created, then create the button. After the button is created, you may change the window's foreground and background colors at any time without affecting picture buttons.

Each picture button remembers the color on which it is created, and uses this color when any erasing is performed by the picture button. This is required if you are using multiple images with masks that are not identical, because the picture button must erase the difference in space between the larger and smaller image.

```
CONST                                                  {Pict Button Behavior and Appearance Specs: }
               picbutInstantEvent        = $80000000;  {   Report event on mouse-down              }
               picbutTrackWithHilite     = $40000000;  {   Track w/highlight, like radio button    }
               picbutLockSelected        = $20000000;  {   Lock if selected (mouse can't deselect) }
               picbutSwitchSelected      = $10000000;  {   Switch 'selected' state if clicked      }
               picbutRepeatEvents        = $08000000;  {   Repeat event when button is held down   }
               picbutAutoValueChg        = $04000000;  {   Automatically change button's value     }
               picbutScaleLinear         = $00000000;  {   Rate of automatic value change…         }
               picbutScaleSlowAccel      = $01000000;  {   Linear, Slow Acceleration,              }
               picbutScaleMedAccel       = $02000000;  {   Medium Acceleration, and                }
               picbutScaleFastAccel      = $03000000;  {   Fast Acceleration.                      }
               picbutLinear              = 0;          {   Linear (use in structure)               }
               picbutSlowAccel           = 1;          {   Slow (use in structure)                 }
               picbutMedAccel            = 2;          {   Medium (use in structure)               }
               picbutFastAccel           = 3;          {   Fast (use in structure)                 }
               picbutValueWrap           = $00800000;  {   Button's range of values 'wrap' around  }
               picbutLeftRightSplit      = $00400000;  {   Left side reduces value, right increases}
               picbutTopBottomSplit      = $00200000;  {   Top increases value, bottom reduces     }
               picbutMultiStage          = $00100000;  {   Button has multiple stages              }
               picbutBigSICN3D           = $00080000;  {   Create a larger SICN 3D button          }
```

```
picbutUsePICTS            = $00040000; {   Use PICTs instead of icons            }
picbutGray4use8           = $00020000; {   Use 8-bit color pict on 4-bit gray mon. }
picbutHidden              = $00000200; { Create hidden picture button            }
picbutAutoMoveSize        = $00000001; { Auto-resize as window's size changes    }
                                       { Selection Effects…                      }
picbutSelectDarken        = $00010000; {   Darken image                          }
picbutSelectDarkenSICN3D  = $00008000; {   Darken (and push in) a 3D SICN icon   }
picbutSelectLightenSICN3D = $00004000; {   Lighten (and push in) a 3D SICN icon  }
picbutSelectPushedSICN3D  = $00002000; {   Same color (and push in) a 3D SICN icon }
picbutSelectAltImage      = $00000400; {   Use an alternate image                }
                                       { Disabling Effects…                      }
picbutDimUsingBlackLite   = $00000100; {   Overlay Black color using Lt Gray pat. }
picbutDimUsingWhiteLite   = $00000080; {   Overlay White color using Lt Gray pat. }
picbutDimUsingWhite       = $00000040; {   Overlay White color using Gray pat.   }
picbutDimLeaveBorder      = $00000008; {   Leave border when applying effect     }
picbutDimAltImage         = $00000004; {   Use an alternate image                }
picbutDimNoChange         = $00000002; {   Button looks the same when disabled   }
```

NewDialogPictButton

Create a new picture button in a dialog using a dialog item's co-ordinates.

C
```
pascal void NewDialogPictButton (short Button, short BaseID, long Spec,
            Boolean EnabledFlag, Boolean SelectedFlag,
            short minimum, short value, short maximum);
```

Pascal
```
procedure NewDialogPictButton (Button, BaseID: INTEGER; Spec: LONGINT;
            EnabledFlag, SelectedFlag: BOOLEAN;
            minimum, value, maximum: INTEGER);
```

NewDialogPictButton is identical to the NewButton routine, except that the button is created in a dialog (a window opened with the LoadDialog routine, or one that had a dialog list attached with the LoadDialogList routine). The button's co-ordinates are obtained from the dialog item whose number matches the button number.

GetFreePictButtonNum

Get the first unused picture button number.

C
```
pascal short GetFreePictButtonNum (void);
```

Pascal
```
function GetFreePictButtonNum: INTEGER;
```

Some developers may prefer to write code that more closely resembles a traditional Macintosh application, in that creating an object returns a reference to it such as a handle or pointer. Instead of having to assign your own picture button number, GetFreePictButtonNum returns the first unused (free) picture button number. Using this routine, you can assign an unused picture button number to a variable, then use that variable throughout your application without concern for the true picture button number.

GetFreePictButtonNum returns the first free picture button number on the current window. If the current window doesn't belong to your application, if no windows are open, or if the maximum number of picture buttons has already been created on the current window (no new ones can be created), GetFreePictButtonNum returns a value of zero (0).

DeletePictButton

Delete a picture button.

C
```
pascal void DeletePictButton (short Button);
```

Pascal
```
procedure DeletePictButton (Button: INTEGER);
```

Button specifies the picture button number (from 1 to 511) that is deleted from the current window. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, DeletePictButton does nothing. Use KillPictButton if you want to delete the picture button without removing its image from the window.

KillPictButton

Delete a picture button without affecting its image on the window.

C
```
pascal void KillPictButton (short Button);
```

Pascal
```
procedure KillPictButton (Button: INTEGER);
```

KillPictButton is identical to DeletePictButton except that it does not remove the picture button's image from the window. This routine is useful for scrolling picture buttons in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetPictButton repositions the picture button's co-ordinates without affecting its image (since ScrollRect has already moved it). KillPictButton then deletes the picture buttons that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

PictButtonDisplay

Hide or show a picture button.

C
```
pascal void PictButtonDisplay (short Button, Boolean Show);
```

Pascal
```
procedure PictButtonDisplay (Button: INTEGER; Show: BOOLEAN);
```

PictButtonDisplay hides or shows a picture button on the current window. The result is seen immediately. Use discretion with this routine since picture buttons should be enabled and disabled to indicate if they are accessible by the user.

Button specifies the picture button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, PictButtonDisplay does nothing.

Show indicates if the picture button is being hidden or displayed. The two constants that can be used for this flag are *on* and *off*.

PictButtonIsVisible

Determine if a picture button is visible.

C
```
pascal Boolean PictButtonIsVisible (short Button);
```

Pascal
```
function PictButtonIsVisible (Button: INTEGER): BOOLEAN;
```

PictButtonIsVisible reports if a picture button is visible on the current window, or if it is hidden.

Button specifies the picture button number (from 1 to 511) that is queried in the current window.

This routine's value returns *true* if the picture button is visible, and *false* if the button is hidden. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, PictButtonIsVisible returns *false*.

ObscurePictButton

Hide a picture button without removing its image from the window.

C
```
pascal void ObscurePictButton (short Button);
```

Pascal
```
procedure ObscurePictButton (Button: INTEGER);
```

ObscurePictButton hides a picture button on the current window without removing its image from the window. This routine is useful for scrolling buttons in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetPictButton repositions the button's co-ordinates without affecting its image (since ScrollRect has already moved it). ObscurePictButton then hides the buttons that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

Button specifies the picture button number (from 1 to 511) that is hidden in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, ObscurePictButton does nothing.

GetPictButtonRect

Get a picture button's co-ordinates.

C
```
pascal void GetPictButtonRect (short Button, Rect *Bounds);
```

Pascal
```
procedure GetPictButtonRect (Button: INTEGER; var Bounds: RECT);
```

Button specifies the picture button number (from 1 to 511) that is queried in the current window.

Bounds returns the picture button's bounding rectangle specified in the window's local co-ordinates. These co-ordinates match those used to create the picture button. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, Bounds returns with all co-ordinates set to zero (0). The left and top co-ordinates of *Bounds* are identical to those specified when creating a picture button. The bottom and right co-ordinates are determined each time the picture button is displayed using the button's current image co-ordinates.

EnablePictButton

Enable or disable a picture button.

C
```
pascal void EnablePictButton (short Button, Boolean EnabledFlag);
```

Pascal
```
procedure EnablePictButton (Button: INTEGER; EnabledFlag: BOOLEAN);
```

Button specifies the picture button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, EnablePictButton does nothing.

The *EnabledFlag* indicates if the picture button is enabled or not. All three picture button types can be either enabled or disabled. When a picture button is disabled, it becomes dim and cannot be selected by the user. All picture buttons automatically become disabled when the window containing them is inactive. When the window is activated, the picture buttons assume their state as set by the NewPictButton routine, and subsequent calls to the EnablePictButton routine. The two constants that can be used for this flag are *enabled* and *disabled.*

```
CONST                               {Button state                            }
               enabled      = true;  {picture button is enabled              }
               disabled     = false; {picture button is disabled             }
```

See the NewPictButton routine for additional information pertaining to the picture button's enabling, disabling, and selection.

PictButtonIsEnabled

Determine if a picture button is enabled or disabled.

C
```
pascal Boolean PictButtonIsEnabled (short Button);
```

Pascal
```
function PictButtonIsEnabled (Button: INTEGER): BOOLEAN;
```

Button specifies the picture button number (from 1 to 511) that is queried in the current window.

The routine's value returns *true* if the picture button is enabled, and *false* if the button is disabled. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, PictButtonIsEnabled returns *false*. PictButtonIsEnabled returns the button's enabled state as it is currently displayed, so if the button's window is inactive and has temporarily disabled the button, PictButtonIsEnabled returns *false*.

SelectPictButton

Select or deselect a picture button.

C
```
pascal void SelectPictButton (short Button, Boolean SelectedFlag);
```

Pascal
```
procedure SelectPictButton (Button: INTEGER; SelectedFlag: BOOLEAN);
```

Button specifies the picture button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, SelectPictButton does nothing.

The *SelectedFlag* indicates if the picture button is selected or not. The two constants that can be used for this flag are *selected* and *notSelected.*

```
CONST                             {Button state                        }
              selected     = true;   {picture button is selected          }
              notSelected  = false;  {picture button is not selected      }
```

See the NewPictButton routine for additional information pertaining to the picture button's enabling, disabling, and selection.

PictButtonIsSelected

Determine if a picture button is selected.

C
```
pascal Boolean PictButtonIsSelected (short Button);
```

Pascal
```
function PictButtonIsSelected (Button: INTEGER): BOOLEAN;
```

Button specifies the picture button number (from 1 to 511) that is queried in the current window.

The routine's value returns *true* if the picture button is selected, and *false* if the picture button is not selected. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, PictButtonIsSelected returns *false*.

```
CONST                             {Button state                        }
              selected     = true;   {picture button is selected          }
              notSelected  = false;  {picture button is not selected      }
```

See the NewPictButton routine for additional information pertaining to the picture button's enabling, disabling, and selection.

GetPictButtonMin

Get a picture button's minimum value limit.

C
```
pascal short GetPictButtonMin (short Button);
```

Pascal
```
function GetPictButtonMin (Button: INTEGER): INTEGER;
```

Button specifies the picture button number (from 1 to 511) that is queried in the current window.

GetPictButtonMin returns a picture button's minimum value limit. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, GetPictButtonMin will return a value of zero (0).

SetPictButtonMin

Set a picture button's minimum value limit.

C
```
pascal void SetPictButtonMin (short Button, short minimum);
```

Pascal
```
procedure SetPictButtonMin (Button, minimum: INTEGER);
```

Button specifies the picture button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, SetPictButtonMin does nothing. The minimum and maximum limit cannot be changed in multistage buttons.

Minimum specifies the picture button's new minimum value limit. The picture button's current value and maximum limit are automatically adjusted (if necessary) to be consistent with the new minimum limit.

GetPictButtonMax

Get a picture button's maximum value limit.

C
```
pascal short GetPictButtonMax (short Button);
```

Pascal
```
function GetPictButtonMax (Button: INTEGER): INTEGER;
```

Button specifies the picture button number (from 1 to 511) that is queried in the current window.

GetPictButtonMax returns a picture button's maximum value limit. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, GetPictButtonMax will return a value of zero (0).

SetPictButtonMax

Set a picture button's maximum value limit.

C
```
pascal void SetPictButtonMax (short Button, short maximum);
```

Pascal
```
procedure SetPictButtonMax (Button, maximum: INTEGER);
```

Button specifies the picture button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, SetPictButtonMax does nothing. The minimum and maximum limit cannot be changed in multistage buttons

Maximum specifies the picture button's new maximum value limit. The picture button's current value and minimum limit are automatically adjusted (if necessary) to be consistent with the new maximum limit.

GetPictButtonVal

Get a picture button's current value.

C
```
pascal short GetPictButtonVal (short Button);
```

Pascal
```
function GetPictButtonVal (Button: INTEGER): INTEGER;
```

Button specifies the picture button number (from 1 to 511) that is queried in the current window.

GetPictButtonVal returns a picture button's current value. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, GetPictButtonVal will return a value of zero (0).

SetPictButtonVal

Set a picture button's current value.

C
```
pascal void SetPictButtonVal (short Button, short value);
```

Pascal
```
procedure SetPictButtonVal (Button, Value: INTEGER);
```

Button specifies the picture button number (from 1 to 511) which is to be affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, SetPictButtonVal does nothing.

Value specifies the picture button's new current value. In multistage buttons, the value is adjusted to fit within the button's minimum and maximum limit. In single stage buttons, the minimum and maximum limits are automatically adjusted (if necessary) to be consistent with the new current value.

SetPictButtonValSelect

Set a picture button's current value, and simultaneously select or deselect the button.

C
```
pascal void SetPictButtonValSelect (short Button, short value,
             Boolean SelectedFlag);
```

Pascal
```
procedure SetPictButtonValSelect (Button, Value: INTEGER;
             SelectedFlag: BOOLEAN);
```

Sometimes, it is necessary to simultaneously change a button's value and to select or deselect it, otherwise the transition from one stage to another would look jerky. An example of this is a multistage button that locks in the selected position, then lets your application determine if conditions allow the button to be deselected in the *next* stage (indicating acceptance of the button's action), or deselected in the same stage (indicating rejection of the button's action).

Button specifies the picture button number (from 1 to 511) which is to be affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, SetPictButtonValSelect does nothing.

Value specifies the picture button's new current value. In multistage buttons, the value is adjusted to fit within the button's minimum and maximum limit. In single stage buttons, the minimum and maximum limits are automatically adjusted (if necessary) to be consistent with the new current value.

The *SelectedFlag* indicates if the picture button is selected or not. The two constants that can be used for this flag are *selected* and *notSelected*.

```
CONST                               {Button state                            }
          selected     = true;   {picture button is selected              }
          notSelected  = false;  {picture button is not selected          }
```

SetPictButtonAccel

Set a picture button's value change rate.

C
```
pascal void SetPictButtonAccel (short Button, short Rate);
```

Pascal
```
procedure SetPictButtonAccel (Button, Rate: INTEGER);
```

Button specifies the picture button number (from 1 to 511) which is to be affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, SetPictButtonAccel does nothing.

Rate specifies the rate at which the picture button's value changes. The four constants that can be used for this setting are:

`picbutLinear`	*Linear*: The button's value changes when the button is selected. After a brief pause, the value continues to change at a slow and consistent rate.
`picbutSlowAccel`	*Slow Acceleration*: The button's value changes when the button is selected. After a brief pause, the value continues to change at a rate that slowly accelerates.
`picbutMedAccel`	*Medium Acceleration*: The button's value changes when the button is selected. After a brief pause, the value continues to change at a moderately accelerating rate.
`picbutFastAccel`	*Fast Acceleration*: The button's value changes when the button is selected. After a brief pause, the value continues to change at a rate that rapidly accelerates.

The affected picture button must be created with the "automatic value change" and "repeating events" options both turned on for this routine to have any effect. Using SetPictButtonAccel overrides the settings established by the SetPictButtonSpeed routine.

```
CONST                                 {Value change rates:
              picbutLinear    = 0;    {Linear (does not accelerate)         }
              picbutSlowAccel = 1;    {Slow acceleration                    }
              picbutMedAccel  = 2;    {Medium acceleration                  }
              picbutFastAccel = 3;    {Fast acceleration                    }
```

See the NewPictButton routine for additional information pertaining to the picture button's automatic value change rate. Also see the SetPictButtonSpeed routine for another method of setting the button's speed.

SetPictButtonSpeed

Set a picture button's value change speed.

C
```
pascal void SetPictButtonSpeed (short Button, short Rate);
```

Pascal
```
procedure SetPictButtonSpeed (Button, Rate: INTEGER);
```

Button specifies the picture button number (from 1 to 511) which is to be affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the picture button does not exist in the current window, SetPictButtonSpeed does nothing.

Rate specifies the constant speed at which the picture button's value changes. The rate is expressed as an amount that is incremented per second (i.e., "30" means change the button's value by 30 for each second it is held down).

The picture button's value will change at the specified speed as soon as the user presses down on the button (there is no pause before the event starts repeating). The affected picture button must be created with the "automatic value change" and "repeating events" options both turned on for this routine to have any effect. Using SetPictButtonSpeed overrides the settings established by the SetPictButtonAccel routine.

See the SetPictButtonAccel routine for another method of setting the button's value change rate.

FlashPictButton

Flash a picture button as though it was clicked by the user.

C
```
pascal void FlashPictButton (short Button);
```

Pascal
```
procedure FlashPictButton (Button: INTEGER);
```

Button specifies the picture button number (from 1 to 511) that is affected in the active window. If the active window doesn't belong to your application, or if no windows are open, FlashPictButton does nothing.

FlashPictButton can be used in some specific instances. Advanced programmers may decide to display a modal window when the Macintosh is busy with a lengthy process. If a picture button (such as "Cancel") on this window is equivalent to typing ⌘-., your application should flash the picture button when a ⌘-. is reported to your event handler routine. This makes the user feel that the key triggered the picture button. Another example is double-clicking in a list box; this action can be interpreted as "select line and OK" in which case the OK picture button should be flashed.

MovePictButton

Move a picture button to a new location on the window.

C
```
pascal void MovePictButton (short Button, short toHoriz, short toVert);
```

Pascal
```
procedure MovePictButton (Button, toHoriz, toVert: INTEGER);
```

Button specifies the picture button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Button* specifies a picture button that does not exist, MovePictButton does nothing. The change is seen immediately providing that the picture button is not hidden. The picture button's width and height are not changed.

ToHoriz is the new horizontal co-ordinate at which the left side of the picture button appears.

ToVert is the new vertical co-ordinate at which the top of the picture button appears.

OffsetPictButton

Change a picture button's co-ordinates without affecting its image on the window.

C
```
pascal void OffsetPictButton (short Button, short distHoriz, short distVert);
```

Pascal
```
procedure OffsetPictButton (Button, distHoriz, distVert: INTEGER);
```

When you scroll an area that contains picture buttons, first use ScrollRect to scroll the pixel image containing the affected objects in the window. OffsetPictButton is used to offset a picture button's co-ordinates without altering its image (since ScrollRect has already done so). At this point, the picture button's co-ordinates match the scrolled image of the picture button. ObscurePictButton or KillPictButton can be used to hide or delete picture buttons that are scrolled out of view.

Button specifies the picture button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Button* specifies a picture button that does not exist, OffsetPictButton does nothing.

DistHoriz and *distVert* specify the horizontal and vertical amount by which the picture button's co-ordinates are offset. Positive numbers are right and down. The picture button's co-ordinates are updated but no change is seen.

AutoMoveSizePictButton

Specify how a picture button is automatically moved as its window's size is changed.

C
```
pascal void AutoMoveSizePictButton (short Button, Boolean left, Boolean top);
```

Pascal
```
procedure AutoMoveSizePictButton (Button: INTEGER; left, top: BOOLEAN);
```

Button specifies the picture button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Button* specifies a picture button that does not exist, AutoMoveSizePictButton does nothing.

The *left* and *top* parameters specify if that side of the picture button is automatically adjusted when the window's size changes. These setting are applied to the picture button and are used the next time the window's size changes:

left	Do the picture button's left and right side track the window's right edge?
top	Do the picture button's top and bottom track the window's bottom edge?

You can think of each *false* value as locking that side of the picture button to a fixed co-ordinate regardless of the window's size (this is the default). Each *true* value establishes a fixed distance between that side of the picture button and the window's edge. For example, setting only *left* to *true* makes the picture button move horizontally as the window widens and narrows, but the picture button does not move vertically when the window's height changes.

If you are setting these values identically for a group of objects, use AutoMoveSize to define the settings then add the appropriate *x*AutoMoveSize constant (such as picbutAutoMoveSize for picture buttons) to the objects' spec as they are created. The objects will adopt the settings specified by the AutoMoveSize routine.

☞ **Warning**: Make sure that you resize objects in a way that makes sense. Don't allow a window to shrink down to a size where objects become unusable or disappear altogether.

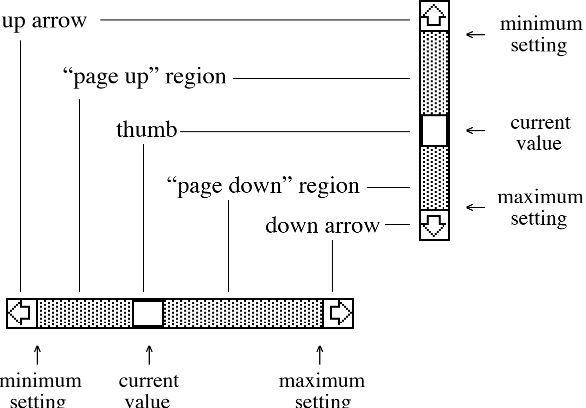
8 Scroll Bars

Tools Plus supports the use of scroll bars on any Tools Plus window. Additionally, custom control definitions (CDEFs) that are similar to scroll bars, such as sliders, can also be used. Within this manual, the term “scroll bar” refers to a real scroll bar or a custom CDEF that works like a scroll bar.

Scroll bars are created on the current window by the NewScrollBar routine. Each scroll bar is referenced by a unique scroll bar number that can be from 1 to 511. This number is specified when the scroll bar is created, and refers to the specific scroll bar until that scroll bar is deleted. Note that the scroll bar number is related to its associated window. This means that two different windows can each have a scroll bar numbered “1” without interfering with each other. Whenever a scroll bar is used by the user, Tools Plus calls your event handler routine and reports the scroll bar number, the part (see below) that was used, and its window number. You can also create a scroll bar from a ‘CNTL’ resource by using the LoadScrollBar routine.

Scroll bars can be moved to a new location with MoveScrollBar and have their width and/or height changed with SizeScrollBar. MoveSizeScrollBar combines both tasks by letting you specify new co-ordinates for the scroll bar.

Scroll bars can be either horizontal or vertical, and are made up of five distinct parts: [1] up arrow, [2] “page up” region, [3] thumb (also called an indicator), [4] “page down” region, and [5] down arrow.



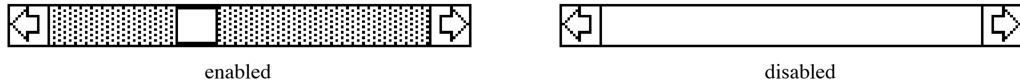
A scroll bar’s minimum and maximum settings can be obtained by the GetScrollBarMin and GetScrollBarMax routines. The current value can be obtained by the GetScrollBarVal routine. Conversely, these values can be set by using the SetScrollBarMin, SetScrollBarMax, and SetScrollBarVal routines.

When a scroll bar is no longer required, it is deleted by the DeleteScrollBar routine, which releases the memory used by the scroll bar. This is done automatically if a window is closed. A scroll bar can be hidden or displayed with the ScrollBarDisplay routine.

Tools Plus also supports the use of custom CDEFs as scroll bars, as well as the extended set of controls that are part of the Appearance Manager which first appeared in Mac OS 8. Many of these controls are implemented as scroll bars and are detailed in this section. See the chapter on Buttons for details on the remaining Appearance Manager controls.

Scroll Bar States

A scroll bar is enabled or disabled by the EnableScrollBar routine. When a window is inactive, all the associated scroll bars are automatically hidden (only the outline is displayed) and cannot be selected. When the window is activated, the scroll bars return to their normal status as set by your application. This standard behavior can be overridden to merely disable scroll bars on inactive windows, which is more appropriate for custom CDEFs like sliders.



enabled disabled

Colors

By default, Apple's scroll bars have a black frame and a background that matches their parent window's backdrop color (which is white by default). The rest of the scroll bar is colored appropriately. Third party CDEFs used as scroll bars may behave differently. Optionally, each scroll bar can adopt unique color settings as it is created. The colors for the various scroll bar parts are defined by the ScrollBarColors routine, and are optionally adopted by scroll bars as they are created. Scroll bars' colors can be changed afterwards using the SetScrollBarColors routine. Conversely, the GetScrollBarColors routine retrieves a scroll bar's color settings.

When designing applications, always design them in black and white then apply color (if required) to add value to your application. Don't add color just because you can. In the case of color scroll bars, test your color selection thoroughly on a monitor set to 8, 4, and 2-bit color and gray scale, and black and white to ensure that your colors and window backdrop color map to usable colors. Note that some controls ignore color settings, particularly those in the Appearance Manager.

Text

Scroll bars do not normally have any associated text. A custom CDEF like a slider may have text such as numbers that are part of a numeric scale. When your application creates a scroll bar, it memorizes the window's text settings as set by TextFont, TextSize, and TextFace, and it retains those setting whenever the scroll bar is used. This way, custom CDEFs that have a "use window font" variant code will be able to use the window's font setting that were established when the scroll bar was created. This also facilitates custom CDEFs that always display text using the window's current font settings. CDEFs that do not display any text will suffer no ill effects from this strategy.

If you are creating scroll bars that display text, set the window's font settings as required using TextFont, TextSize and TextFace before the scroll bar is created. Once the scroll bar is created, it will automatically have access to those text settings. You can use the GetScrollBarFontSettings and SetScrollBarFontSettings routines to get and set the scroll bar's font, size and style settings.

Scroll Bar Speed

Normally, scroll bars generate doScrollBar events as quickly as your application can handle them. Tools Plus lets you control the rate at which doScrollBar events are generated, thereby letting you slow down a scroll bar to an ideal speed. As a beneficial side effect, scroll bars move at a *consistent* speed to compensate for time-consuming processes in your application, such as those that display the material that is scrolled by a scroll bar.

ScrollBarLineTime and ScrollBarPageTime are used to specify the rate at which scroll bars move a line at a time (when the up/down arrows are used), and a page at a time (when the page up/down regions are used). Subsequently created scroll bars adopt the settings specified by these routines. Similarly, SetScrollBarLineTime and SetScrollBarPageTime set the rate for an individual scroll bar.

Substituting Scroll Bar ProcIDs

Certain system resources may or may not be available to your application depending on the system version of the Macintosh that is running your application. A good example of this is the sliders that are part of the Appearance Manager in Mac OS 8 or later. With Tools Plus, you can design and write your application to use a custom slider (CDEF resource) to provide sliders in your application, such as those in SuperCDEFs. Then at the beginning of your application it can determine the Mac's capabilities, specifically if the Appearance Manager is running to make the system's sliders available to your application. If this is the case, your application can easily substitute the use of the custom slider CDEF with the Appearance Manager's slider throughout your application.

Two routines in the Miscellaneous Routines chapter of this manual help facilitate determining the capabilities of the Macintosh that is running your application: HasAppearanceManager and UsingAppearanceManager. You can also use the toolbox's Gestalt routines to determine whether other features are available or not. Tools Plus's ReplaceControlProcID routine is used to replace a specific scroll bar procID with another procID throughout your application, thereby substituting the use of one type of scroll bar (or slider) with another. The ReplaceControlProcID routine is detailed in the Buttons chapter of this manual.

Handling Scroll Bars

Your application specifies if a scroll bar is enabled or disabled. When a window in inactive, Tools Plus disables all scroll bars on that window. When the window is activated again, all the scroll bars regain their correct status as specified by your application. If a window contains a scroll bar along its right side and/or bottom (such as on word processing documents and spreadsheets), these scroll bars are automatically sized and moved if the user drags the window's "size box" (providing that the window has a "size box") or clicks the "zoom box."

Processing doScrollBar Events

There are two basic ways your application can respond to the user's interaction with a scroll bar. The first one, the easier of the two alternatives, is to have your application simply respond to doScrollBar events. When Tools Plus detects a mouse-down event in a scroll bar, it calls your event handler routine and reports it as a doScrollBar event. The event also includes information about which part was clicked: up button, down button, page up region, page down region, or thumb. In the case of all the up/down possibilities, your application should respond by scrolling the screen's image if required (using the toolbox's ScrollRect routine), updating the scroll bar with its new value (using the SetScrollBarVal routine), and possibly offsetting user interface element co-ordinates. With Tools Plus, your application will get a series of doScrollBar events as long as the user holds the mouse down and has the cursor in the originally clicked region. See the tutorials for examples of how to scroll user interface elements.

When Tools Plus reports a doScrollBar event, the part code may indicate the event is a result of the user moving the scroll bar's thumb, in which case your application can obtain the scroll bar's value by using the GetScrollBarVal routine then scrolling the required area. Tools Plus supports optional "live scrolling" that causes the scroll bar to move its thumb in real time as it tracks the cursor. During this tracking, your event handler routine gets a doScrollBar event each time the scroll bar's value changes. The live scrolling feature works with virtually any CDEF that behaves like a scroll bar, including Apple's scroll bars and third party sliders. It's easy for you to program this because as far as your application is concerned, the user is moving the scroll bar's thumb in a series of steps.

Action routine

The second method of responding to the user's interaction with a scroll bar is the one originally designed by Apple, that being creating an "action routine" and installing it in a scroll bar with the SetScrollBarAction routine. Your action routine is called continuously while the user interacts with the scroll bar, be it holding the mouse in the scroll bar's up button or while dragging the scroll bar's thumb. Your action routine can call GetScrollBarActionInfo to determine which scroll bar is being called, its parent window, the part that was clicked by the user, and if the mouse is still in the originally clicked part (i.e., is the mouse still in the page up region).

In some cases, your application will experience better performance by using the action routine. A typical case where this is true is using a slider to control the volume of music in real time, such as in an audio mixer. Beware that some CDEFs (like the Apple scroll bar) do not change the scroll bar's value when the user drags the thumb. The value is changed only when the user releases the thumb, so an action routine is ineffective in trying to regulate or control something in real time. This is when you should use Tools Plus's live scrolling option in conjunction with processing doScrollBar events.

Water's Edge Software offers a set of slider CDEFs that can be used with either method of handling scroll bar. They can easily be integrated into your existing application without having to change any code. They can also be used in the most demanding real-time control environments where you can create an action routine that responds to the indicator's movement in real time.

Warning: If you have obtained a handle to a scroll bar, do not change any of the fields in the scroll bar's record.

Appearance Manager Controls

The Appearance Manager, first introduced in mid 1997 with Mac OS 8, gives your application a number of 3D controls in addition to the ordinary push button, check box, radio button, and scroll bar that were originally supplied by Apple when Macintosh debuted in 1984. All the new Appearance Manager controls are implemented as CDEFs, but unlike third party CDEF resources that must be installed in your application when it is built, the Appearance Manager's controls are available to your application without having to install them. They are available from the system, just like regular system controls, if the Macintosh running your application has an Appearance Manager.

Your application can access the Appearance Manager's 3D push buttons, check boxes, radio buttons and scroll bars without any special programming. In fact, you can replace the standard controls throughout your application with the equivalent Appearance Manager controls as a default behavior when you initialize Tools Plus libraries with the InitToolsPlus routine. However, if you want to make use of other Appearance Manager controls and features, you need to make your application "Appearance Manager aware." 680x0 applications are automatically Appearance Manager aware. To make your PowerPC application Appearance Manager aware, see the Designing Your Application chapter of this manual for details in the "Using the Appearance Manager" section.

Many of the Appearance Manager's controls are considerably more complex than the standard controls, and understandably so because they offer considerably more features. Many controls place special significance on their initial values when they are created, specifically the control's minimum limit, maximum limit and current value (these items equate to the contrlMin, contrlMax and contrlValue fields of the Control Manager's ControlRecord record). Constants for these controls and all their options appear in the Appearance.h (C/C++ header) and Appearance.p (Pascal interface) files, as well as in Controls.h and Controls.p files.

See the chapters on Buttons, Editing Fields, List Boxes and Pop-Up Menus in this user manual for additional Appearance Manager controls.

Note: For complete information on Appearance Manager concepts, the Appearance Manager's features, and how to best use the Appearance Manager's new controls, please read the documentation pertaining to the Appearance Manager. It is available from Apple or in the latest issue of Inside Macintosh. This manual does not duplicate that material.

Scroll Bar (CDEF 24)

This scroll bar works identically to a standard scrollBarProc scroll bar.

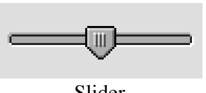
Enabled

Disabled

```
CONST
   kControlScrollBarProc     = 384;     {Normal Scroll Bar ProcID            }
   kControlScrollBarLiveProc = 386;     {Live scrolling variant              }
```

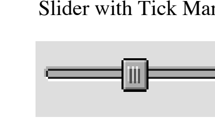
Slider (CDEF 3)

The slider works similarly to a scroll bar, except it has no page up, page down, line up or line down regions. The user can only set a slider by dragging the indicator.



Slider

```
CONST
   kControlSliderProc             = 48;   {Slider ProcID                      }
   kControlSliderLiveFeedback     = $01;  {Live scrolling variant             }
                                          {Add these variants to the ProcID… }
   kControlSliderHasTickMarks     = $02;  {Slider has tick marks              }
   kControlSliderReverseDirection = $04;  {Thumb points in opposite direction
                                          }
   kControlSliderNonDirectional   = $08;  {Thumb is non-directional           }
```



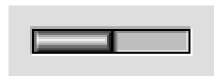
Slider with Tick Marks

Non-Directional

When creating a slider with tick marks, the control's initial value is used to determine the number of tick marks that appear in the slider.

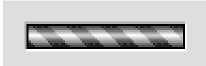
Progress Indicator or "Thermometer" (CDEF 5)

The progress indicator is implemented like a scroll bar, but unlike a scroll bar, the user cannot interact with this control. The standard progress indicator's height is 14 pixels.



Determinate

```
CONST
   kControlProgressBarProc     = 80;      {Progress Indicator ProcID         }
   scrlBusyThermometerMinLimit = -32768;  {Minimum value for indeterminate   }
                                          {  indicator.                      }
```



Indeterminate

Your application sets the progress indicator's current value such that it indicates a relative progress between the indicator's minimum limit and maximum limit. An "indeterminate" state can exist when the application does not know how long a task will take. In Tools Plus, the progress indicator assumes an indeterminate state when its minimum limit is set to -32768.

An indeterminate indicator animates automatically each time your event handler routine finishes executing. If you need to animate the indicator more frequently, see the Process1EventWhileBusy routine for details.

Little Arrows (CDEF 6)

Little Arrows are used to increase or decreased a value, as seen in the Clock control. In Tools Plus, this control can be implemented either as a button to allow the user to step through a series of values one at a time with each click, or as a scroll bar to allow the user to also hold the up arrow or down arrow to continuously increase or decrease a value while the button is held down.

Little Arrows

If you are using a 'CNTL' resource to create this control, add 1 to the procID to tell Tools Plus that you want to implement the Little Arrows control as a scroll bar, otherwise it is implemented as a button. Little Arrows should always be created in a rectangle that is 13 pixels wide by 23 pixels high.

```
CONST
   kControlLittleArrowsProc = 96;       {Little Arrows ProcID                }
```

Appearance Manager and Keyboard Focus

Before the Appearance Manager's arrival, the only user interface element that could process keystrokes was an editing field. With the Appearance Manager present, a variety of user interface elements can process keystrokes, such as editing fields, list boxes and the clock control. Keystrokes are directed at only one user interface element at a time (and possibly at no element at all). When a user interface element processes keystrokes in such a way, it is said to have the "keyboard focus." Only one user interface element can have the keyboard focus at a time, and it is visually indicated with a highlighted "band" around the object. Tools Plus takes care of the focus highlight, and of applying keystrokes to the element that has the keyboard focus.

The process of moving the keyboard focus between objects, either by tabbing or clicking, is identical to that of navigating between editing fields. For details, see the "Clicking and Tabbing" section in the Editing Fields chapter.

NewScrollBar

Create a new scroll bar.

C
```
pascal void NewScrollBar (short ScrollBar,
            short left, short top, short right, short bottom,
            long Spec, Boolean EnabledFlag,
            short minimum, short value, short maximum);
```

Pascal
```
procedure NewScrollBar (ScrollBar: INTEGER;
            left, top, right, bottom: INTEGER;
            Spec: LONGINT; EnabledFlag: BOOLEAN;
            minimum, value, maximum: INTEGER);
```

ScrollBar specifies the scroll bar number (from 1 to 511) that is created in the current window. Once a scroll bar is created, it is referenced by this scroll bar number. If a scroll bar has been previously created in the current window using the same number, it is replaced with a new scroll bar as specified by the parameters in the NewScrollBar routine. If the current window doesn't belong to your application, or if no windows are open, NewScrollBar does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in local co-ordinates that determines the scroll bar's size and location in the window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). A scroll bar is vertical or horizontal depending on whether the height or width of the rectangle is greatest. Scroll bars should be exactly 16 pixels wide, so there should be a 16 pixel difference between the scroll bar's top and bottom, or left and right side. If there isn't, the scroll bar is scaled to fit into the rectangle and will not look as attractive. Also, the scroll bar must be at least 40 pixels long in order to contain the up arrow, down arrow, and thumb.

For windows with a ProcID of documentProc (i.e., with a "size box") special scroll bars may be created along the right side of a window and/or along the bottom. These scroll bars are special because they are automatically sized and moved if the window's size is changed. Here are some useful measurements and constants for these specialized scroll bars:

	Co-Ordinates			
Type of Scroll Bar	**Left**	**Top**	**Right**	**Bottom**
At window's right edge	scrlRightEdge *(window width - 15)*	scrlTopEdge *(or -1) (optional)*	scrlRightEdge *(window width + 1)*	*(your co-ordinate)*
At window's right edge, bottom linked to grow box	scrlRightEdge *(window width - 15)*	scrlTopEdge *(or -1) (optional)*	scrlRightEdge *(window width + 1)*	scrlBottomEdge
At window's bottom	scrlLeftEdge *(or -1) (optional)*	scrlBottomEdge *(window height-15)*	*(your co-ordinate)*	scrlBottomEdge *(window height + 1)*
At window's bottom, right side linked to grow box	scrlLeftEdge *(or -1) (optional)*	scrlBottomEdge *(window height-15)*	scrlRightEdge	scrlBottomEdge *(window height + 1)*

(The window's dimensions can be obtained from the WindowStatus routine.)

Spec specifies the scroll bar's appearance and behavior characteristics. It is a combination of a control procID plus various Tools Plus options detailed later in this section.

EnabledFlag specifies if the scroll bar is enabled or disabled when the window is active. When a scroll bar is disabled, the thumb and toned "page up" and "page down" regions disappear, and the scroll bar cannot be used by the operator. The two constants that can be used for this purpose are *enabled* and *disabled*. All scroll bars automatically become hidden (only the outline is shown) when the window containing them becomes inactive. When the window is activated, the scroll bars will assume their normal state as set by the NewScrollBar routine, and subsequent calls to the EnableScrollBar routine. Below is an example:

EnabledFlag = enabled   EnabledFlag = disabled

Minimum declares the scroll bar's minimum limit.

Value defines the scroll bar's current value. The current value must be greater than or equal to the minimum setting, and less than or equal to the maximum setting.

Maximum declares the scroll bar's maximum limit. The maximum limit must be greater than the minimum limit.

Appearance and Behavior Specification

Spec specifies the scroll bar's appearance and behavior characteristics. The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result. The constants defining the available options are as follows:

Choose only one of the following procIDs…

`scrollBarProc`	Standard Apple scroll bar.
`scrlStandard`	Same as the standard Apple scroll bar procID.
(your own procID)	You can use your own scroll bar or slider procID and Tools Plus will make it work. When using custom control definitions (CDEFs), realize that the procID specifies both the control's resource number as well as optional variants (low 4 bits) that are ignored by Tools Plus but may be used by the CDEF. The procID is calculated as follows: CDEF resource ID x 16 + the optional variants (0-15).

Optionally choose any of the following options…

`scrlColorScrollBar`	Adopt the color settings as defined by the ScrollBarColors routine. By default, scroll bars have a black frame and a background that matches their parent window's backdrop color (which is white by default). Note that some controls ignore color settings, particularly those in the Appearance Manager.
`scrlLiveScroll`	Scroll an object in real time as the user moves a scroll bar's thumb. This is not a Macintosh user interface standard. By default, an outline tracks the mouse as the user drags the scroll bar's thumb, then when the user releases the mouse button, the scroll bar's thumb snaps to the new position and generates an event. When using this option, the thumb tracks the mouse position and generates a doScrollBar event when the thumb moves to a new position.
`scrlValueLimit`	When setting the scroll bar's value, it is always limited by the scroll bar's minimum and maximum limit. By default, if you set a value that is lower than the minimum limit or higher than the maximum limit, Tools Plus adjusts the minimum or maximum limit to accommodate the new value. With this option, if a scroll bar's minimum limit is 0 and maximum is 100 and you specify a new value of 110 it will be adjusted to 100 to prevent

	exceeding the scroll bar's maximum limit.
`scrlNoObscure`	Display the scroll bar as disabled when it is on an inactive window. This is the preferred behavior for custom CDEFs. By default, a scroll bar is drawn as an outlined frame when it is on an inactive window.
`scrlAutoMoveSize`	Automatically move and/or resize the scroll bar when the window's size changes. The AutoMoveSize routine lets you specify which sides are altered. You can use the AutoMoveSizeScrollBar routine as an alternative to setting this option.
`scrlHidden`	Create a hidden scroll bar. This kind of scroll bar is accessible to your application but not to the user.

Custom Control Definitions (CDEFs)

Your application can use custom control definitions (CDEFs) on a per-scroll bar basis. If your CDEF is written to Apple's specifications, Tools Plus will make your custom scroll bar work like a regular scroll bar or slider. When using a custom CDEF, you will need to include a special control definition (CDEF resource) in your application's resource fork. Add the required CDEF resource to your project's resource file before you compile your application. Tools Plus includes custom CDEFs in the "Optional Resources" folder.

You can write your own CDEFs or you can use third-party CDEFs. As per Macintosh standards, a control's procID is comprised from the following formula: CDEF ID x 16 + *variant code*. Your CDEF's ID can be in the range of 2 to 2047. Make sure your CDEF resource IDs do not conflict with System resources (i.e., the standard Apple button CDEF ID is 0).

Note: When using third party CDEFs, make sure you carefully read the documentation that accompanies the CDEF. If your scroll bar is irregularly shaped, like most sliders, and it is on a manually drawn background (other than a window's backdrop), that background must be refreshed in response to a doPreRefresh event. Tools Plus removes your scroll bar's region from the update region when it generates the doRefresh event, thereby protecting it from being overwritten.

Note: Tools Plus makes no attempt to control the placement of scroll bars or to protect them once they have been created. It is your responsibility to ensure that scroll bars are of sufficient length to contain the up/down arrows and the thumb, and that their placement within the window is reasonable and does not conflict with other objects. Furthermore, you should not allow your application's text and drawing processes to interfere with scroll bars. Windows with a "size box" should not allow scroll bars to be obscured or hidden by making the window too small.

Also see: SetAutoEmbed (in the Buttons chapter), NewScrollBarRect, NewDialogScrollBar, ScrollBarColors and ReplaceControlProcID.

```
CONST                                     {Scroll bar appearance and behavior:         }
               enabled            = true;      {Enable the scroll bar                       }
               disabled           = false;     {Disable the scroll bar                      }
                                               {Automatic placement on window's edge…       }
               scrlLeftEdge       =-32768;     {  left edge of document                     }
               scrlTopEdge        =-32768;     {  top edge of document                      }
               scrlRightEdge      = 32767;     {  right edge of document                    }
               scrlBottomEdge     = 32767;     {  bottom edge of document                   }
               scrlStandard       = $00000010; {Standard scroll bar (default)               }
               scrlLiveScroll     = $00010000; {Live scrolling when dragging thumb          }
               scrlValueLimit     = $00020000; {Value is limited by minimum/maximum limit   }
               scrlNoObscure      = $00040000; {Don't obscure scroll bar on inactive window }
               scrlColorScrollBar = $00080000; {Use color settings for this scroll bar      }

               scrlHidden         = $00100000; {Create a hidden scroll bar                  }
               scrlAutoMoveSize   = $00200000; {Auto-resize as window's size changes        }
```

NewScrollBarRect

Create a new scroll bar.

C
```
pascal void NewScrollBarRect (short ScrollBar, const Rect *Bounds, long Spec,
            Boolean EnabledFlag, short minimum, short value, short maximum);
```

Pascal
```
procedure NewScrollBarRect (ScrollBar: INTEGER; Bounds: rect; Spec: LONGINT;
            EnabledFlag: BOOLEAN; minimum, value, maximum: INTEGER);
```

NewScrollBarRect is identical to the NewScrollBar routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

NewDialogScrollBar

Create a new scroll bar in a dialog using a dialog item's co-ordinates.

C
```
pascal void NewDialogScrollBar (short ScrollBar, long Spec,
            Boolean EnabledFlag, short minimum, short value, short maximum);
```

Pascal
```
procedure NewDialogScrollBar (ScrollBar: INTEGER; Spec: LONGINT;
            EnabledFlag: BOOLEAN; minimum, value, maximum: INTEGER);
```

NewDialogScrollBar is identical to the NewScrollBar routine, except that the scroll bar is created in a dialog (a window opened with the LoadDialog routine, or one that had a dialog list attached with the LoadDialogList routine). The scroll bar's co-ordinates are obtained from the dialog item whose number matches the scroll bar number.

LoadScrollBar

Create a new scroll bar using a 'CNTL' resource.

C
```
pascal void LoadScrollBar (short ScrollBar, short ResID);
```

Pascal
```
procedure LoadScrollBar (ScrollBar, ResID: INTEGER);
```

LoadScrollBar creates a scroll bar by calling the NewScrollBar routine and supplying it with values from a 'CNTL' resource, commonly called a control template. This is a good way to create a scroll bar or scroll bar-like control that requires a color table with more elements than those supported by the SetScrollBarColors routines. Note that some controls ignore color settings, particularly those in the Appearance Manager.

ScrollBar specifies the scroll bar number (from 1 to 511) that is created in the current window. Once a scroll bar is created, it is referenced by this scroll bar number. If a scroll bar has been previously created in the current window using the same number, it is replaced with a new scroll bar as specified by the parameters in the 'CNTL' resource. If the current window doesn't belong to your application, or if no windows are open, LoadScrollBar does nothing.

ResID is the 'CNTL' resource ID number that is used to create the scroll bar. If the scroll bar has a 'cctb' color table resource, it must use the same ID number. Any resource ID number can be used, but numbers 128 or higher are safest as stated in Inside Macintosh.

When creating scroll bars using 'CNTL' resources, please note the following:
- Flag your 'CNTL' and 'cctb' resources as purgeable to save memory. Tools Plus makes a copy of their data.
- The RefCon field in the 'CNTL' resource is ignored since Tools Plus uses the control's RefCon field to store its own data.

Also see: NewScrollBar and LoadSpecScrollBar.

LoadSpecScrollBar

Create a new scroll bar using a 'CNTL' resource.

C
```
pascal void LoadSpecScrollBar (short ScrollBar, long Spec, short ResID);
```

Pascal
```
procedure LoadSpecScrollBar (ScrollBar: INTEGER; Spec: LONGINT;
           ResID: INTEGER);
```

LoadSpecScrollBar is identical to the LoadScrollBar routine, except that it requires the additional *Spec* parameter to give you control over all the appearance and behavior options offered by Tools Plus. See the NewScrollBar routine for details about the Spec parameter.

EmbedScrollBarInButton

Embed a scroll bar into a button or into the window's root control (Appearance Manager only).

C
```
pascal void EmbedScrollBarInButton (short ScrollBar, short ContainerButton);
```

Pascal
```
procedure EmbedScrollBarInButton (ScrollBar, ContainerButton: INTEGER);
```

The Appearance Manager lets you embed a control into a parent control such that when the parent is hidden or disabled, all embedded controls are similarly affected. All Tools Plus routines that load a dialog item list (LoadDialog, LoadSpecDialog, LoadDialogList, etc.) automatically embed controls at all times. EmbedScrollBarInButton lets you manually embed a scroll bar into a button, or into the window's root control. Note that the term "button" does not literally mean a button control. It means any control that is implemented as a button in Tools Plus. The most likely candidate is a Group Box control. The same applies to the term "scroll bar." If the Appearance Manager is not available, EmbedScrollBarInButton does nothing.

ScrollBar specifies the scroll bar number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, EmbedScrollBarInButton does nothing.

ContainerButton specifies the button number (from 1 to 511) into which *ScrollBar* is embedded. This control must exist in the current window, and it must be a "container" type control such as the Appearance Manager's Group Box. The scroll bar must fit entirely within the container control or EmbedScrollBarInButton does nothing. If a value of 0 is provided for a container button, *ScrollBar* is embedded into the window's root control.

Also see: EmbedScrollBarInScrollBar and SetAutoEmbed.

EmbedScrollBarInScrollBar

Embed a scroll bar into a scroll bar or into the window's root control (Appearance Manager only).

C
```
pascal void EmbedScrollBarInScrollBar (short ScrollBar,
           short ContainerScrollBar);
```

Pascal
```
procedure EmbedScrollBarInScrollBar (ScrollBar, ContainerScrollBar: INTEGER);
```

The Appearance Manager lets you embed a control into a parent control such that when the parent is hidden or disabled, all embedded controls are similarly affected. All Tools Plus routines that load a dialog item list (LoadDialog, LoadSpecDialog, LoadDialogList, etc.) automatically embed controls at all times. EmbedScrollBarInScrollBar lets you manually embed a scroll bar into a scroll bar, or into the window's root control. Note that the term "scroll bar" does not literally mean a scroll bar control. It means any control that is implemented as a scroll bar in Tools Plus. If the Appearance Manager is not available, EmbedScrollBarInScrollBar does nothing.

ScrollBar specifies the scroll bar number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, EmbedScrollBarInScrollBar does nothing.

ContainerScrollBar specifies the scroll bar number (from 1 to 511) into which *ScrollBar* is embedded. This control must exist in the current window, and it must be a "container" type control. The scroll bar must fit entirely within the container control or EmbedScrollBarInScrollBar does nothing. If a value of 0 is provided for a container scroll bar, *ScrollBar* is embedded into the window's root control.

Also see: EmbedScrollBarInButton and SetAutoEmbed.

GetFreeScrollBarNum

Get the first unused scroll bar number.

C
```
pascal short GetFreeScrollBarNum (void);
```

Pascal
```
function GetFreeScrollBarNum: INTEGER;
```

Some developers may prefer to write code that more closely resembles a traditional Macintosh application, in that creating an object returns a reference to it such as a handle or pointer. Instead of having to assign your own scroll bar number, GetFreeScrollBarNum returns the first unused (free) scroll bar number. Using this routine, you can assign an unused scroll bar number to a variable, then use that variable throughout your application without concern for the true scroll bar number.

GetFreeScrollBarNum returns the first free scroll bar number on the current window. If the current window doesn't belong to your application, if no windows are open, or if the maximum number of scroll bars has already been created on the current window (no new ones can be created), GetFreeScrollBarNum returns a value of zero (0).

ScrollBarColors

Set the colors for new scroll bars as they are created.

C
```
pascal void ScrollBarColors (const RGBColor *Frame, const RGBColor *Body,
             const RGBColor *Text, const RGBColor *Thumb,
             const RGBColor *Back);
```

Pascal
```
procedure ScrollBarColors (Frame, Body, Text, Thumb, Back: RGBColor);
```

When new scroll bars are created, by default they have a black outline and they adopt their parent window's backdrop as a background color. When you use the ScrollBarColors routine, new scroll bars adopt the colors specified in this routine (providing that the scroll bar is created with the scrlColorScrollBar option in the scroll bar's spec). This is the most efficient way to color multiple scroll bars using the same colors. Note that some controls ignore color settings, particularly those in the Appearance Manager.

Frame is the scroll bar's frame color.

Body is the scroll bar's body color.

Text is the scroll bar's text color. Apple's standard scroll bars as well as most other scroll bars do not have text. Custom CDEFs like sliders may have text as a part of a numeric scale.

Thumb is the scroll bar's thumb color. The thumb is typically outlined using the frame color.

Back is the scroll bar's background color. The standard Apple scroll bar uses this color only when the scroll bar is on an inactive window and is displayed as and empty rectangle. Tools Plus overrides the window's backdrop color with this setting so that custom CDEFs are drawn using this color as a background.

Also see: NoScrollBarColors and SetScrollBarColors.

NoScrollBarColors

Reset the colors for new scroll bars to the default.

C
```
pascal void NoScrollBarColors (void);
```

Pascal
```
procedure NoScrollBarColors;
```

When new scroll bars are created, by default they have a black outline and they adopt their parent window's backdrop as a background color. When you use the ScrollBarColors routine, new scroll bars adopt the colors specified by that routine (providing that the scroll bar is created with the scrlColorScrollBar option in the scroll bar's spec).

This routine resets the settings of the ScrollBarColors routine to the default values (black frame, white body and background). It is seldom required since you can create default scroll bars by simply excluding the scrlColorScrollBar constant from the scroll bar's spec parameter.

Also see: ScrollBarColors.

DeleteScrollBar

Delete a scroll bar.

C
```
pascal void DeleteScrollBar (short ScrollBar);
```

Pascal
```
procedure DeleteScrollBar (ScrollBar: INTEGER);
```

ScrollBar specifies the scroll bar number (from 1 to 511) that is deleted from the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, DeleteScrollBar does nothing. Use KillScrollBar if you want to delete the scroll bar without removing its image from the window.

KillScrollBar

Delete a scroll bar without affecting its image on the window.

C
```
pascal void KillScrollBar (short ScrollBar);
```

Pascal
```
procedure KillScrollBar (ScrollBar: INTEGER);
```

KillScrollBar is identical to DeleteScrollBar except that it does not remove the scroll bar's image from the window. This routine is useful for scrolling scroll bars in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetScrollBar repositions the scroll bar's co-ordinates without affecting its image (since ScrollRect has already moved it). KillScrollBar then deletes the scroll bars that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

ScrollBarDisplay

Hide or show a scroll bar.

C
```
pascal void ScrollBarDisplay (short ScrollBar, Boolean Show);
```

Pascal
```
procedure ScrollBarDisplay (ScrollBar: INTEGER; Show: BOOLEAN);
```

ScrollBarDisplay hides or shows a scroll bar on the current window. The result is seen immediately. Use discretion with this routine since scroll bars should be enabled and disabled to indicate if they are accessible by the user.

ScrollBar specifies the scroll bar number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, ScrollBarDisplay does nothing.

Show indicates if the scroll bar is being hidden or displayed. The two constants that can be used for this flag are *on* and *off*.

ScrollBarIsVisible

Determine if a scroll bar is visible.

C
```
pascal Boolean ScrollBarIsVisible (short ScrollBar);
```

Pascal
```
function ScrollBarIsVisible (ScrollBar: INTEGER): BOOLEAN;
```

ScrollBarIsVisible reports if a scroll bar (or a control that is implemented as a scroll bar) is visible on the current window, or if it is hidden.

ScrollBar specifies the scroll bar number (from 1 to 511) that is queried in the current window.

This routine's value returns *true* if the scroll bar is visible, and *false* if the scroll bar is hidden. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, ScrollBarIsVisible returns *false*. This routine takes control embedding into account, so it will return *false* if the target scroll bar is embedded and its container control is hidden.

ObscureScrollBar

Hide a scroll bar without removing its image from the window.

C
```
pascal void ObscureScrollBar (short ScrollBar);
```

Pascal
```
procedure ObscureScrollBar (ScrollBar: INTEGER);
```

ObscureScrollBar hides a scroll bar on the current window without removing its image from the window. This routine is useful for scrolling scroll bars (moving their position) in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetScrollBar repositions the scroll bar's co-ordinates without affecting its image (since ScrollRect has already moved it). ObscureScrollBar then hides the scroll bars that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

ScrollBar specifies the scroll bar number (from 1 to 511) that is hidden in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, ObscureScrollBar does nothing.

ActivateScrollBar

Activate a scroll bar to give it the keyboard focus.

C
```
pascal void ActivateScrollBar (short ScrollBar, short PartCode);
```

Pascal
```
procedure ActivateScrollBar (ScrollBar, PartCode: INTEGER);
```

ScrollBar specifies the scroll bar number (from 1 to 511) that acquires the keyboard focus in the current window. ActivateScrollBar does nothing under any of these conditions: the current window doesn't belong to your application, no windows are open, the scroll bar does not exist in the current window, the scroll bar is disabled or hidden, the scroll bar cannot accept the keyboard focus, or the Appearance Manager is not available to your application.

PartCode is the control's part number that is being activated. The part number is available either in the Appearance Manager documentation, or from the author of the custom control you are using.

Activating a scroll bar allows the user to interact with the scroll bar by typing on the keyboard. On an active window, the scroll bar acquires the keyboard focus making it the item that automatically processes keystrokes. Visually, this is indicated by having the text highlighted or with a flashing caret. Additionally, the scroll bar is encompassed with a highlighting keyboard focus band to indicate that it has the focus. Using ActivateScrollBar in an active window removes the keyboard focus from any other object that may have the focus within the same window or any other active window such as a tool bar or floating palette. This action may deactivate an active editing field.

If the scroll bar being activated is in an active window that allows access to pull-down menus, the Edit menu's "Undo" item is changed to "Can't Undo" and is disabled. The "Cut", "Copy", "Paste", "Clear" and "Select All" items are also disabled.

Your application can activate virtually any editing field or Appearance Manager control that accepts the keyboard focus. This flexibility can lead to a confusing user interface by allowing the keyboard focus to jump between active windows. A good rule to observe is to activate a single item only on a standard window (not a tool bar or a floating palette) when the window first opens. This sets up the default keyboard focus item for that window. At all other times, activate a scroll bar only in response to a user's actions.

Also see: HaveTabInFocus, TabToFocus, the doClickToFocus event, and ClickToFocus for other activating services.

GetScrollBarRect

Get a scroll bar's co-ordinates.

C
```
pascal void GetScrollBarRect (short ScrollBar, Rect *Bounds);
```

Pascal
```
procedure GetScrollBarRect (ScrollBar: INTEGER; var Bounds: RECT);
```

ScrollBar specifies the scroll bar number (from 1 to 511) that is queried in the current window.

Bounds returns the scroll bar's bounding rectangle specified in the window's local co-ordinates. These co-ordinates match those used to create the scroll bar. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, *Bounds* returns with all co-ordinates set to zero (0).

EnableScrollBar

Enable or disable a scroll bar.

C
```
pascal void EnableScrollBar (short ScrollBar, Boolean EnabledFlag);
```

Pascal
```
procedure EnableScrollBar (ScrollBar: INTEGER; EnabledFlag: BOOLEAN);
```

ScrollBar specifies the affected scroll bar number (from 1 to 511) in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, EnableScrollBar does nothing.

EnabledFlag specifies if the scroll bar is enabled or disabled when the window is active. When a scroll bar is disabled, the thumb and toned "page up" and "page down" regions disappear and the scroll bar cannot be selected by the user. The two constants that can be used for this purpose are *enabled* and *disabled*.

All scroll bars automatically become disabled if the window containing them becomes inactive. When the window is activated, the scroll bars assume their normal state as set by the EnableScrollBar routine.

```
CONST                         {Scroll bar state                         }
               enabled  = true;   {enable the scroll bar                    }
               disabled = false;  {disable the scroll bar                   }
```

ScrollBarIsEnabled

Determine if a scroll bar is enabled or disabled.

C
```
pascal Boolean ScrollBarIsEnabled (short ScrollBar);
```

Pascal
```
function ScrollBarIsEnabled (ScrollBar: INTEGER): BOOLEAN;
```

ScrollBar specifies the scroll bar number (from 1 to 511) that is queried in the current window.

The routine's value returns *true* if the scroll bar is enabled, and *false* if the scroll bar is disabled. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, ScrollBarIsEnabled returns *false*. ScrollBarIsEnabled returns the scroll bar's enabled state as it is currently displayed, so if the scroll bar's window is inactive and has temporarily disabled the scroll bar, ScrollBarIsEnabled returns *false*.

GetScrollBarMin

Get a scroll bar's minimum value limit.

C
```
pascal short GetScrollBarMin (short ScrollBar);
```

Pascal
```
function GetScrollBarMin (ScrollBar: INTEGER): INTEGER;
```

ScrollBar specifies the affected scroll bar number (from 1 to 511) in the current window.

GetScrollBarMin returns a scroll bar's minimum value limit. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, GetScrollBarMin will return a value of zero (0).

SetScrollBarMin

Set a scroll bar's minimum value limit.

C
```
pascal void SetScrollBarMin (short ScrollBar, short minimum);
```

Pascal
```
procedure SetScrollBarMin (ScrollBar, minimum: INTEGER);
```

ScrollBar specifies the affected scroll bar number (from 1 to 511) in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, SetScrollBarMin does nothing.

Minimum specifies the scroll bar's new minimum value limit. The scroll bar's current value and maximum limit are automatically adjusted (if necessary) to be consistent with the new minimum limit.

GetScrollBarMax

Get a scroll bar's maximum value limit.

C
```
pascal short GetScrollBarMax (short ScrollBar);
```

Pascal
```
function GetScrollBarMax (ScrollBar: INTEGER): INTEGER;
```

ScrollBar specifies the affected scroll bar number (from 1 to 511) in the current window.

GetScrollBarMax returns a scroll bar's maximum value limit. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, GetScrollBarMax will return a value of zero (0).

SetScrollBarMax

Set a scroll bar's maximum value limit.

C
```
pascal void SetScrollBarMax (short ScrollBar, short maximum);
```

Pascal
```
procedure SetScrollBarMax (ScrollBar, maximum: INTEGER);
```

ScrollBar specifies the affected scroll bar number (from 1 to 511) in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, SetScrollBarMax does nothing.

Maximum specifies the scroll bar's new maximum value limit. The scroll bar's current value and minimum limit are automatically adjusted (if necessary) to be consistent with the new maximum limit.

GetScrollBarVal

Get a scroll bar's current value.

C
```
pascal short GetScrollBarVal (short ScrollBar);
```

Pascal
```
function GetScrollBarVal (ScrollBar: INTEGER): INTEGER;
```

ScrollBar specifies the affected scroll bar number (from 1 to 511) in the current window.

GetScrollBarVal returns a scroll bar's current value. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, GetScrollBarVal will return a value of zero (0).

SetScrollBarVal

Set a scroll bar's current value.

C
```
pascal void SetScrollBarVal (short ScrollBar, short value);
```

Pascal
```
procedure SetScrollBarVal (ScrollBar, Value: INTEGER);
```

ScrollBar specifies the affected scroll bar number (from 1 to 511) in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, SetScrollBarVal does nothing.

Value specifies the scroll bar's new current value. The scroll bar's minimum and maximum limits are automatically adjusted (if necessary) to be consistent with the new current value.

MoveScrollBar

Move a scroll bar to a new location on the window.

C
```
pascal void MoveScrollBar (short ScrollBar, short toHoriz, short toVert);
```

Pascal
```
procedure MoveScrollBar (ScrollBar, toHoriz, toVert: INTEGER);
```

ScrollBar specifies the scroll bar number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *ScrollBar* specifies a scroll bar that does not exist, MoveScrollBar does nothing. The change is seen immediately providing that the scroll bar is not hidden. The scroll bar's width and height are not changed.

ToHoriz is the new horizontal co-ordinate at which the left side of the scroll bar appears.

ToVert is the new vertical co-ordinate at which the top of the scroll bar appears.

Also see: SizeScrollBar and MoveSizeScrollBar.

OffsetScrollBar

Change a scroll bar's co-ordinates without affecting its image on the window.

C
```
pascal void OffsetScrollBar (short ScrollBar,
           short distHoriz, short distVert);
```

Pascal
```
procedure OffsetScrollBar (ScrollBar, distHoriz, distVert: INTEGER);
```

When you scroll an area that contains scroll bars, first use ScrollRect to scroll the pixel image containing the affected objects in the window. OffsetScrollBar is used to offset a scroll bar's co-ordinates without altering its image (since ScrollRect has already done so). At this point, the scroll bar's co-ordinates match the scrolled image of the scroll bar. ObscureScrollBar or KillScrollBar can be used to hide or delete scroll bars that are scrolled out of view.

ScrollBar specifies the scroll bar number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *ScrollBar* specifies a scroll bar that does not exist, OffsetScrollBar does nothing.

DistHoriz and *distVert* specify the horizontal and vertical amount by which the scroll bar's co-ordinates are offset. Positive numbers are right and down. The scroll bar's co-ordinates are updated but no change is seen.

SizeScrollBar

Change a scroll bar's size.

C
```
pascal void SizeScrollBar (short ScrollBar, short width, short height);
```

Pascal
```
procedure SizeScrollBar (ScrollBar, width, height: INTEGER);
```

SizeScrollBar changes a scroll bar's width and/or height without altering the scroll bar's top or left co-ordinate. The change is seen immediately providing that the scroll bar is not hidden.

ScrollBar specifies the scroll bar number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *ScrollBar* specifies a scroll bar that does not exist, SizeScrollBar does nothing.

Width and *height* specify the scroll bar's new width and height in pixels. If either parameter is less than 1, SizeScrollBar does nothing.

Also see: MoveScrollBar and MoveSizeScrollBar.

MoveSizeScrollBar

Change a scroll bar's co-ordinates.

C
```
pascal void MoveSizeScrollBar (short ScrollBar,
           short left, short top, short right, short bottom);
```

Pascal
```
procedure MoveSizeScrollBar (ScrollBar, left, top, right, bottom: INTEGER);
```

MoveSizeScrollBar changes any of the scroll bar's four co-ordinates. The change is seen immediately providing that the scroll bar is not hidden. This routine combines the functions of MoveScrollBar and SizeScrollBar.

ScrollBar specifies the scroll bar number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *ScrollBar* specifies a scroll bar that does not exist, MoveSizeScrollBar does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in local co-ordinates that determines the scroll bar's size and location in the window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). If these parameters specify an empty rectangle, MoveSizeScrollBar does nothing.

Also see: GetScrollBarRect.

MoveSizeScrollBarRect

Change a scroll bar's co-ordinates.

C
```
pascal void MoveSizeScrollBarRect (short ScrollBar, const Rect *Bounds);
```

Pascal
```
procedure MoveSizeScrollBarRect (ScrollBar: INTEGER; Bounds: RECT);
```

MoveSizeScrollBarRect is identical to the MoveSizeScrollBar routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

AutoMoveSizeScrollBar

Specify how a scroll bar is automatically moved and/or resized as its window's size is changed.

C
```
pascal void AutoMoveSizeScrollBar (short ScrollBar,
              Boolean left, Boolean top, Boolean right, Boolean bottom);
```

Pascal
```
procedure AutoMoveSizeScrollBar (ScrollBar: INTEGER;
              left, top, right, bottom: BOOLEAN);
```

ScrollBar specifies the scroll bar number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *ScrollBar* specifies a scroll bar that does not exist, AutoMoveSizeScrollBar does nothing.

The *left*, *top*, *right* and *bottom* parameters specify if that side of the scroll bar is automatically adjusted when the window's size changes. These setting are applied to the scroll bar and are used the next time the window's size changes:

left	Does the scroll bar's left side track the window's right edge?
top	Does the scroll bar's top track the window's bottom edge?
right	Does the scroll bar's right side track the window's right edge?
bottom	Does the scroll bar's bottom track the window's bottom edge?

You can think of each *false* value as locking that side of the scroll bar to a fixed co-ordinate regardless of the window's size (this is the default). Each *true* value establishes a fixed distance between that side of the scroll bar and the window's edge. For example, setting only *left* and *right* to *true* makes the scroll bar move horizontally as the window widens and narrows, but the scroll bar does not move vertically when the window's height changes.

If you are setting these values identically for a group of objects, use AutoMoveSize to define the settings then add the appropriate *x*AutoMoveSize constant (such as scrlAutoMoveSize for scroll bars) to the objects' spec as they are created. The objects will adopt the settings specified by the AutoMoveSize routine.

☞ **Warning**: Make sure that you resize objects in a way that makes sense. Don't allow a window to shrink down to a size where objects become unusable or disappear altogether.

SetScrollBarFontSettings

Set a scroll bar's font, size and style settings.

C
```
pascal void SetScrollBarFontSettings (short ScrollBar,
           short theFont, short theSize, Style theStyle);
```

Pascal
```
procedure SetScrollBarFontSettings (ScrollBar: INTEGER;
           theFont: INTEGER; theSize: INTEGER; theStyle: Style);
```

ScrollBar specifies the scroll bar number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if the scroll bar does not exist, SetScrollBarFontSettings does nothing. Otherwise, the change is seen immediately.

TheFont specifies the scroll bar's new font. The default is Chicago, which is represented by the systemFont constant.

TheSize specifies the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle specifies the scroll bar's new style. Special character constants defined by the Font Manager are bold, italic, underline and shadow. C programmers use the Font Manager's constants to specify a composite style, such as SetScrollBarFontSettings(1, geneva, 9, bold + outline) for bold and outlined, or for plain text, SetScrollBarFontSettings(1, geneva, 9, 0). Pascal programmers use the Font Manager's constants to specify a style set, such as SetScrollBarFontSettings(1, geneva, 9, [bold, outline]) for bold and outlined, or for plain text, SetScrollBarFontSettings(1, geneva, 9, [ ]).

A scroll bar's font settings are set when a scroll bar is created, so this routine is not normally used by many applications.

Note: This routine works on Appearance Manager savvy controls (ones that were written to take advantage of the Appearance Manager's extended features) that accept the "set font" command. This routine also works on classic controls (those that were not written to take advantage of the Appearance Manager, including Apple's controls in System 6 and System 7, and SuperCDEFs) as well as third party controls that observe two rules:

1. The high bit of the variant code (8) indicates that the control uses the window's font.
2. All parameters that are used to *create* the control, specifically the control's rectangle, title, visible state, initial value, minimum limit, maximum limit, and reference constant, all have no special significance.

You may experience issues with third-party CDEFs that place special significance on the initial settings that are used to create the control. For example, you may experience issues if you use a third-party slider CDEF that initially uses the "current value" setting to determine which pictures it should display for the slider's parts, then it later changes the control's "current value" setting to reflect the slider's real value. Your only solutions are: (1) create the control with the high bit of the variant code set on (+8 or bUseWFont), or (2) use another CDEF that does not place special significance on initial settings when the control is created, or (3) do not use the SetScrollBarFontSettings routine on that control.

GetScrollBarFontSettings

Get a scroll bar's font, size and style settings.

C
```
pascal void GetScrollBarFontSettings (short ScrollBar,
           short *theFont, short *theSize, Style *theStyle);
```

Pascal
```
procedure GetScrollBarFontSettings (ScrollBar: INTEGER;
           var theFont: INTEGER; var theSize: INTEGER; var theStyle: Style);
```

ScrollBar specifies the scroll bar number (from 1 to 511) in the current window whose font settings are being retrieved. If the current window doesn't belong to your application, if no windows are open, or if *ScrollBar* specifies a scroll bar that does not exist, GetScrollBarFontSettings returns default values.

TheFont is the scroll bar's font number. The default is 0 which is represented by the systemFont constant.

TheSize is the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle is the field's font style. The default is plain text, which is represented by 0 in C and [ ] in Pascal.

SetScrollBarColors

Set a scroll bar's colors.

C
```
pascal void SetScrollBarColors (short ScrollBar, const RGBColor *Frame,
            const RGBColor *Body, const RGBColor *Text, const RGBColor *Thumb,
            const RGBColor *Back);
```

Pascal
```
procedure SetScrollBarColors (ScrollBar: INTEGER;
            Frame, Body, Text, Thumb, Back: RGBColor);
```

ScrollBar specifies the scroll bar number (from 1 to 511) in the current window whose colors are being set. If the current window doesn't belong to your application, or if no windows are open, SetScrollBarColors does nothing. Also, if *ScrollBar* specifies a scroll bar that does not exist, SetScrollBarColors does nothing. The change is seen immediately, regardless if the scroll bar was originally created with the scrlColorScrollBar option or not. Note that some controls ignore color settings, particularly those in the Appearance Manager.

Frame is the scroll bar's frame color.

Body is the scroll bar's body color.

Text is the scroll bar's text color. Apple's standard scroll bars as well as most other scroll bars do not have text. Custom CDEFs like sliders may have text as a part of a numeric scale.

Thumb is the scroll bar's thumb color. The thumb is typically outlined using the frame color.

Back is the scroll bar's background color. The standard Apple scroll bar uses this color only when the scroll bar is on an inactive window and is displayed as and empty rectangle. Tools Plus overrides the window's backdrop color with this setting so that custom CDEFs are drawn using this color as a background.

Also see: ScrollBarColors and GetScrollBarColors.

GetScrollBarColors

Get a scroll bar's colors.

C
```
pascal void GetScrollBarColors (short ScrollBar, RGBColor *Frame,
            RGBColor *Body, RGBColor *Text, RGBColor *Thumb, RGBColor *Back);
```

Pascal
```
procedure GetScrollBarColors (ScrollBar: INTEGER; var Frame: RGBColor;
            var Body: RGBColor; var Text: RGBColor; var Thumb: RGBColor;
            var Back: RGBColor);
```

ScrollBar specifies the scroll bar number (from 1 to 511) in the current window whose colors are being retrieved. If the current window doesn't belong to your application, or if no windows are open, or if *ScrollBar* specifies a scroll bar that does not exist, GetScrollBarColors returns default color values.

Frame is the scroll bar's frame color.

Body is the scroll bar's body color.

Text is the scroll bar's text color. Apple's standard scroll bars as well as most other scroll bars do not have text.

Custom CDEFs like sliders may have text as a part of a numeric scale.

Thumb is the scroll bar's thumb color. The thumb is typically outlined using the frame color.

Back is the scroll bar's background color. The standard Apple scroll bar uses this color only when the scroll bar is on an inactive window and is displayed as and empty rectangle. Tools Plus overrides the window's backdrop color with this setting so that custom CDEFs are drawn using this color as a background.

Also see: ScrollBarColors and SetScrollBarColors.

ScrollBarLineTime

Set the line scrolling speed for new scroll bars.

C
```
pascal void ScrollBarLineTime (short Ticks);
```

Pascal
```
procedure ScrollBarLineTime (Ticks: INTEGER);
```

Subsequently created scroll bars adopt the specified speed when their up arrow or down arrow is used. Use this routine when you want a number of scroll bars (or all of them) to scroll at the same rate.

Ticks specifies the number of ticks (1/60 of a second) that elapse between doScrollBar events when the user clicks and holds the up arrow or down arrow. A value of zero (0) will generate doScrollBar events as rapidly as your application can handle them (no throttling), whereas the maximum value of 60 causes the scroll bar to wait one second between each doScrollBar event.

Also see: ScrollBarPageTime, SetScrollBarLineTime and SetScrollBarPageTime.

ScrollBarPageTime

Set the page scrolling speed for new scroll bars.

C
```
pascal void ScrollBarPageTime (short Ticks);
```

Pascal
```
procedure ScrollBarPageTime (Ticks: INTEGER);
```

Subsequently created scroll bars adopt the specified speed when their page up or page down region is used. Use this routine when you want a number of scroll bars (or all of them) to scroll at the same rate.

Ticks specifies the number of ticks (1/60 of a second) that elapse between doScrollBar events when the user clicks and holds the page up or page down region. A value of zero (0) will generate doScrollBar events as rapidly as your application can handle them (no throttling), whereas the maximum value of 60 causes the scroll bar to wait one second between each doScrollBar event.

Also see: ScrollBarLineTime, SetScrollBarLineTime and SetScrollBarPageTime.

SetScrollBarLineTime

Set a scroll bar's line scrolling speed.

C
```
pascal void SetScrollBarLineTime (short ScrollBar, short Ticks);
```

Pascal
```
procedure SetScrollBarLineTime (ScrollBar: INTEGER; Ticks: INTEGER);
```

ScrollBar specifies the affected scroll bar number (from 1 to 511) in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, SetScrollBarLineTime does nothing.

Ticks specifies the number of ticks (1/60 of a second) that elapse between doScrollBar events when the user clicks and holds the up arrow or down arrow. A value of zero (0) will generate doScrollBar events as rapidly as your application can handle them (no throttling), whereas the maximum value of 60 causes the scroll bar to wait one second between each doScrollBar event.

Also see: ScrollBarLineTime, ScrollBarPageTime and SetScrollBarPageTime.

SetScrollBarPageTime

Set a scroll bar's page scrolling speed.

C
```
pascal void SetScrollBarPageTime (short ScrollBar, short Ticks);
```

Pascal
```
procedure SetScrollBarPageTime (ScrollBar: INTEGER; Ticks: INTEGER);
```

ScrollBar specifies the affected scroll bar number (from 1 to 511) in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, SetScrollBarPageTime does nothing.

Ticks specifies the number of ticks (1/60 of a second) that elapse between doScrollBar events when the user clicks and holds the page up or page down region. A value of zero (0) will generate doScrollBar events as rapidly as your application can handle them (no throttling), whereas the maximum value of 60 causes the scroll bar to wait one second between each doScrollBar event.

Also see: ScrollBarLineTime, ScrollBarPageTime and SetScrollBarLineTime.

SetScrollBarAction

Set a scroll bar's action routine.

C
```
pascal void SetScrollBarAction (short ScrollBar,
            ScrollBarActionUPP ActionProc);
```

Pascal
```
procedure SetScrollBarAction (ScrollBar: INTEGER;
            ActionProc: ScrollBarActionUPP);
```

SetScrollBarAction sets a routine that is called repeatedly when a scroll bar is tracked. This occurs as long as the user holds the mouse button down in the scroll bar, regardless if the cursor wanders off the scroll bar. Each scroll bar can have its own action routine or several scroll bars can share the same routine, even if they are on different windows.

ScrollBar specifies the affected scroll bar number (from 1 to 511) in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the scroll bar does not exist in the current window, SetScrollBarAction does nothing.

ActionProc is the routine that is called repeatedly while the scroll bar is being tracked.

The ScrollBarActionUPP type is a Universal Procedure Pointer used for consistency across all interfaces (C/C++ and Pascal using the original Apple interfaces or the newer universal interfaces required for PowerMacs). In 680x0 applications, the ScrollBarActionUPP is nothing more than a ProcPtr, or a pointer to a Pascal routine. This is how you set an action routine in C/C++:

```
SetScrollBarAction(5, myActionProc);
```

In Pascal, a similar statement is used except the "@" symbol indicates the address of a routine which is the same thing as a pointer to a routine:

```
SetScrollBarAction(5, @myActionProc);
```

In PowerMac applications, the ScrollBarActionUPP is a pointer to a structure that is allocated using the NewScrollBarActionProc routine. If you are writing a PowerMac application, or if your source code will compile to both 680x0 and PowerMac-native code, you will need to use the new universal headers (or universal interfaces for Pascal) and do the following to ensure 680x0 and PowerPC compatibility:

1. Create a global variable for each action routine you will use throughout your application. If you are using the same action routine for several scroll bars, all the scroll bars can share a single global variable. Declare the variable as a ScrollBarActionUPP type. In 680x0 applications, this variable will be used as a pointer to an action routine. In PowerMac applications, it will be used as a pointer to a universal procedure structure. In this example, define a global variable named myActionUPP of type ScrollBarActionUPP.
2. Populate myActionUPP so that it points to your scroll bar's action routine. In this example, the action routine is named myScrollAction. In C/C++, the code looks like this:

   ```
   myActionUPP = NewScrollBarActionProc(myScrollAction);
   ```

   In Pascal, the code is identical except the "@" symbol indicated the address of a routine:

   ```
   myActionUPP := NewScrollBarActionProc(@myScrollAction);
   ```

   Do this very early in your application because you are creating a non-relocatable structure and allocating it early will prevent memory fragmentation.
3. After you create your scroll bar, you can install the action routine into the scroll bar with the following code. This example assumes the action routine is being installed into scroll bar number 5 on the current window:

   ```
   SetScrollBarAction(5, myActionUPP);
   ```

The action routine is written as a Pascal procedure that has no parameters. Here is an example of how your routine should be written:

C

```
pascal void myScrollAction (void)
  {
  // Your code goes here
  }
```

Pascal

```
procedure myScrollAction;
  begin
    {Your code goes here}
  end;
```

If you want to deallocate the UPP for scroll bar's action routine in a PowerMac application or plug-in, use the DisposeRoutineDiscriptor routine. PowerMac plug-ins will certainly want to do this as part of their quitting logic along with calling DeinitToolsPlus.

Your action routine will likely need to know some information about the scroll bar that called the action routine and how the user is interacting with the scroll bar. See the GetScrollBarActionInfo routine to obtain this information.

Also see: GetScrollBarActionInfo.

GetScrollBarActionInfo

Get info about the caller of a scroll bar's action routine.

C
```
pascal void GetScrollBarActionInfo (short *Window,
            short *ScrollBar, short *Part, Boolean *InPart);
```

Pascal
```
procedure GetScrollBarActionInfo (var Window: INTEGER;
            var ScrollBar: INTEGER; var Part: INTEGER; var InPart: BOOLEAN);
```

This routine can be used in a scroll bar's action routine to learn about the scroll bar that has called the routine. Its values will be valid only when called from inside a scroll bar action routine.

Window is the window number containing the scroll bar that the user is using.

ScrollBar is the scroll bar number that is being used.

Part is the part code that corresponds to the region in the scroll bar where the mouse when down. The values for standard scroll bars and CDEFs that are written to behave like scroll bars are available through the constants inUpButton, inDownButton, inPageUp, inPageDown and inThumb.

InPart tells your action routine if the user's mouse is still in the region that is specified by the *Part* parameter. If, for example, the user starts by pressing the mouse on the up button, *Part* will always be set to inUpButton each time your action routine is called but *InPart* is set to *true* only when the cursor is on the up button. When *InPart* returns *false*, your action routine should behave as though the scroll bar is not being used and not perform any scrolling.

```
CONST                          {Scroll Bar parts                      }
          inUpButton   = 20;   {up arrow of a scroll bar              }
          inDownButton = 21;   {down arrow of a scroll bar            }
          inPageUp     = 22;   {"page up" region of a scroll bar      }
          inPageDown   = 23;   {"page down" region of a scroll bar    }
          inThumb      = 129;  {thumb of a scroll bar                 }
```

GetScrollBarHandle

Get a handle to a scroll bar's control record.

C
```
pascal ControlHandle GetScrollBarHandle (short ScrollBar);
```

Pascal
```
function GetScrollBarHandle (ScrollBar: INTEGER): ControlHandle;
```

This routine returns a standard ControlHandle to a scroll bar that was created by a Tools Plus routine. You should never need to use this routine. It is provided for advanced programmers who may have specialized needs. Always use Tools Plus routines to create and manipulate scroll bars.

ScrollBar specifies the scroll bar number (from 1 to 511) in the current window whose handle is being retrieved. If the current window doesn't belong to your application, or if no windows are open, or if *ScrollBar* specifies a scroll bar that does not exist, GetScrollBarHandle returns nil.

☞ **Warning**: If you need to lock the handle or change its attributes, do so temporarily then restore the original settings before using any Tools Plus routines. If you alter this handle or any data that is made accessible by this handle, you do so at your own risk. The only exception is the control's reference constant (contrlRfCon field) which can safely be set using the toolbox's SetControlReference routine, and retrieved using the toolbox GetControlReference routine.

9 Editing Fields

Editing fields are supported by Tools Plus windows with some notable enhancements over the Macintosh's standard TextEdit fields. Editing fields (or simply "fields") are created on the current window with the NewField routine. Each field is referenced by a unique field number that can be from 1 to 511. Fields that are not implemented using the Appearance Manager's Edit Text control or Static Control can be numbered from 1 to 32767. This number is specified when the field is created, and refers to the specific field until that field is deleted. Note that the field number is related to its associated window. This means that two different windows can each have a field numbered "1" without interfering with each other. Whenever any field event occurs, such as the user clicking on a field, Tools Plus calls your event handler routine and reports the field number as well as the window number to which the field belongs.

Fields can edit and store as much as 32K of text. Tools Plus also accommodates Pascal's 255 characters strings (Str255). Your application can limit the number of characters that can be typed by the user, as well as the number of characters that are eventually stored by the field. Details are provided later in this chapter. Both C and Pascal programmers can make use of both C and Pascal strings within an application.

Tools Plus fields offer an option called a "static text" field, that being a field that cannot be edited by the user. It is used to display information like a title or caption, and can be created as part of a dialog resource, or dynamically just like regular fields. Additional information about creating and maintaining fields in dialogs can be found in the Windows chapter, specifically in the LoadDialog routine.

Tools Plus also supports a "read only" field that looks like an editing field, but cannot be changed by the user. With this option, the user can select text in the field and copy text from the field, but the user cannot change the text in the field.

The Field's String

When a field is first created, your application initiates a permanent association between the editing field and its related text by providing a handle that points to the *field's string*. The handle can point to either a Pascal string (up to 255 characters long plus a length-byte prefix), or to a C string (up to 32767 characters long plus a null termination byte). Each field must have its own string handle that is valid while the field exists. You must either allocate memory for the handle and set it to a default string value before using it in the NewField routine, or let NewField allocate the string for you. Tools Plus's NewStrHandle routine is perfect for allocating a handle of a specific size and initializing its string to zero characters.

Note: Your application must allocate memory for a text handle when creating a new field. The NewStrHandle routine is best suited for this. If your editing fields contain random characters, it is a sure sign that you provided an invalid handle when creating the field.

Dynamic String Handles

A field can optionally store its string more efficiently by resizing its string handle on an ongoing basis to accommodate only the number of valid characters currently stored in the string as opposed to the *maximum* number of characters the string may contain. This feature is especially useful if you let NewField dynamically allocate a string handle for you. You should consider turning this feature on for all fields if your application has numerous editing fields. To make a field store its string handle more efficiently, call DynamicFieldHandles(true) and all subsequently created fields will automatically resize their associated text handle. You can also do this on a field-by-field basis when a field is being created by including the teResizeHdl constant in the field's specification.

The one thing you must pay attention to is that a dynamic handle *may* be smaller than you expect, and that you can accidentally damage your application's heap if you write more data into the handle than the handle can physically accommodate. To prevent this, make a habit of using PasteIntoField to change a field's text instead of writing to the handle directly.

The Active Field

Text editing can occur in only *one* field at a time even if multiple applications are running concurrently or multiple windows are open in an a variety of applications. The *application's active field* is the field in which the user is working when an application is active. The active field indicates it is active by containing either a flashing insertion point (called a "caret"), or several highlighted (selected) characters as seen below.

an illustration   an illustration

A field can be activated by using the ActivateField routine. When a field is first activated by your application, an insertion point can be placed at the beginning or at the end of the field's text, or the field's entire text can be selected. In most cases, the field's entire text is selected. When a window becomes inactive, the flashing insertion point disappears and selected text becomes deselected. When the window is activated again, the insertion point or selection reappears. A field can be deactivated by either calling the DeactivateField routine, or by activating another field.

Each window can have its own editing field that becomes the application's active editing field when the window is active. This is called the *window's active field.* For example, field 3 could be active in the "add customer" window, and when the "add shipping address" window is active, field 8 could be active in that window. This concept is simple as long as only one window is active at a time.

When your application uses a tool bar and/or floating palettes, multiple windows are active simultaneously: the tool bar, all floating palettes, and the frontmost standard window. Tools Plus manages the additional functionality required to make editing fields work on all windows in your application, including the tool bar and floating palettes. It ensures that only one field is active at a time, and that it is the appropriate one.

Editing Field Window

The Editing Field Window is the *one* window in your application containing the active editing field. The field either has a flashing insertion point, or its selected text is highlighted. Tools Plus automatically keeps track of which window contains your application's active editing field.

If your application does not use a tool bar or floating palettes, this window is the active window providing it has an editing field. If your application uses a tool bar and/or floating palettes, potentially any active window (tool bar, any floating palette, or the active standard window) can contain the active editing field.

Activating a Field and Editing Text

Your application can activate a field by using the ActivateField routine to specify the field that is active when its parent window is active. This is usually done to specify the default field when the window is first opened. When a field is activated, a *copy* of its associated text (the field's string) is made. It is the copy that is edited by the user while the field's original string is left untouched. The copy being edited by the user is called the *edited text*. The edited text can be inspected by your application by using GetEditString which returns a copy of the edited text, or GetEditHandle which returns a handle to the edited text. The edited text is saved as the field's string by calling the SaveFieldString routine.

When a field is deactivated, the text edited by the user is destroyed without being saved and the field's string is redrawn. This is done in case your application determines that the edited text should *not* be saved. If you saved the edited text with SaveFieldString, no apparent change is seen. Tools Plus has options to automatically save edited text, as explained later in the chapter under "Clicking and Tabbing."

Editing fields are fully integrated with all Tools Plus windows. As a result, Tools Plus prevents fields from being mysteriously activated or deactivated as windows are opened, closed, hidden, displayed, activated and deactivated. The user's interaction with editing fields is completely intuitive.

Length Limited Fields

In some applications, it may be necessary to limit the length of edited text to a specific number of characters. This can be done by calling FieldLengthLimit(true). All subsequently created fields will be length limited. You can also length limit a field as it is being created by including the teLengthLimit constant in the field's specification.

If a field is length limited, the user can type characters until the field's maximum length is reached. After this point, a beep is heard when the user types any keys. For example, if you create a 30 character string handle and length limit the field, the user can type only 30 characters into that field. This also applies when pasting text into a length limited field.

Length limited fields look their best if you use a monospaced font like Monaco 9 in a single-line editing field that is long enough to hold the maximum number of characters without scrolling.

Clicking and Tabbing

The task of moving between active fields, either by tabbing or clicking, is accomplished in one of two very different ways:

(1) When your application is initialized, you can use the initAutoFocusChanges option to automatically let the user tab to the next/previous field, or to activate an inactive (but enabled) field by clicking on it. This option makes for much simpler coding, but it does not give your application the opportunity to validate fields' contents on a field-by-field basis. Instead, your application can edit the fields' contents in a batch when the user clicks the OK button to process the entire window.

(2) If you need to edit your fields on a field-by-field basis, meaning the user may get an error message as he tries to tab to another field thereby forcing him to correct his error before moving on, then you should not use the initAutoFocusChanges option when initializing Tools Plus. Instead, your application will be informed when the user wants to move to another field by way of events. This lets you edit the field before letting the user activate another field.

If you chose the second option to allow editing on a field-by-field basis, Tools Plus reports a doClickToFocus event to your event handler routine when the user clicks in an inactive field. When this happens, your event handler will likely call GetEditString (or GetEditHandle) to obtain a copy of the edited text and check the string for errors. If an error is detected, display an appropriate alert and ignore the doClickToFocus event. If no error is detected, call the SaveFieldString routine to save the edited text in the field's string, then call the ClickToFocus routine to activate the required field and places the insertion point at the appropriate place. The following code shows you how to respond to a doClickToFocus event in your application's event handler routine:

C

```
case doClickToFocus:
  GetEditString(&theString);
  // your code to validate string for errors
  if (errorInString)
    // Show alert
  else {
    SaveFieldString();
    ClickToFocus();
  }
  break;
```

Pascal

```
doClickToFocus:
  begin
    GetEditString(theString);
    {your code to validate string for errors}
    if errorInString then
      {Show alert}
    else
      begin
        SaveFieldString;
        ClickToFocus;
      end;
  end;
```

If your application allows field validation on a field-by-field basis, it will also be interested in validating the active field when the user tabs out of it. When Tools Plus reports a doKeyDown or doAutoKey event, your event handler can call HaveTabInFocus to determine if a tab or shift-tab was typed by the user in an editing field. If HaveTabInFocus returns *true*, you will likely validate the edited text as described in the previous paragraph, then activate the next field if the tab was pressed or the previous field if shift-tab is pressed. A simpler alternative is to use TabToFocus which takes care of tabbing to the correct field. The Event Management chapter details key and mouse events. The following code illustrates how to tab to another field:

C

```
case doKeyDown: case doAutoKey:
  if (HaveTabInFocus()) {
    GetEditString(&theString);
    // your code to validate string for errors
    if (errorInString)
      // Show alert
    else {
      SaveFieldString();
      TabToFocus();
    }
  break;
```

Pascal

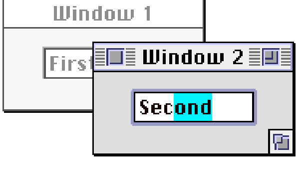
```
doKeyDown, doAutoKey:
  If HaveTabInFocus then
    begin
      GetEditString(theString);
      {your code to validate string for errors}
      if errorInString then
        {Show alert}
      else
        begin
          SaveFieldString;
          TabToFocus;
        end;
    end;
```

Keyboard Focus on Tool Bars and Floating Palettes

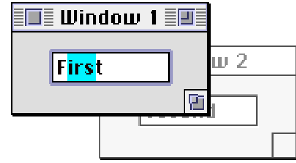
If your application includes a tool bar and/or a floating palette, and any of these windows contains an editing field or any other object that can assume the keyboard focus (such as a list box or a clock control), then you may have to write your application with a few special considerations. The easiest way to address these special cases is to initialize your application with the initAutoFocusChanges option. That way, Tools Plus will automatically save any edited text as and when required, and it will move the keyboard focus as the user tabs and clicks to other user interface elements. Without the initAutoFocusChanges option, you may have to account for some special cases as described in this section.

Normally, a Macintosh application has only one active window at a time, so when a window with the keyboard focus is deactivated, that window remembers all the information about its own focus while the application's keyboard focus is temporarily transferred to a newly activated window. When the user activates the original window again, the keyboard focus returns to that window exactly as it was left. When using only standard window (no tool bars or floating palettes), each window's keyboard focus is specific to that window and it never conflicts or interacts with the focus on any other window. This is what Macintosh users and developers are used to.

User working in Window 1



User activates window 2 by clicking on it.
Keyboard focus moves to window 2.



User activates window 1 by clicking on it.
Keyboard focus returns to window 1 exactly as it was left.

When a tool bar or a floating palette with a keyboard focus item is introduced to your application, your application then has two or more active windows, only one of which can have the keyboard focus at a time. Two potentially problematic situations can arise:

- When your application gets a doClickToFocus event, it *may* indicate that the user is clicking on an object in another active window. An example of this is when the user is working on a standard window, then clicks in an editing field on a floating palette. This means you may have to save the edited text on one window before allowing the user to move to another by calling ClickToFocus.
- Once the keyboard focus has been placed on a tool bar or floating palette, it will not automatically move to a standard window that is being activated. This may impact your application because the user could do something like this:
    - User edits text in a field in window 1
    - User clicks on window 2 to activate it (window 1 and its field with edited text are deactivated)
    - User clicks on a floating palette and enters text in a field (focus now stays on the floating palette)
    - User clicks on window 1. Window 1 activates but the editing field is deactivated because the floating palette, an active window, is still holding onto the keyboard focus.

The actions required by your application are as follows:

- When your application gets a doClickToFocus event, save the edited text on the window that has the active field (the EditFldWindowNumber routine tells you which one it is). This must be done before you allow the focus to move to the other window by calling ClickToFocus.
- Before a window is activated, check to see if the keyboard focus or active editing field is on a tool bar or floating palette. If so, save the edited text on the window that is being activated before you activate it.

Alignment of Text in a Field

Each field can be either left aligned, right aligned, or centered. It cannot, however, be fully justified (i.e., margin to margin). Each field can have its own font, font size, and font styling. Variations of font, size and style within a single field are not possible. Static text fields can only be left aligned.

Fonts

All fields default to using the Chicago 12pt font. When a field is created, it can optionally adopt and remember the window's current font, size and style settings (as set by the TextFont, TextSize, and TextFace routines) by including the teUseWFont option in the spec parameter. The window's settings can then be changed without affecting the field. Unlike regular fields, Tools Plus fields can each have a different font. You can use the GetFieldFontSettings and SetFieldFontSettings routines to get and set the field's font, size and style settings.

Colors

By default, a field is displayed using black text on a white background. You can change this by adding the teColorText, teColorBack or teBackdrop constants to the field's specification parameter when you create the field. When doing so, the field stores the window's foreground and background colors and displays its text using these colors. After the field is created, you can change the window's colors without affecting the field. The GetFieldColors and SetFieldColors routines are used to set and retrieve a field's text and background colors.

The Appearance Manager does not support the use of colors in static text fields (it supplies colors and patterns that are consistent with the user-selected theme). Initializing Tools Plus with the initPureAppearanceManager option enforces this principle by ignoring custom color information when the Appearance Manager is available.

Disabled Fields

Individual fields can be disabled to prevent the user from making changes to those fields. This can be done by calling the EnableField routine with a value of *false*. A disabled field cannot be activated by your application. The user can't tab to it, click in it, or otherwise change the disabled field's contents. By default, a disabled field's text and its outlining box (if it has one) are grayed out, but you can override this appearance with the DisabledFieldLook and SetDisabledFieldLook routines. You can also disable a field as it is being created by including the teDisabled constant in the field's specification.

Filtering Characters

Tools Plus supports field filters that act like gates to either allow or disallow specified characters into fields. The filter affects any action that puts text into a field: typing, pasting from the Edit menu, and pasting under your application's control.

You create a new filter with the NewFieldFilter routine by specifying characters and the behavior characteristics you want the filter to exhibit. By default, the filter is sensitive to case and diacritical marks, so if you specify only the characters "ABCDE", the filter will consider the characters "e" and "É" to be alien to the character set. You can optionally override a filter's sensitivity to case and/or diacritical marks. When you do this, NewFieldFilter expands the set of characters you specify to account for the other implied characters. The table below illustrates this:

Specified Characters	Option	Filter's Character Set	Comments
ABCDE	(none)	ABCDE	Contains only the characters you specify
ABCDE	ignore case	ABCDEabcde	Also includes upper and lower case equivalents
ABCDE	ignore diacritical marks	AÄÅÀÃÂÁB CÇDEÉÊËÈ	Also includes diacritical equivalents of the same case
ABCDE	ignore case and ignore diacritical marks	AÄÅÀÃÂÁBCÇDEÉÊ ËÈaáàâäãåªbcçdeéèêë	Also includes upper and lower case equivalents, and diacritical equivalents

A filter can also optionally shift all characters to upper case or lower case letters, so a user typing "Smith" would see "SMITH" appear in the field if the "shift to upper case" option is used. This is useful for entering postal codes or part numbers where case uniformity is required.

Use NewFieldFilter to create a filter and to obtain a unique Filter Reference Number (1 through 32767) that is used to reference the filter at a later time.

To make an editing field (or set of fields) adopt a specific filter, use the CurrentFieldFilter routine to specify a filter before the field is created. Subsequently created fields that include the teFilter constant in their specification code adopt the filter specified by CurrentFieldFilter. CurrentFieldFilter also specifies if the filter allows or disallows the characters in its set. You can also assign filters on a field-by-field basis using SetFieldFilter.

If you want to prevent all characters from being typed into the field, consider creating the field with the teReadOnly option which makes the field "read only," thereby allowing the user to select the text and copy it, but not alter it.

Word Wrap

Word wrap occurs automatically in fields. When a word is too long to appear on the current line of a field, the entire word is moved to the next line and scrolling is performed (if necessary) to ensure that the insertion point is visible. A word is defined as any series of characters excluding the space (ASCII character 32) and carriage return (ASCII character 13).

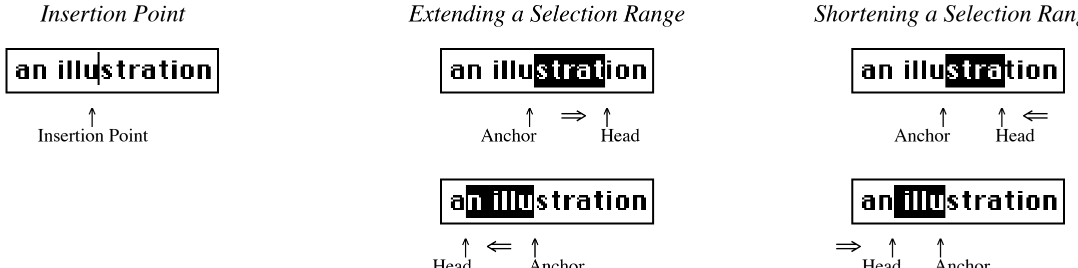
If a field's height is less than or equal to its font's height (font height can be determined by calling the GetFontInfo routine and adding *Ascent* + *Descent* + *Leading*), and you disallow carriage return characters ($0D) in the field, the field is deemed to be a "single line field." Word wrap does not occur in single line fields. Instead, the field's text automatically scrolls to ensure that the selection always remains in view.

User Interaction with Fields

An inactive field cannot be edited by the user. It must first be activated by clicking on the field or tabbing to it. Note, however, that the contents of any field can be changed by your application by using the PasteIntoField routine.

An active field contains either an insertion point (a flashing caret) or a selection (one or more highlighted characters). A selection of characters is made by *extending* (dragging the mouse inside a field) away from the insertion point. The insertion point's position becomes a fixed "anchor," and the selection is *lengthened* by moving the "head" away from the anchor, or *shortened* by moving it towards the anchor. Fields automatically scroll to insure that the selection remains in view.

The following illustration demonstrates the head and anchor of a selection range.



When a field is active, all key-down and auto-key events are automatically intercepted and processed by the field. Notable exceptions and features are listed below.

Note: Older keyboards, namely those found on the Macintosh 512KE and Macintosh Plus, have a "Backspace" key instead of a "Delete" key. However, it performs the same function.

When the user types characters on the Mac's keyboard and an editing field is active, Tools Plus automatically applies those characters to the active field (just as you would expect). However, there are some keys on the keyboard that do not produce characters in the field. Instead, they perform some function pertaining to the active field. Below, is a list of those keys and the function they perform:

⌘	Any ⌘-*key* sequence entered from the keyboard is first interpreted as a menu event. If it matches a menu's ⌘-key equivalent, the corresponding menu is highlighted and a doMenu event is generated. ⌘-key equivalents for Undo, Cut, Copy, and Paste in the Edit menu are processed automatically without generating an event. If the key does *not* match a menu's ⌘-key equivalent, a doKeyDown or doAutoKey event is reported. The edited text and selection are not changed
Enter	If an enabled default button exists, the Enter key is interpreted as a doButton event for the default button. Otherwise, a doKeyDown or doAutoKey event is reported. The edited text and selection are not changed.
Return	If a window is open that contains an active editing field that accepts the Return key, the field executes a carriage return (move insertion point to the next line). If the active field doesn't accept a Return key, the keystroke is ignored. If a field is not active, and a default button exists on the active standard window, the Return key is interpreted as a doButton event for the default button. Otherwise, a doKeyDown or doAutoKey event is reported.
Tab	The Tab key generates a doKeyDown or doAutoKey event and must be processed by your application. The edited text and selection are not changed.
Delete	If the Delete key is pressed at an insertion point, the character immediately to the left is erased without being placed on the clipboard. Backspacing on a selection of characters erases the selection.

Key	Description
Delete Forward	If the Delete Forward key (available on extended keyboards) is pressed at an insertion point, the character immediately to the right is erased without being placed on the clipboard. Deleting Forward on a selection of characters erases the selection.
Home	Scroll field vertically to top. Insertion point or selection is not altered. The field is not scrolled horizontally.
End	Scroll field vertically to bottom. Insertion point or selection is not altered. The field is not scrolled horizontally.
Page Up	Scroll field up by one page (visible area less one line). Insertion point or selection is not altered. The field is not scrolled horizontally.
Page Down	Scroll field down by one page (visible area less one line). Insertion point or selection is not altered. The field is not scrolled horizontally.
Clear	Clear the selected characters in the field without placing them on the clipboard. The clear key and the Clear item in the Edit menu perform the same function.
←	When used at an insertion point, the caret is moved one character to the left. When used at a selection range, the selection becomes an insertion point at the left side of the selection. The edited text is not changed.
Shift ←	Lengthen or shorten the selection by one character (i.e., the range's *head* is moved one character to the left). The edited text is not changed.
Option ←	Move the insertion point one word to the left. When used at an insertion point, the caret is moved leftward to the beginning of a word. When used at a selection range, the selection becomes an insertion point and moves leftward to the beginning of a word. The edited text is not changed.
Option Shift ←	Lengthen or shorten the selection range by one word. When used at an insertion point, a word is selected by moving leftward to the beginning of a word, then the selection is extended by moving the *anchor* rightward to the end of a single word. When used at a selection range, the selection is first checked to ensure that it starts at the beginning of a word, and ends at the end of a word. If this is the case, the selection range is lengthened or shortened by one word (i.e., the range's *head* is moved one word to the left). If the selection does *not* start at the beginning of a word and/or terminate at the end of a word, a word is selected by moving leftward to the beginning of a word, then the selection is extended by moving the *anchor* rightward to the end of a single word. In all cases, the edited text is not changed.
→	Move the insertion point one character to the right. When used at an insertion point, the caret is moved one character to the right. When used at a selection range, the selection becomes an insertion point at the right side of the selection. The edited text is not changed.
Shift →	Lengthen or shorten the selection by one character (i.e., the range's *head* is moved one character to the right). The edited text is not changed.
Option →	Move the insertion point one word to the right. When used at an insertion point, the caret is moved rightward to the beginning of a word. When used at a selection range, the selection becomes an insertion point and is moved rightward to the beginning of a word. The edited text is not changed.
Option Shift →	Lengthen or shorten the selection range by one word. When used at an insertion point, a word is selected by moving rightward to the beginning of a word, then the selection is extended by moving the *anchor* leftward to the end of a single word. When used at a selection range, the selection is first checked to ensure that it starts at the beginning of a word, and terminates at the end of a word. If this is the case, the selection range is lengthened or shortened by one word (i.e., the range's *head* is moved one word to the right). If the selection does *not* begin at the beginning of a word and/or terminate at the end of a word, a word is selected by moving rightward to the beginning of a word, then the selection is extended by moving the *anchor* leftward to the end of a single word. In all cases, the edited text is not changed.

↑ or Option ↑	Move the insertion point up one line. When used at an insertion point, the caret is moved up one line. When used at a selection range, the selection becomes an insertion point at the left side of the selection. Nothing happens if the insertion point is on the field's first line. The edited text is not changed.
Shift ↑ or Shift Option ↑	Lengthen or shorten the selection by one line (i.e., the range's *head* is moved up one line). The edited text is not changed.
↓ or Option ↓	Move the insertion point down one line. When used at an insertion point, the caret moves down one line. When used at a selection range, the selection becomes an insertion point at the right side of the selection. Nothing happens if the insertion point is on the field's last line. The edited text is not changed.
Shift ↓ or Shift Option ↓	Lengthen or shorten the selection by one line (i.e., the range's *head* is moved down one line). The edited text is not changed.
Clicking or Click/Drag	Clicking in an active field deselects the current selection and places an insertion point where the mouse was clicked. If the mouse button is held down, the insertion point may be dragged to form a selection range. The field's text scrolls automatically to keep the selection in view. If a click occurs in an *inactive* field, a doClickToFocus event is reported. Your application should then respond by validating the active field's text (GetEditString or GetEditHandle routine), then saving the field's edited text by using the SaveFieldString routine. Then, by calling the ClickToFocus routine, the click is processed as previously described.
Double-Clicking or Double Click/Drag	Double-clicking in an active field deselects the current selection and selects the word that was clicked. If the mouse button is held down, the selection range may be dragged to extend or shorten the range by one word. The field's text scrolls automatically to keep the selection in view. If a double-click occurs in an *inactive* field, a doClickToFocus event is generated. Your application should then respond by validating the active field's text (GetEditString or GetEditHandle routine), then saving the field's edited text by using the SaveFieldString routine. Then, by calling the ClickToFocus routine, the double-click is processed as previously described.

Mac 512KE and Mac Plus keyboard with numeric pad

In order to provide the Shift-Arrow combinations as previously described, Tools Plus must discern an Arrow key from a Shift-Arrow key. This causes a slight problem on Macintosh 512KE and Macintosh Plus keyboards. Shift-←, for example, produces the same key code as the "+" on the numeric pad. A problem arises when Tools Plus cannot discern between a Shift-← and a "+" key on the numeric pad.

It is for this reason that the =, /, *, and + keys on the numeric pad are treated as Shift-↓, Shift-↑, Shift-→, and Shift-← respectively. This occurs only in an active field on a Macintosh 512KE and Macintosh Plus.

The Edit Menu

If a second menu exists, it must be called "Edit" and must contain "Undo", "Cut", "Copy", "Paste" and "Clear" as the first five items in the listed order. A dividing line must exist between "Undo" and "Cut". See the chapter on Menus for more details.

When a field is active in a window that allows access to pull-down menus, the Edit menu's "Undo", "Cut", "Copy", "Paste", and "Clear" items automatically affect the active field in the following manner:

Undo	Undo is disabled when a field is activated or deactivated. It is enabled and changed to "Undo Cut," "Undo Copy" and "Undo Paste" when the respective menus are selected, and changed to "Undo Typing" when keys are typed or the Delete key is used at an insertion point. Selecting "Undo *task*" performs all the necessary operations that are required to undo the previous operation, and changes the item to "Redo *task*." Selecting "Redo *task*" restores the field to a state before "Undo *task*" was used. Undoing and Redoing also remembers insertion point positions and selection ranges.
Cut	Cut deletes the selected text from the field and places it on the Clipboard. This item is disabled when a field is deactivated or when an insertion point exists in an active field (i.e., a selection range has not been made).

Copy — Copy takes a copy of the selected text and places it on the Clipboard without changing the field. Copy is disabled when a field is deactivated or when an insertion point exists in an active field (i.e., a selection range has not been made).

Paste — At an insertion point, Paste inserts a copy of the Clipboard's text. At a selection range, Paste replaces the selected characters with the Clipboard's text. The Clipboard's contents are not changed.
In a length limited field, the Clipboard's text is pasted a character at a time (although very quickly) until all the Clipboard's text has been pasted, or the field is full (whichever comes first).
Paste does not paste Carriage Returns into fields where they are disallowed.
Paste is disabled when a field is deactivated or when no text exists on the Clipboard. Paste is also disabled if a field has been length limited and the insertion point is in a field that has reached its limit of characters (i.e., a full field).

Clear — Clear is disabled when a field is deactivated or when an insertion point exists in an active field (i.e., a selection range has not been made). Clear deletes the selected characters in the field without placing them on the clipboard. The clear key and the Clear item in the Edit menu perform the same function.

Select All — The "Select All…" item is optional, but frequently used in applications. It selects all the text in the active field.

Note: See the Multiple Languages chapter for changing the text that appears in the Edit menu's Undo item. Tools Plus supports multiple languages, and you can replace the English text with equivalent words of your own.

Large Fields and Buffers

Tools Plus is designed to be quick and to make efficient use of memory. One of the methods it uses to accomplish this is to create only one TextEdit record for each window, regardless of the number of fields the window has. (A TextEdit record is a mechanism used by the Macintosh toolbox to display the contents of a field and to allow its text to be edited.) This strategy saves at least 70 bytes per field as compared to having a TextEdit record for each field. A negative side effect is that there is a performance penalty when activating a field or when an inactive field is refreshed. This effect is inperceivable when working with fields containing small amounts of text, however it becomes quite annoying when working with fields that hold larger amounts.

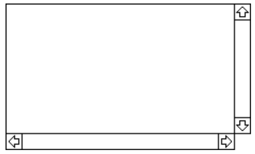
As an option, you can create a Tools Plus field with its own TextEdit record by adding the teBuffered option to the field's specification. When you do this, Tools Plus creates and maintains a TextEdit record for the field thereby giving it maximum performance at all times. This is called a *buffered* field. The price you pay for this additional performance is that more memory is required for fields using this option: an additional 96 bytes + 2 bytes per line of text in the field.

You should consider using a buffered field under any of the following conditions:
- field's string is over 1K and your application is expected to run on a Mac Plus
- you need excellent response on a mid-powered Mac (25 MHz '040) and the field's string is over 4K
- you need acceptable response on a mid-powered Mac and the field's string is over 8K

Fields with Scroll Bars

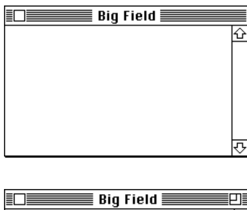
Tools Plus's fields can optionally have a vertical and/or horizontal scroll bar (except single-line fields which cannot have any). The field's text and its scroll bars are kept synchronized at all times: changing the text or moving through it updates the scroll bars, and moving the scroll bars scrolls the field's text.

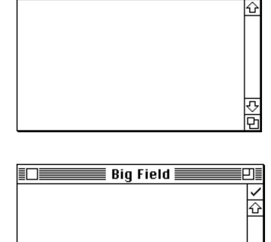
To add a vertical scroll bar along the right side of your editing field, add the teVScroll option to the field's specification. The vertical scroll bar typically runs along the field's right side, so for the best visual results, make sure your field is at least 50 pixels high so that you can see the scroll bar's up arrow, down arrow and thumb. The top of the scroll bar can be brought down in 15 pixel increments by adding the teVScrollDown option. This lets you place picture buttons or other user interface elements just above the vertical scroll bar. The bottom of the scroll bar is brought up automatically to accommodate a window's grow box if necessary.



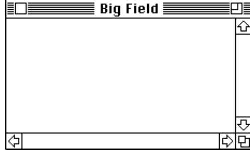
A vertical scroll is placed just outside the field's right edge by adding teVScroll to the field specification. Use NewWideField to create a field with a horizontal scroll bar.

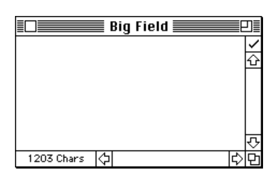
To add a horizontal scroll bar along the bottom edge of the field, use the NewWideField routine. It requires one additional parameter that specifies how wide the area containing the text is (the width at which word wrapping takes place, referred to as the *destination rectangle* in Inside Macintosh). The left side of the scroll bar can be moved to the right in 15 pixel increments by adding the teHScrollRight option. This lets you place picture buttons or other user interface elements just below the bottom left corner of the field. The right side of the scroll bar is automatically moved left to accommodate a window's grow box if necessary.





When placing a field against a window's right or bottom edge, Tools Plus automatically makes room for the grow box (if the window has one) by shortening the scroll bars.





Scroll bars can be pushed away from the left and top of the field to make room for other user interface elements by adding teHScrollRight or teVScrollDown constants to the field's specification.

Normally when the user drags the scroll bar's thumb, an outline of the thumb tracks the cursor then when the user releases the mouse button, the scroll bar's thumb and the related text snap to the new position. A useful feature that is not a Macintosh standard is *live scrolling*. If you add the teLiveScroll option to the field's specification, the text is scrolled in real time when the user drags a scroll bar's thumb. If you use live scrolling in your fields, you should do so with all fields to give your interface a consistent feel.

Memory Management

This section describes how Tools Plus manages memory in relation to editing fields and the clipboard. If your application uses large fields (1K to 32K), this section will help you understand how Tools Plus is protecting your application in low memory situations, and how a "tiny" application can suddenly become "memory hungry." It is also a good idea to understand how much memory Tools Plus expects or will consume in certain situations so that you can write your application appropriately in anticipation of this.

Tools Plus automatically maintains all the "memory objects" (data) related to your editing fields and the clipboard. Even though you may not immediately realize it, in a worse-case scenario a single 32K editing field can consume over 128K of your application's memory, even if only on a temporary basis. If you are not aware of this or your application is not prepared for a sudden consumption of memory, Tools Plus will do various things to protect your application from behaving ungracefully or from getting dangerously low on memory. Details are provided later in this section.

The following is a list of "memory objects" maintained by Tools Plus and details on *how* they are maintained. Throughout this section, references to *memory* refer to your application's heap and not the stack. If you don't know what a heap or stack is, it is sufficient to say that it means the section of Macintosh memory that is reserved by (and exclusively for) your application while it is running.

Desk Scrap

The desk (or System) scrap is equivalent to what users call the Clipboard. It is used to transfer text, images, or other kinds of data between applications and desk accessories. The desk scrap can contain multiple "versions" of data: a paragraph represented as plain text, and a copy of the paragraph represented as text formatted by its originating word processor. Nearly all applications include a "plain text" version of the text they are cutting or copying to the clipboard (they're all supposed to), and Tools Plus accesses only the plain text ignoring other kinds of data.

When you launch your application, the desk scrap normally shares your application's memory. During the launch, if the desk scrap is bigger than half your application's memory, the Macintosh toolbox automatically unloads the desk scrap to disk so that it does not occupy any application memory. Finally, InitToolsPlus (Tools Plus initialization) unloads the desk scrap to disk if there is less that 90K of free memory just before Tools Plus libraries are initialized (although you can override this default).

It is important to remember that the desk scrap can potentially occupy up to half your application's memory at startup. This can also happen while your application is running if you switch to a word processor and copy a large selection of text to the clipboard. That copied text can potentially consume all your free application memory.

Whenever any application cuts or copies anything, it ends up in the desk scrap, possibly with multiple copies in different formats. When your application cuts or copies text from an editing field, it too ends up in the desk scrap but only as plain text. When you paste text in a Tools Plus editing field, the plain text is copied from the desk scrap to your editing field.

TextEdit Scrap

The TextEdit scrap is a local copy of the desk scrap. It contains only text data ignoring images and other kinds of data. TextEdit scrap is necessary only if your application uses editing fields that are not created by Tools Plus. The TextEdit scrap always has a "plain text" copy of the text in the desk scrap. Having a TextEdit scrap can consume up to 32K of memory. By default, Tools Plus does not create or maintain a TextEdit scrap although you can override this default at initialization in the InitToolsPlus routine.

The size of the TextEdit scrap will not grow beyond the "buffer size" you specify when initializing Tools Plus (in the InitToolsPlus routine).

Scrap "Undo" Text

Tools Plus automatically lets you undo and redo edits in an editing field. The undo/redo services also let you undo a copy or cut, which obviously put new text in the scrap (Clipboard). Just before you cut or copy text from a Tools Plus editing field, a copy of the clipboard's text is automatically made which can consume up to 32K of memory. Undo Cut, Redo Cut, Undo Copy and Redo Copy automatically swap the text that is currently on the clipboard with the "scrap 'undo' text."

The size of the "scrap 'undo' text" will not grow beyond the "buffer size" you specify when initializing Tools Plus (in the InitToolsPlus routine).

Field's String

The field's string is a Pascal string or C string that is referenced by a handle. It contains the field's permanent text (not the text being edited by the user), and usually does not change its size. You may optionally decide to have the handle automatically resize to accommodate the text it contains ("dynamic handles" option when creating a field). This saves memory overall, but may introduce additional risk by suddenly consuming additional memory when you save your field's edited text using the SaveFieldString routine.

The size of the field's string handle will not grow if you have not specified that you are using dynamic handles in editing fields. If you are using a dynamic handle, it will not grow beyond its own maximum size (the size is was initialized to).

Field's Edited Text

The field's edited text is the text being edited by the user when the field is active. Each window can have one active field, and can therefore consume up to 32K of memory (per field) as the user types or pastes text into the field. When you deactivate a field (using DeactivateField or TabToFocus, or by closing the parent window), the edited text is destroyed and its memory is released.

Edited text can consume up to 32K of memory per active field. You can reduce this amount by using length limited fields which allow a user to type a certain number of characters into a field. Additional characters are not accepted when the field is full, and the user is beeped when they try to type more characters.

Edited “Undo” Text

Tools Plus automatically lets you undo and redo edits in an editing field. Just before any change is made in a field (cut, paste, clear, typing, backspace, or delete forward), a copy if the field’s edited text is automatically made. This copy can consume up to 32K of memory. Undo/redo automatically swap the field’s edited text with the “edited ‘undo’ text.”

The size of the “edited ‘undo’ text” will not grow beyond the “buffer size” you specify when initializing Tools Plus (in the InitToolsPlus routine). Also, it won’t exceed the number of characters of edited text in the field.

”Low Memory” Protection

When it comes to editing fields, Tools Plus automatically protects your application from running out of memory and from getting into situations where memory is dangerously low. Your application can define several memory thresholds that trigger certain actions:

- *Low memory for editing*: If the largest piece of continuous memory is smaller than this specified value after the Undo/Redo services have been set up, the user is warned with a message stating “Low memory… Continue without ‘Undo/Redo’?” A “Continue” button lets the user continue without the Undo/Redo services being set up (i.e., the Edit menu’s “Undo…” item is disabled and set to “Can’t Undo”). A “Cancel” button lets the user cancel the editing operation without making any changes. See the SetTENoUndoThresh routine to set this value.
- *No memory for editing*: If there is not enough memory to set up the Undo/Redo services and the largest piece of continuous memory is smaller than this specified value after the edit is performed (such as a paste or typing), the user is warned with a message stating “WARNING… Not enough memory for this operation.” A “Cancel” button lets the user cancel the editing operation without making any changes. See the SetTENoEditThresh routine to set this value.
- *Low memory while typing*: If the largest piece of continuous memory is smaller than this specified value after the user types a character, the user is warned with a message stating “WARNING… Low memory!” An “OK” button lets the user proceed. This message is displayed every 90 seconds as long as the user continues to type while memory is low. See the SetTELowMemThresh routine to set this value

All these thresholds are set to reasonable default values when Tools Plus is initialized, so your application benefits from low-memory protection without you having to explicitly do anything. You can change the messages displayed by Tools Plus as described in the Multiple Languages chapter.

Tips for Conserving Memory

(1) Use UnloadScrap at the beginning of your application to store the desk scrap on the disk. This can be done as part of initialization by InitToolsPlus. Remember to use LoadScrap just before you quit your application.

(2) Don’t use a TextEdit scrap. This is done as part of initialization by InitToolsPlus.

(3) Limit copy, paste, undo/redo to less than 32K. This is done as part of initialization by InitToolsPlus. If your largest field is only 400 characters, why be able to cut, copy, paste or undo/redo more than that?

(4) Length limit your fields to prevent the user from being able to type or paste a full 32K into a field.

(5) If you don’t use dynamic handles (declared when creating a field), you won’t have to worry about the field’s string suddenly growing. However, you can save memory overall by using dynamic handles in your fields.

Handling Fields

Once a field is activated, Tools Plus performs all the editing required within the field. When a window in inactive, Tools Plus deselects the text in the active field on that window and hides the insertion point. When the window is activated again, the active field regains its original state.

Tools Plus constantly inquires about any events that have occurred, including typing in fields. It also maintains the enabled/disabled status for the Edit menu’s Undo, Cut, Copy, Paste, and Clear items. The active field is automatically affected if the user selects the Edit Menu’s Undo, Cut, Copy, Paste, or Clear command. Command key equivalents for these items have the same effect.

If you did not use the initAutoFocusChanges option when initializing Tools Plus, many types of events *may* indicate that the user has completed using the field. For example, a doKeyDown event may report that Tab or Return was pressed, indicating the user wishes to advance to the next field. The doClickToFocus event may indicate that the user

has clicked on an inactive field, or the doMenu event may indicate that the user wants to quit your application. In these cases, you will likely want to validate the active field's edited text before accepting it as the field's string. A copy of the field's edited text can be obtained by GetEditString or GetEditHandle. If your application determines that the edited text is invalid, display an appropriate alert box and ignore the event. If no error is detected, call the SaveFieldString routine to save the edited text as the field's string, then process the event.

If your application needs to monitor changes as they occur in an editing field, it can do so by responding to the doChgInField event. An example of this is disabling a "save" button when a field is empty.

See the Event Management chapter for details.

Special Handling of Fields

Some applications may find it necessary to reposition a field. An example of this occurs in an application that has a matrix of fields aligned as cells: 3 columns across and 10 lines down. If your application needs to "scroll" this block of fields, it is necessary to change the position of existing fields. This can be done by using the OffsetField routine.

Another unusual circumstance occurs if your application needs to paste text into a field under your application's control. For example, your application may choose to insert a commonly used word or phrase into a field when the user selects a menu item or a specific command key. The PasteIntoField routine allows text to be pasted directly into a field.

If your application sets other user interface elements depending on a field's contents (i.e., disabling a "Save" button if a field is empty), then you can use the doChgInField event which is reported whenever the active field's contents are changed. See the Event Management chapter for details.

Appearance Manager and Keyboard Focus

Before the Appearance Manager's arrival, the only user interface element that could process keystrokes was an editing field. With the Appearance Manager present, a variety of user interface elements can process keystrokes, such as editing fields, list boxes and the clock control. Keystrokes are directed at only one user interface element at a time (and possibly at no element at all). When a user interface element processes keystrokes in such a way, it is said to have the "keyboard focus." Only one user interface element can have the keyboard focus at a time, and it is visually indicated with a highlighted "band" around the object. Tools Plus takes care of the focus highlight, and of applying keystrokes to the element that has the keyboard focus.

The process of moving the keyboard focus between objects, either by tabbing or clicking, is identical to that of navigating between editing fields. For details, see the "Clicking and Tabbing" section in this chapter.

Appearance Manager Controls

The Appearance Manager, first introduced in mid 1997 with Mac OS 8, gives your application a number of 3D controls including editing fields with a 3D box around them, and static text controls. All the new Appearance Manager controls are implemented as CDEFs, but unlike third party CDEF resources that must be installed in your application when it is built, the Appearance Manager's edit text and static text control is available to your application without having to install them. They are available from the system, just like regular system controls, if the Macintosh running your application has an Appearance Manager.

If you want to use the Appearance Manager's edit text control or static text control, you need to make your application "Appearance Manager aware." 680x0 applications are automatically Appearance Manager aware. To make your PowerPC application Appearance Manager aware, see the Designing Your Application chapter of this manual for details in the "Using the Appearance Manager" section.

See the chapters on Buttons, Scroll Bars, List Boxes and Pop-Up Menus in this user manual for additional Appearance Manager controls.

Note: For complete information on Appearance Manager concepts, the Appearance Manager's features, and how to best use the Appearance Manager's new controls, please read the documentation pertaining to the Appearance Manager. It is available from Apple or in the latest issue of Inside Macintosh. This manual does not duplicate that material.

Edit Text (CDEF 17)

The Edit Text control is implemented as a Tools Plus editing field, so you can use the same Tools Plus routines to create and work with standard TextEdit fields or with the Appearance Manager's Edit Text controls.

Sample

Edit Text control

The Appearance Manager's Edit Text controls do not support some of features that are part of Tools Plus. This is due to the limitations of the Appearance Manager.

- Scroll bars cannot be attached to the field
- Tools Plus's "smart scrolling" (text scrolling that accelerates as you drag the mouse further out of the editing field)
- The following cursor control keys do nothing: Home, End, Page Up and Page Down
- A field's text is always reset to the top when the field is deactivated
- Word-wrap is always on. This is most noticeable in 1-line fields that are not length limited because as the text wraps, all the user sees in the caret at the left of an empty field. The user is in fact on the second line of the field, but he cannot see the first line of text so it may appear that the text disappeared. This is standard behavior for editing fields, but in Tools Plus's fields that don't use the Appearance Manager's Edit Text control, text only scrolls horizontally without word wrap.

```
CONST
   kControlEditTextProc = 272;          {Edit Text ProcID                   }
```

Static Text (CDEF 18)

The Static Text control can be implemented as a non-selectable button (see the Buttons chapter), or as a special kind of field called a static text field. You can use static text controls in place of standard static text items in dialogs. The advantage this provides is that the text looks disabled on an inactive window (it is dimmed) and you can easily manipulate the text as you would any other control, such as hiding and showing the control. The user cannot interact with this control.

Static Text

Enabled

Static Text

Disabled

```
CONST
   kControlStaticTextProc = 288;        {Static Text ProcID                 }
```

Creating a Field Using a 'CNTL' Resource

Tools Plus offers considerable versatility in the way it supports the creation of editable fields and static fields from 'CNTL' resources. These features are most often used when opening a dialog ('DLOG' resource) that contains fields or static text items. In all cases, the 'CNTL' resource specifies a CDEF ID of 17 which produces a procID of 272 plus any variants for an editable field, or a CDEF ID of 18 which produces a procID of 288 plus any variants for a static text field. When you open a dialog, 'CNTL' resources that reference these CDEF IDs (the edit text or static text control) create a Tools Plus field. The translation from a 'CNTL' resource to a Tools Plus field takes place as follows:

- Tools Plus starts by assuming that you want to use the Appearance Manager's edit text control (CDEF 17) or static text control (CDEF 18) and it attempts to create the control.
- If the Appearance Manager is not available, a regular TextEdit field is created. You can use the same Tools Plus routines to access standard TextEdit fields as you would an edit text or static text control.

- The field is created using a default appearance and behavior specification. You can change this default value using the SetDialogEditTextSpec and SetDialogStaticTextSpec routines.
- To set the appearance and behavior specifications for a field, place the specification's value in the 'CNTL' resource's contrlRfCon field, the reference constant. A list of possible values can be found in the NewField description.

Flag your 'CNTL' resources as purgeable to save memory. Tools Plus makes a copy of their data.

NewStrHandle

Allocate memory for a string and initialize it to an empty string.

C
```
pascal StringHandle NewStrHandle (short StringLen);
```

Pascal
```
function NewStrHandle (StringLen: INTEGER): StringHandle;
```

StringLen is the size of the string being allocated, from 1 to 32767 characters. Although a Pascal string (Str255) can contain a maximum of 255 characters, Pascal applications can still reference the larger 32K structure as a large block of text. NewStrHandle automatically allocates one additional byte to account for a Pascal string's length byte or a C string's null terminator.

The routine's value returns a handle to a string of the specified length. It is initialized to an empty string (''), thus making is ready for immediate use with the NewField or NewFieldRect routines.

When you are finished using this handle, you can deallocate it using the toolbox's DisposeHandle routine. Be careful not to deallocate this handle after you assign it to a field unless the field is deleted first.

☞ **Warning**: When allocating a Pascal string handle, make sure you do not use a value that is greater than 255 (Pascal's maximum string size). If you do, NewField will reject the handle and tell you that you have given it an invalid parameter.

NewField

Create a new field. The new field is *not* activated.

C
```
pascal void NewField (short Field, short left, short top, short right,
            short bottom, Handle hStr, long Spec, short Just);
```

Pascal
```
procedure NewField (Field, left, top, right, bottom: INTEGER;
            hStr: HANDLE; Spec: LONGINT; Just: INTEGER);
```

Field specifies the field number (from 1 to 32767, or 1 to 511 when using the Appearance Manager's Edit Text control or Static Text control) that is created in the current window. Once a field is created, it is referenced by this field number. If a field has been previously created in the current window using the same number, it is replaced with a new field as specified by the parameters in the NewField routine. If the current window doesn't belong to your application, or if no windows are open, NewField does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in local co-ordinates that determines the field's size and location in the current window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). The field must be wide enough for at least 1 character (minimum of 5 pixels). The height of the field should be the same as a font's height (font height can be determined by calling the GetFontInfo routine and adding *Ascent* + *Descent* + *Leading*). In multiple-line fields, the height should be in increments of the font height. If top and bottom are the same, NewField calculates the line height for you and creates a field that is one line high. If bottom is less that top, NewField takes the absolute value of bottom and creates a field that is that many lines

high (i.e., if top is 30 and bottom is -4, a four line field is created starting downward at the top co-ordinate).

You can align the field's edge to a window's edge by using the teLeftEdge, teTopEdge, teRightEdge or teBottomEdge constants. Tools Plus calculates the field's co-ordinates correctly regardless if your field has scroll bars or not.

HStr is a handle that points to a Pascal string or a C string. It provides this editing field with a reference to its associated text. You must either allocate memory for this handle and set it to a default string value before using it here, or specify nil and let NewField allocate the string for you. Using the NewStrHandle routine is a good way to allocate and initialize a string handle of a required size. When you specify nil, NewField allocates a handle to a 255 character string structure and initializes it to zero characters. NewField can automatically allocate a 32767 character C string if you include the teCstring and teBuffered options (high speed buffering). An automatically allocated string is automatically deallocated when the field is deleted or when its parent window is closed. If you must lock this handle, do so temporarily and make sure it is unlocked before calling any Tools Plus routines.

Spec specifies the appearance and behavior of the editing field. See "Appearance and Behavior Specification" for details later in this section.

Just specifies if a field is left aligned, right aligned, or centered. The constants teJustLeft, teJustRight or teJustCenter can be used.

Appearance and Behavior Specification

Spec specifies a field's appearance and behavior. The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result. The constants defining the available options are as follows:

Optionally choose any of the following options…

`teSystemBody`	Create the field using the Appearance Manager's edit text control or static text control. If the Appearance Manager is not available, a regular TextEdit field is created. When you create a field with this option, the text and optional box are dimmed when the field is on an inactive window. **Note**: Only fields that are implemented as a control can be embedded into other controls.
`teStaticText`	Create the field as a static text field. This option is used to create display-only fields that do not have a box around them. They look just like ordinary text does on a window, except static text fields are automatically refreshed and your application can change their text as required.
`teReadOnly`	This option prevents the user from editing the field in any way (typing, Edit menu, etc.) If the field is editable and enabled, the user can select text in the field and copy it only.
`teUseWFont`	Display the field using the window's current font, size and style settings (as set by the TextFont, TextSize, and TextFace routines). The field stores this information for future reference. By default, fields are drawn using the system font (Chicago, 12 pt).
`teDimWhenInactive`	Dim the field as though it is disabled when it is displayed on an inactive window. This option is automatically turned on when the teSystemBody option is used.
`teAllowCR`	Allow carriage return characters (Return) in the field. By default, carriage returns are disallowed and are ignored while a user is typing in a field.
`teNoCR`	Disallow carriage return characters (Return) in the field. This is the default and it does not need to be explicitly included.
`teBox`	A 1-pixel wide box is drawn around the field as a rectangle that is exactly 3 pixels larger than the coordinates specified by left, top, right and bottom. This is the default and it does not need to be explicitly included.

`teNoBox`	A box is not drawn around the field. One reason you may decide not to have a box drawn around your field is to accommodate many small fields that are laid out as a grid to resemble a spreadsheet.
`teLengthLimit`	The user is prevented from typing more characters than the field's maximum string length. By default, the user can type up to 32767 characters in a field. All fields can be length limited by using FieldLengthLimit(on) immediately after Tools Plus initialization.
`teResizeHdl`	Automatically resize the field's text handle to accommodate the exact amount of text stored in it. By default, the handle's size is not changed by Tools Plus. When using this option, the handle's size may shrink, but it will not grow larger than the originally supplied handle size. If you specify nil in place of a string handle (thereby indicating that NewField should allocate the handle for you), a Pascal string handle may grow up to 255 characters while a C string handle may grow up to 32767 characters. This option saves memory but introduces additional risk because memory consumption becomes more dynamic.
`teFilter`	Apply the current field filter to this field (the current filter is specified by the CurrentFilter routine). The field remembers the current filter. Whenever the user types or pastes into the field, that filter is used to allow or disallow a specific character set and/or to convert characters to upper or lower case letters. See NewFieldFilter and CurrentFilter for details on creating and setting a filter.
`teCstring`	The hStr handle points to a null-terminated C string. By default, NewField expects the handle to point to a Pascal string which is prefixed by a length byte.
`teDisabled`	The field is disabled. By default, a new field is enabled.
`teAutoMoveSize`	Automatically move and/or resize the field and its scroll bars when the window's size changes. The AutoMoveSize routine lets you specify which sides are altered. You can use the AutoMoveSizeField routine as an alternative to setting this option.
`teHidden`	Create a hidden field. This kind of field is accessible to your application but not to the user.
`teBuffered`	Increase the field's performance by assigning a TextEdit record to this field. This requires more memory but speeds up activating the field and refreshing it when it is inactive. Not needed for fields storing less that 1K. **Note**: If the field's text handle was automatically allocated (by specifying nil as a string handle) Tools Plus does not allocate a Pascal or C string. It allocates a generic text handle that is the exact length of the text. It is not prefixed with a Pascal-styled length byte or null-terminated like a C string.
`teNoResetOnDeactivate`	By default, a field scrolls to the top left (or right if it's right aligned) when it is deactivated. Use this option to leave the field as the user leaves it when the field is deactivated (i.e., when tabbing to another field, when the user clicks in another field, or when the application activates another field).
`teVScroll`	Include a vertical scroll bar for this field. By default, fields do not have scroll bars. The scroll bar is created *outside* the field's co-ordinates.
`teLiveScroll`	Scroll the field's text in real time as the user moves the scroll bar's thumb. This is not a Macintosh user interface standard. By default, an outline tracks the mouse as the user drags the scroll bar's thumb. When the user releases the mouse button, the scroll bar's thumb and the field's text snap to their new position.

Optionally choose only one of the following text selection options…

`teTabSelectAll`	Select the entire field's text when tabbing into this field. This is the default and it does not need to be explicitly included.
`teTabSelectEnd`	Place an insertion point at the end of the field's text when tabbing into this field.
`teTabSelectStart`	Place an insertion point at the beginning of the field's text when tabbing into this field.

Optionally choose only one of the following vertical scroll bar offset options…

`teVScrollDown15`
`teVScrollDown30`
`teVScrollDown45`
⇓
`teVScrollDown195`
`teVScrollDown210`
`teVScrollDown225`
	Bring the top of the vertical (right) scroll bar down by multiple of 15 pixels to allow additional user interface elements to be placed above the scroll bar. Normally, the vertical scroll bar extends to the top of field.
	No offset — With teVScrollDown15 offset

Optionally choose only one of the following background options…

`teWhiteBack`	The field's background is white. This is the default and it does not need to be explicitly included.
`teBackdrop`	The field's background is the same color as the window's backdrop color.
`teColorBack`	The field remembers the window's background color and uses it as the field's background color. If Color QuickDraw is unavailable (or unused), the background is white.

Optionally choose only one of the following text color options…

`teBlackText`	Text is black. This is the default, so omitting all options implies using this one.
`teColorText`	The field remembers the window's foreground color and uses it as the field's text color. If Color QuickDraw is unavailable (or unused), the text is black.

Single Line Fields

Tools Plus lets you create single line editing fields in which word wrap does not occur. Instead, characters are automatically scrolled horizontally along a single line. To create a single line field, set the field's height equal to its font's height (font height can be determined by calling the GetFontInfo routine and adding *Ascent* + *Descent* + *Leading*). You can also accomplish this by specifying a bottom co-ordinate that is the same as the top and letting NewField calculate the exact line height. You must also specify carriage returns are disallowed in the field.

Also see: NewFieldRect and NewDialogField
FieldLengthLimit to limit the length of editable text
EnableField to enable or disable a field
NewFieldFilter, CurrentFieldFilter and SetFieldFilter to utilize character filtering and or/case shifting
NewWideField and NewWideFieldRect for fields with a horizontal scroll bar

✍ **Note**: The numeric range for the field number (1 through 32767) for each window is a *theoretical* limit. Your actual limit will be determined by the amount of available memory and your processor's speed. Even though the Macintosh running your application may have enough memory, a large number of fields can slow down operations that access fields such as tabbing between fields and activating fields. This is only a concern with a very large number of fields since even the slowest Macintosh can easily handle a few hundred fields in a single window.

✍ **Note**: Tools Plus makes no attempt to control the placement of fields or to protect them once they have been created. It is your responsibility to ensure that fields are a sufficient size (at least 1 character wide and high), and that their placement within the window is reasonable and does not conflict with other objects. Furthermore, you should not allow your application's text and drawing processes to interfere with fields. Windows with a "size box" should not allow fields to be obscured or hidden by making the window too small.

☞ **Warning**: Your application must allocate memory for each handle that it provides to NewField by using the NewHandle routine. Tools Plus does not automatically allocate this memory. If your editing fields contain random characters, it is a sure sign that you have not allocated memory for your handle.

```
CONST                                               {Behavior and Appearance Options:           }
                                                    {Field co-ordinates:                        }
               teLeftEdge    =-32768;               {  Window's left edge                       }
               teTopEdge     =-32768;               {  Window's top edge                        }
               teRightEdge   = 32767;               {  Window's right edge                      }
               teBottomEdge  = 32767;               {  Window's bottom edge                     }
                                                    {Carriage Returns and Box:                  }
               teAllowCR     = $00000100;           {  Allow Carriage Return in text            }
               teNoCR        = $00000000;           {  Disallow Carriage Return in text         }
               teBox         = $00000000;           {  Draw box around field                    }
               teNoBox       = $00000200;           {  Don't draw box around field              }
                                                    {Combined Box and/or CR constants:          }
               teBoxNoCR     = teBox + teNoCR;      {  Box around field, no CR allowed          }
               teBoxCR       = teBox + teAllowCR;   {  Box around field, CR allowed             }
               teNoBoxNoCR   = teNoBox + teNoCR;    {  No box, no CR allowed                    }
               teNoBoxCR     = teNoBox + teAllowCR; {  No box, CR allowed                       }
                                                    {Other Options:                             }
               teSystemBody      = $00000020;       {Use Appearance Manager Edit Text control   }
               teStaticText      = $00000040;       {Create 'Static Text' field                 }
               teReadOnly        = $00000080;       {Field cannot be edited by user             }
               teDimWhenInactive = $00000004;       {Dim when field is on an inactive window    }
               teUseWFont        = $00000008;       {Use window's font                          }
               teLengthLimit     = $00000400;       {  Limit typing to field's length           }
               teResizeHdl       = $00000800;       {  Resize text handle to save memory        }
               teFilter          = $00001000;       {  Use current field filter                 }
               teCstring         = $00002000;       {  C string text format (32K max)           }
               teDisabled        = $00004000;       {  Field is disabled                        }
               teHidden          = $00008000;       {  Create a hidden field                    }
               teBuffered        = $00040000;       {  Accelerate large field with buffer       }
               teAutoMoveSize    = $00080000;       {  Auto-resize as window's size changes     }
               teNoResetOnDeactivate = $00100000;   {  Retain scrolling on deactivate           }
               teLiveScroll      = $00200000;       {  Live scrolling using scroll bars         }
               teVScroll         = $00400000;       {  Field has a vertical scroll bar          }
                                                    {Default selection on activation:           }
               teTabSelectEnd    = $00010000;       {  Insertion point at end                   }
               teTabSelectStart  = $00020000;       {  Insertion point at start                 }
               teTabSelectAll    = $00000000;       {  Select all text                          }
                                                    {Move top of vertical scroll bar down by:   }
               teVScrollDown15   = $01000000;       {  15 pixels                                }
               teVScrollDown30   = $02000000;       {  30 pixels                                }
               teVScrollDown45   = $03000000;       {  45 pixels                                }
               teVScrollDown60   = $04000000;       {  60 pixels                                }
               teVScrollDown75   = $05000000;       {  75 pixels                                }
               teVScrollDown90   = $06000000;       {  90 pixels                                }
               teVScrollDown105  = $07000000;       {  105 pixels                               }
               teVScrollDown120  = $08000000;       {  120 pixels                               }
               teVScrollDown135  = $09000000;       {  135 pixels                               }
               teVScrollDown150  = $0A000000;       {  150 pixels                               }
               teVScrollDown165  = $0B000000;       {  165 pixels                               }
               teVScrollDown180  = $0C000000;       {  180 pixels                               }
               teVScrollDown195  = $0D000000;       {  195 pixels                               }
               teVScrollDown210  = $0E000000;       {  210 pixels                               }
               teVScrollDown225  = $0F000000;       {  225 pixels                               }
                                                    {Background Color:                          }
               teWhiteBack   = $0000;               {  White background (default)               }
               teBackdrop    = $0001;               {  Draw on backdrop color                   }
               teColorBack   = $0002;               {  Color background                         }
                                                    {Text Color:                                }
               teBlackText   = $0000;               {  Black text (default)                     }
               teColorText   = $0010;               {  Foreground colored text                  }
                                                    {Combined text & background color constants:}
               teBlackOnBackdrop = teBlackText + teBackdrop;     {Black text on backdrop        }
```

```
teBlackOnWhite    = teBlackText + teWhiteBack;    {Black text on white             }
teBlackOnColor    = teBlackText + teColorBack;    {Black text on color             }
teColorOnBackdrop = teColorText + teBackdrop;     {Color text on backdrop          }
teColorOnWhite    = teColorText + teWhiteBack;    {Color text on white             }
teColorOnColor    = teColorText + teColorBack;    {Color text on color             }

                                            {Text alignment:                       }
teJustLeft    = 0;                          {  Left aligned (default)              }
teJustCenter  = 1;                          {  Centered                            }
teJustRight   =-1;                          {  Right aligned                       }
```

NewFieldRect

Create a new field. The new field is *not* activated.

C
```
pascal void NewFieldRect (short Field, const Rect *Bounds, Handle hStr,
              long Spec, short Just);
```

Pascal
```
procedure NewFieldRect (Field: INTEGER; Bounds: RECT; hStr: HANDLE;
              Spec: LONGINT; Just: INTEGER);
```

NewFieldRect is identical to the NewField routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

NewDialogField

Create a new field in a dialog using a dialog item's co-ordinates.

C
```
pascal void NewDialogField (short Field, Handle hStr, long Spec, short Just);
```

Pascal
```
procedure NewDialogField (Field: INTEGER; hStr: HANDLE; Spec: LONGINT;
              Just: INTEGER);
```

NewDialogField is identical to the NewField routine, except that the field is created in a dialog (a window opened with the LoadDialog routine, or one that had a dialog list attached with the LoadDialogList routine). The field's co-ordinates are obtained from the dialog item whose number matches the field number.

NewWideField

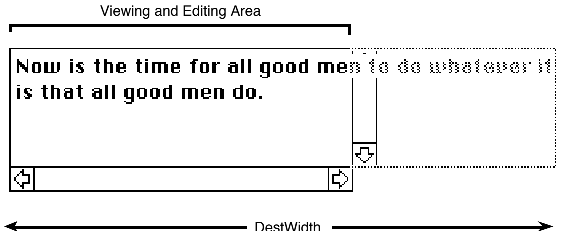
Create a new field with a horizontal scroll bar. The new field is *not* activated.

C
```
pascal void NewWideField (short Field, short left, short top, short right,
              short bottom, short DestWidth, Handle hStr, long Spec,
              short Just);
```

Pascal
```
procedure NewWideField (Field: INTEGER; left, top, right, bottom: INTEGER;
              DestWidth: INTEGER; hStr: HANDLE; Spec: LONGINT; Just: INTEGER);
```

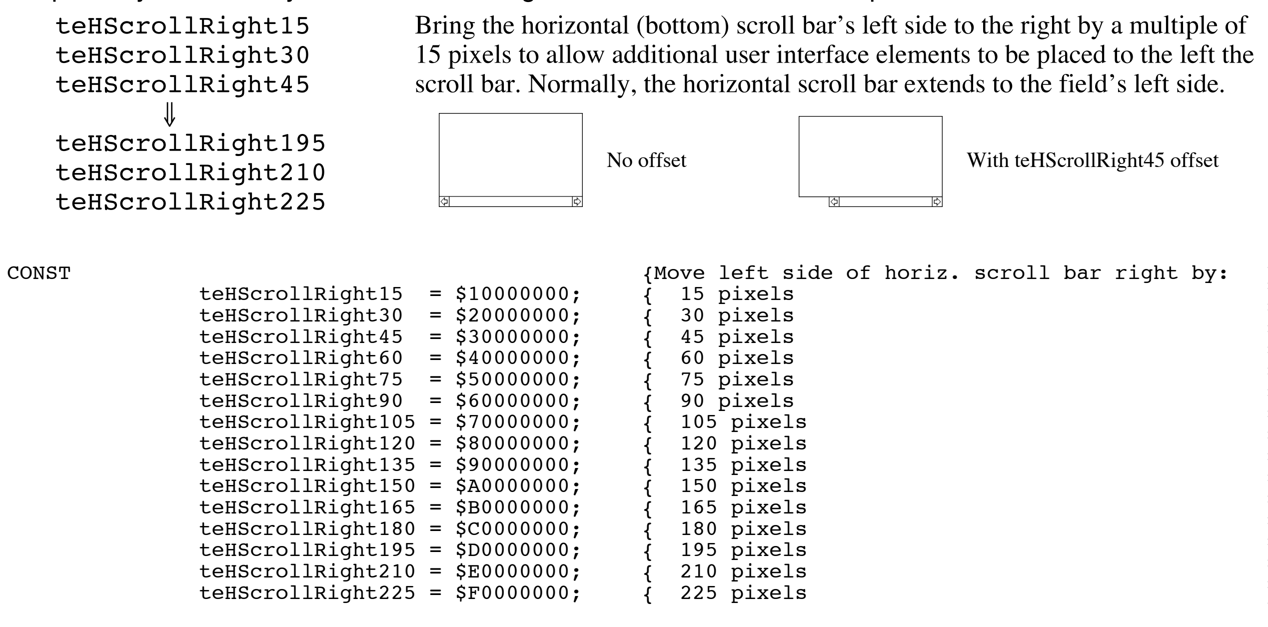
NewWideField is identical to the NewField routine except that it creates a field with a horizontal scroll bar. The scroll bar is created *outside* the field's co-ordinates.

DestWidth is an additional parameter that specifies the width in pixels at which word wrap takes place. Inside Macintosh refers to this as the *destination rectangle's* width. The co-ordinates specified by left, top, right and bottom are the viewing area inside which text can be seen and edited. DestWidth, on the other hand, is the width of the rectangle containing the text behind the scenes. DestWidth can have a value from 50 to 15000.



You can have Tools Plus calculate the destination rectangle's width by specifying a value of zero (0) for DestWidth. For field's whose right side is attached to the window's right edge, or if you have included the teAutoMoveSize option with the field's right side tracking the window's width, NewWideField creates a destination rectangle whose width is as wide as the field's width when the window is sized to its maximum width.

Spec specifies the appearance and behavior of the editing field. The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result. In addition to the values specified in NewField, you can also optionally offset the horizontal scroll bar's left side.



Optionally choose only one of the following horizontal scroll bar offset options…

```
teHScrollRight15
teHScrollRight30
teHScrollRight45
      ⇓
teHScrollRight195
teHScrollRight210
teHScrollRight225
```

Bring the horizontal (bottom) scroll bar's left side to the right by a multiple of 15 pixels to allow additional user interface elements to be placed to the left the scroll bar. Normally, the horizontal scroll bar extends to the field's left side.

No offset

With teHScrollRight45 offset

```
CONST                                       {Move left side of horiz. scroll bar right by:   }
              teHScrollRight15  = $10000000;     {  15 pixels                                     }
              teHScrollRight30  = $20000000;     {  30 pixels                                     }
              teHScrollRight45  = $30000000;     {  45 pixels                                     }
              teHScrollRight60  = $40000000;     {  60 pixels                                     }
              teHScrollRight75  = $50000000;     {  75 pixels                                     }
              teHScrollRight90  = $60000000;     {  90 pixels                                     }
              teHScrollRight105 = $70000000;     {  105 pixels                                    }
              teHScrollRight120 = $80000000;     {  120 pixels                                    }
              teHScrollRight135 = $90000000;     {  135 pixels                                    }
              teHScrollRight150 = $A0000000;     {  150 pixels                                    }
              teHScrollRight165 = $B0000000;     {  165 pixels                                    }
              teHScrollRight180 = $C0000000;     {  180 pixels                                    }
              teHScrollRight195 = $D0000000;     {  195 pixels                                    }
              teHScrollRight210 = $E0000000;     {  210 pixels                                    }
              teHScrollRight225 = $F0000000;     {  225 pixels                                    }
```

NewWideFieldRect

Create a new field with a horizontal scroll bar. The new field is *not* activated.

C

```
pascal void NewWideFieldRect (short Field, const Rect *Bounds,
             short DestWidth, Handle hStr, long Spec, short Just);
```

Pascal

```
procedure NewWideFieldRect (Field: INTEGER; Bounds: RECT;
             DestWidth: INTEGER; hStr: HANDLE; Spec: LONGINT; Just: INTEGER);
```

NewWideFieldRect is identical to the NewWideField routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

NewDialogWideField

Create a new field with a horizontal scroll bar in a dialog using a dialog item's co-ordinates.

C
```
pascal void NewDialogWideField (short Field, short DestWidth, Handle hStr,
            long Spec, short Just);
```

Pascal
```
procedure NewDialogWideField (Field: INTEGER; DestWidth: INTEGER;
            hStr: HANDLE; Spec: LONGINT; Just: INTEGER);
```

NewDialogWideField is identical to the NewWideField routine, except that the field is created in a dialog (a window opened with the LoadDialog routine, or one that had a dialog list attached with the LoadDialogList routine). The field's co-ordinates are obtained from the dialog item whose number matches the field number.

EmbedFieldInButton

Embed a field into a button or into the window's root control (Appearance Manager only).

C
```
pascal void EmbedFieldInButton (short Field, short ContainerButton);
```

Pascal
```
procedure EmbedFieldInButton (Field, ContainerButton: INTEGER);
```

The Appearance Manager lets you embed a control into a parent control such that when the parent is hidden or disabled, all embedded controls are similarly affected. All Tools Plus routines that load a dialog item list (LoadDialog, LoadSpecDialog, LoadDialogList, etc.) automatically embed controls at all times. EmbedFieldInButton lets you manually embed a field into a button, or into the window's root control. Note that the term "button" does not literally mean a button control. It means any control that is implemented as a button in Tools Plus. The most likely candidate is a Group Box control. If the Appearance Manager is not available, EmbedFieldInButton does nothing.

Field specifies the field number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, EmbedFieldInButton does nothing. Note that the only fields that can be embedded are those that are drawn using a CDEF (use the teSystemBody option when creating the field).

ContainerButton specifies the button number (from 1 to 511) into which *Field* is embedded. This control must exist in the current window, and it must be a "container" type control such as the Appearance Manager's Group Box. The field must fit entirely within the container control or EmbedFieldInButton does nothing. If a value of 0 is provided for a container button, *Field* is embedded into the window's root control.

Also see: EmbedFieldInScrollBar and SetAutoEmbed.

EmbedFieldInScrollBar

Embed a field into a scroll bar or into the window's root control (Appearance Manager only).

C
```
pascal void EmbedFieldInScrollBar (short Field, short ContainerScrollBar);
```

Pascal
```
procedure EmbedFieldInScrollBar (Field, ContainerScrollBar: INTEGER);
```

The Appearance Manager lets you embed a control into a parent control such that when the parent is hidden or disabled, all embedded controls are similarly affected. All Tools Plus routines that load a dialog item list (LoadDialog, LoadSpecDialog, LoadDialogList, etc.) automatically embed controls at all times. EmbedFieldInScrollBar lets you manually embed a field into a scroll bar, or into the window's root control. Note that the term "scroll bar" does not literally mean a scroll bar control. It means any control that is implemented as a scroll bar in Tools Plus. If the Appearance Manager is not available, EmbedFieldInScrollBar does nothing.

Field specifies the field number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, EmbedFieldInScrollBar does nothing. Note that the only fields that can be embedded are those that are drawn using a CDEF (use the teSystemBody option when creating the field).

ContainerScrollBar specifies the scroll bar number (from 1 to 511) into which *Field* is embedded. This control must exist in the current window, and it must be a "container" type control. The field must fit entirely within the container control or EmbedFieldInScrollBar does nothing. If a value of 0 is provided for a container scroll bar, *Field* is embedded into the window's root control.

Also see: EmbedFieldInButton and SetAutoEmbed.

GetFreeFieldNum

Get the first unused field number.

C
```
pascal short GetFreeFieldNum (void);
```

Pascal
```
function GetFreeFieldNum: INTEGER;
```

Some developers may prefer to write code that more closely resembles a traditional Macintosh application, in that creating an object returns a reference to it such as a handle or pointer. Instead of having to assign your own field number, GetFreeFieldNum returns the first unused (free) field number. Using this routine, you can assign an unused field number to a variable, then use that variable throughout your application without concern for the true field number.

GetFreeFieldNum returns the first free field number on the current window. If the current window doesn't belong to your application, if no windows are open, or if the maximum number of fields has already been created on the current window (no new ones can be created), GetFreeFieldNum returns a value of zero (0).

DeleteField

Delete a field.

C
```
pascal void DeleteField (short Field);
```

Pascal
```
procedure DeleteField (Field: INTEGER);
```

Field specifies the field number (from 1 to 32767) that is deleted from the current window. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, DeleteField does nothing.

If the field being deleted is the active field then it is deactivated before being deleted. If field was also in an active window that allows access to pull-down menus, the Edit menu's "Undo" item is changed to "Can't Undo" and is disabled along with the "Cut", "Copy", "Paste" and "Clear" items. Use KillField if you want to delete the field without removing its image from the window.

KillField

Delete a field without affecting its image on the window.

C
```
pascal void KillField (short Field);
```

Pascal
```
procedure KillField (Field: INTEGER);
```

KillField is identical to DeleteField except that it does not remove the field's image from the window. This routine is useful for scrolling fields in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetField repositions the field's co-ordinates without affecting its image (since ScrollRect has already moved it). KillField then deletes the fields that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

FieldDisplay

Hide or show a field.

C
```
pascal void FieldDisplay (short Field, Boolean Show);
```

Pascal
```
procedure FieldDisplay (Field: INTEGER; Show: BOOLEAN);
```

FieldDisplay hides or shows a field on the current window. The result is seen immediately. Use discretion with this routine since fields should be enabled and disabled to indicate if they are accessible by the user.

Field specifies the field number (from 1 to 32767) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, FieldDisplay does nothing.

Show indicates if the field is being hidden or displayed. The two constants that can be used for this flag are *on* and *off*. If the affected field is the window's active field, it is automatically deactivated before it is hidden. This will result in the loss of the edited text if you do not call SaveFieldString before hiding the field.

FieldIsVisible

Determine if a field is visible.

C
```
pascal Boolean FieldIsVisible (short Field);
```

Pascal
```
function FieldIsVisible (Field: INTEGER): BOOLEAN;
```

FieldIsVisible reports if a field is visible on the current window, or if it is hidden.

Field specifies the field number (from 1 to 32767) that is queried in the current window.

This routine's value returns *true* if the field is visible, and *false* if the field is hidden. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, FieldIsVisible returns *false*. This routine takes control embedding into account, so it will return *false* if the target field is embedded and its container control is hidden.

ObscureField

Hide a field without removing its image from the window.

C
```
pascal void ObscureField (short Field);
```

Pascal
```
procedure ObscureField (Field: INTEGER);
```

ObscureField hides a field on the current window without removing its image from the window. This routine is useful for scrolling fields in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetField repositions the field's co-ordinates without affecting its image (since ScrollRect has already moved it). ObscureField then hides the fields that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

Field specifies the field number (from 1 to 32767) that is hidden in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, ObscureField does nothing. If the affected field is the window's active field, it is automatically deactivated before it is hidden. This will result in the loss of the edited text if you do not call SaveFieldString before hiding the field.

GetFieldRect

Get a field's co-ordinates.

C
```
pascal void GetFieldRect (short Field, Rect *Bounds);
```

Pascal
```
procedure GetFieldRect (Field: INTEGER; var Bounds: RECT);
```

Field specifies the field number (from 1 to 32767) that is queried in the current window.

Bounds returns the field's bounding rectangle specified in the window's local co-ordinates. These co-ordinates match those used to create the field. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, Bounds returns with all co-ordinates set to zero (0).

SetFieldFontSettings

Set a field's font, size and style settings.

C
```
pascal void SetFieldFontSettings (short Field,
            short theFont, short theSize, Style theStyle);
```

Pascal
```
procedure SetFieldFontSettings (Field: INTEGER;
            theFont: INTEGER; theSize: INTEGER; theStyle: Style);
```

Field specifies the field number (from 1 to 32767) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if the field does not exist, SetFieldFontSettings does nothing. Otherwise, the change is seen immediately.

TheFont specifies the field's new font. The default is Chicago, which is represented by the systemFont constant.

TheSize specifies the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle specifies the field's new style. Special character constants defined by the Font Manager are bold, italic, underline and shadow. C programmers use the Font Manager's constants to specify a composite style, such as SetFieldFontSettings(1, geneva, 9, bold + outline) for bold and outlined, or SetFieldFontSettings(1, geneva, 9, 0) for plain text. Pascal programmers use the Font Manager's constants to specify a style set, such as

SetFieldFontSettings(1, geneva, 9, [bold, outline]) for bold and outlined, or SetFieldFontSettings(1, geneva, 9, [ ]) for plain text.

A field's font settings are set when a field is created, so this routine is not normally used by many applications.

GetFieldFontSettings

Get a field's font, size and style settings.

C
```
pascal void GetFieldFontSettings (short Field,
              short *theFont, short *theSize, Style *theStyle);
```

Pascal
```
procedure GetFieldFontSettings (Field: INTEGER;
              var theFont: INTEGER; var theSize: INTEGER; var theStyle: Style);
```

Field specifies the field number (from 1 to 32767) in the current window whose font settings are being retrieved. If the current window doesn't belong to your application, if no windows are open, or if *Field* specifies a field that does not exist, GetFieldFontSettings returns default values.

TheFont is the field's font number. The default is 0 which is represented by the systemFont constant.

TheSize is the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle is the field's font style. The default is plain text, which is represented by 0 in C and [ ] in Pascal.

SetFieldColors

Set a field's colors.

C
```
pascal void SetFieldColors (short Field,
              const RGBColor *TextColor, const RGBColor *BackColor);
```

Pascal
```
procedure SetFieldColors (Field: INTEGER;
              TextColor: RGBColor; BackColor: RGBColor);
```

Field specifies the field number (from 1 to 32767) in the current window whose colors are being set. If the current window doesn't belong to your application, or if no windows are open, SetFieldColors does nothing. Also, if *Field* specifies a field that does not exist, or if Color QuickDraw is unavailable or not used, SetFieldColors does nothing. The change is seen immediately, regardless if the field was originally created with colors or not.

TextColor is the color of the field's text.

BackColor is the field's background color upon which the text is drawn.

Normally, a field's colors are set when this field is created with NewField or NewFieldRect, so this routine would not be used by many applications.

GetFieldColors

Get a field's colors.

C
```
pascal void GetFieldColors (short Field,
            RGBColor *TextColor, RGBColor *BackColor);
```

Pascal
```
procedure GetFieldColors (Field: INTEGER;
            var TextColor: RGBColor; var BackColor: RGBColor);
```

Field specifies the field number (from 1 to 32767) in the current window whose colors are being retrieved. If the current window doesn't belong to your application, or if no windows are open, or if *Field* specifies a field that does not exist, GetFieldColors returns default color values.

TextColor is the color of the field's text. The default color is black.

BackColor is the field's background color upon which the text is drawn. The default color is white.

ActivateField

Activate a field.

C
```
pascal void ActivateField (short Field, short Selection);
```

Pascal
```
procedure ActivateField (Field, Selection: INTEGER);
```

Field specifies the field number (from 1 to 32767) being activated in the current window. ActivateField does nothing under any of these conditions: the current window doesn't belong to your application, no windows are open, the field does not exist in the current window, the field is disabled or hidden, or the field is a static text field.

Selection specifies which part of the text is selected. The constants that can be used to specify a field's text selection are *teSelectStart* (places insertion point at beginning of the field), *teSelectEnd* (places insertion point at end of the field), *teSelectAll* (selects all the text in the field) and *teSelectDefault* (selects text according to the field's default specifications).

Activating a field allows the user to interact with the field by typing on the keyboard. On an active window, the field acquires the keyboard focus making it the item that automatically processes keystrokes. Visually, this is indicated by having the field's text highlighted, or by a flashing caret. Additionally, if the Appearance Manager is available, the field is encompassed by a highlighting keyboard focus band to indicate that it has the focus. Using ActivateField in an active window removes the keyboard focus from any other object that may have the focus within the same window or any other active window such as a tool bar or floating palette. This action may deactivate another active field.

If the field being activated is in an active window that allows access to pull-down menus, the Edit menu's "Undo" item is changed to "Can't Undo" and is disabled, while the "Cut", "Copy", "Paste" and "Clear" items are enabled/disabled according to the insertion point or selection range as previously described in this chapter under "The Edit Menu."

Your application can activate virtually any field. This flexibility can lead to a confusing user interface by allowing the keyboard focus (active field) to jump between active windows. A good rule to observe is to activate a field only on a standard window (not a tool bar or a floating palette) when the window first opens. This sets up the default field for that window. At all other times, activate a field only in response to a user's actions.

Also see: HaveTabInFocus, TabToFocus, the doClickToFocus event, and ClickToFocus for other activating services.

```
CONST                          {Field text selection:                      }
          teSelectStart   =1;  {Insertion point at beginning of field      }
          teSelectEnd     =2;  {Insertion point at end of field            }
          teSelectAll     =3;  {Select the field's entire text             }
          teSelectDefault =0;  {Set selection as specified when creating field }
```

GetFieldSelection

Get a field's selection range.

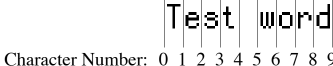
C
```
pascal void GetFieldSelection (short *SelStart, short *SelEnd);
```

Pascal
```
procedure GetFieldSelection (var SelStart: INTEGER; var SelEnd: INTEGER);
```

GetFieldSelection returns the start and end of the selection range in the current window's active field. If the current window doesn't belong to your application, if no windows are open, or if the current window does not have an active field, SelStart and SelEnd return with values of zero (0).

SelStart returns the beginning of the selection range. Character numbering starts from 0 and increases sequentially, therefore a value of 0 indicates the beginning of the selection range is just before the first character.

SelEnd returns the end of the selection range. Character numbering starts from 0 and increases sequentially, therefore a value of 6 indicates the end of the selection range is just before the seventh character, or just after the sixth character.



SetFieldSelection

Set a field's selection range.

C
```
pascal void SetFieldSelection (short SelStart, short SelEnd);
```

Pascal
```
procedure SetFieldSelection (SelStart, SelEnd: INTEGER);
```

SetFieldSelection sets the start and end of the selection range in the current window's active field. If the current window doesn't belong to your application, if no windows are open, or if the current window does not have an active field, SetFieldSelection does nothing. You will only need to use this routine if you need to specify selection on a character by character basis. ActivateField lets you activate a field and place the insertion point at the beginning or end of the field, or select the field's entire text.

SelStart defines the beginning of the selection range. Character numbering starts from 0 and increases sequentially, therefore a value of 0 indicates the beginning of the selection range is just before the first character.

SelEnd defines the end of the selection range. Character numbering starts from 0 and increases sequentially, therefore a value of 6 indicates the end of the selection range is just before the seventh character, or just after the sixth character.

DeactivateField

Deactivate the active field.

C
```
pascal void DeactivateField (void);
```

Pascal
```
procedure DeactivateField;
```

The active field, if one exists in the current window, is deactivated by this routine. If the current window doesn't belong to your application, or if no windows are open, or if a field is not active, DeactivateField does nothing. Once a field is deactivated, its edited text is discarded and replaced with the field's string. Therefore, if you want to save the field's edited text, call GetEditString or GetEditHandle and validate the text, then call SaveFieldString prior to deactivating the field.

If the deactivated field is in an active window that allows access to pull-down menus, the Edit menu's "Undo" item is changed to "Can't Undo" and is disabled along with the "Cut", "Copy", "Paste" and "Clear" items.

EnableField

Enable or disable a field.

C
```
pascal void EnableField (short Field, Boolean EnabledFlag);
```

Pascal
```
procedure EnableField (Field: INTEGER; EnabledFlag: BOOLEAN);
```

Field specifies the field number (from 1 to 32767) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, EnableField does nothing.

EnabledFlag specifies if the field is enabled or disabled. A disabled field cannot be activated by your application. The user can't tab to it, click in it, or otherwise change the disabled field's contents. A disabled field's text and its outlining box (if it has one) are grayed out. If you disable a field while it is active, the field is automatically deactivated and the field's edited text is not saved. The two constants that can be used for this purpose are *enabled* and *disabled*.

Also see: DisabledFieldLook and SetDisabledFieldLook.

```
CONST                          {Field state                          }
               enabled  = true;   {enable the field                     }
               disabled = false;  {disable the field                    }
```

FieldIsEnabled

Determine if a field is enabled or disabled.

C
```
pascal Boolean FieldIsEnabled (short Field);
```

Pascal
```
function FieldIsEnabled (Field: INTEGER): BOOLEAN;
```

Field specifies the field number (from 1 to 511) that is queried in the current window.

The routine's value returns *true* if the field is enabled, and *false* if the field is disabled. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, FieldIsEnabled returns *false*. FieldIsEnabled returns the field's enabled state as it is currently displayed, so if the field's window is inactive and has temporarily disabled the field, FieldIsEnabled returns *false*.

ClickToFocus

Process a mouse click that has occurred in a field or other item that wants the keyboard focus in an active window.

C
```
pascal void ClickToFocus (void);
```

Pascal
```
procedure ClickToFocus;
```

When Tools Plus reports a doClickToFocus event, it indicates that the user clicked the mouse in an a field or in another user interface element that wants the keyboard focus. If a field is active when this event is reported, it may be necessary to first validate the active field before responding to the click (before ClickToFocus is called). First, call GetEditString or GetEditHandle to retrieve the active field's edited text for validation. If the string cannot be validated, display an appropriate alert box and ignore the doClickToFocus event. If no error occurred, save the edited text as the

field's string by calling SaveFieldString, then process the click by calling ClickToFocus. The InitToolsPlus routine offers various options for automatically moving between fields and for saving fields' edited text.

ClickToFocus first deactivates the active field (if one exists) in any active window including a tool bar or floating palette. Similarly, it removes the keyboard focus from any user interface element that may have it. The routine then activates the required item and gives it the keyboard focus. In the case of an editing field, this means that an insertion point is placed at the point of the click. A double-click in the field and/or subsequent dragging is processed automatically.

If an editing field is activated in a window that allows access to pull-down menus, the Edit menu's "Undo" item is changed to "Can't Undo" and is disabled, while the "Cut", "Copy", "Paste" and "Clear" items are enabled/disabled as previously described in this chapter under "The Edit Menu". For other items that accept the keyboard focus, the Edit menu's items are disabled.

Warning: Between the time when the doClickToFocus event is reported and when your application calls ClickToFocus is called, observe the following rules:

- do not call Process1EventWhileBusy or ProcessToolboxEvent
- do not open or close any windows, including alerts and dialogs
- do not hide or show any windows
- do not activate any windows
- do not activate, deactivate, enable, disable or delete any user interface elements

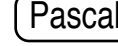
ClickToFocus depends on working with the same window that registered the doClickToFocus event, and will not work if your application gets or processes any subsequent events, or if you alter user interface elements.

HaveTabInFocus

Determine if the user is tabbing to the next or previous field or item that can take the keyboard focus.

C

```
pascal Boolean HaveTabInFocus (void);
```



```
function HaveTabInFocus: boolean;
```

When Tools Plus reports a doKeyDown or doAutoKey event, it simply indicates that Tools Plus is unable to automatically process the key stroke by automatically applying to the active editing field or other item that has the keyboard focus. Several conditions must be tested to determined if the user wants to tab to the next or previous item, and fortunately, HaveTabInFocus performs all the necessary tests.

If the routine returns with a value of *true*, then the user wants to tab to the next or previous item.

HaveTabInFocus looks for the following conditions to return with a *true* value:

- A doKeyDown or doAutoKey event was reported
- The user is in an active editing field or other item that has the keyboard focus
- You do not activate another window in response to a doKeyDown or doAutoKey event
- You do not change the active editing field number (or keyboard focus item number) in response to a doKeyDown or doAutoKey event
- The Tab key was typed (optionally, with the Shift key to tab to the previous field)
- The Command, Option, and Control key modifiers were all up when Tab was typed

It may be necessary to first process the active field before responding to the tab (before activating another field or using TabToFocus). First, call GetEditString or GetEditHandle to retrieve the active field's edited text for validation. If the string cannot be validated, display an appropriate alert box and ignore the doKeyDown or doAutoKey event. If no error occurred, save the edited text as the field's string by calling SaveFieldString, then process the tab by calling TabToFocus. The InitToolsPlus routine offers various options for automatically moving between fields and for saving fields' edited text.

Also see: TabToFocus.

TabToFocus

Tab to the next or previous field or item that can acquire the keyboard focus.

C
```
pascal void TabToFocus (void);
```

Pascal
```
procedure TabToFocus;
```

This routine is used in response to the HaveTabInFocus routine which detects that the user wants to tab to the next or previous field or item that can acquire the keyboard focus. When Tools Plus reports a doKeyDown or doAutoKey event (typing), and HaveTabInFocus returns with a value of *true* it indicates that the user wants to move to the previous/next keyboard focus item. Your application then uses TabToFocus to execute the request.

TabToFocus first deactivates the active field and/or removes the keyboard focus from any user interface element that has the focus in the active window. The routine then activates the previous or next user interface element that can acquire the keyboard focus. If an editing field is activated in a window that allows access to pull-down menus, the Edit menu's "Undo" item is changed to "Can't Undo" and is disabled, while the "Cut", "Copy", "Paste" and "Clear" items are enabled/disabled as previously described in this chapter under "The Edit Menu". For other items that accept the keyboard focus, the Edit menu's items are disabled.

TabToFocus considers the list of keyboard focus items to be cyclical, in that tabbing off the end of the list starts you at the beginning of the list and vice versa.

Also see: HaveTabInFocus.

✍ **Note**: If you are using standard editing fields (without the teSystemBody option) make sure your fields are *numbered* in the order in which you want to tab. If you are using the Appearance Manager and it's controls that can acquire the keyboard focus (fields with the teSystemBody option and other user interface elements that acquire the keyboard focus), make sure the items are *created* in the order in which you want to tab.

☞ **Warning**: Do not mix ordinary editing fields (those created without the teSystemBody option) with Appearance Manager items that acquire the keyboard focus because the user will not be able to tab through all the elements in one continuous cycle. If the user is editing a non Appearance Manager field, then he will be able to tab only between those fields. Similarly, if the user is working in an Appearance Manager control that has the keyboard focus such as a field creed with the teSystemBody option, list box or clock control, he will only be able to tab to other Appearance Manager controls.

☞ **Warning:** Between the time when your application calls HaveTabInFocus and when your application calls TabToFocus is called, observe the following rules:

- Do not call ProcessToolboxEvent or Process1EventWhileBusy
- Do not open or close any windows, including alerts and dialogs
- Do not hide or show any windows
- Do not activate any windows
- Do not activate, deactivate, enable, disable or delete any user interface elements

TabToFocus depends on working with the same window that registered the typing-related event, and may not work as expected if Tools Plus reports or processes any subsequent events or if you alter user interface elements.

GetEditString

Obtain a copy of the active field's edited text.

C
```
pascal void GetEditString (Str255 EditString);
```

Pascal
```
procedure GetEditString (var EditString: Str255);
```

EditString retrieves a copy of the edited text from the active field in the current window. The field's edited text is *not* altered by this routine. Although it is physically possible for the user to type more than 255 characters into a field that is not length limited, only the first 255 characters (the limit of a Pascal string) are retrieved by this routine. If the current window doesn't belong to your application or if it doesn't have an active field, or if no windows are open, GetEditString returns an empty string (string length of 0).

If the field is not length limited, the text retrieved by GetEditString may be longer than the field's associated string handle (as specified by the hStr handle when the field was created). If the field is length limited, EditString's length will never exceed the size limit of the field's associated string.

Also see: GetEditHandle, GetEditLength, GetFieldString, GetFieldHandle and GetFieldLength.

GetEditHandle

Obtain a handle to the active field's edited text.

C
```
pascal Handle GetEditHandle (void);
```

Pascal
```
function GetEditHandle: HANDLE;
```

This routine returns a handle to the edited text in the active field in the current window. Nil is returned if the current window does not have an active field or if the current window does not belong to your application.

The handle points to a generic block of text, and not a Pascal or C string. This is the actual text being edited by the user (it is not a copy of the text), so it is critically important that you do not change the text, resize, or deallocate the handle. If you must lock this handle, do so only on a temporary basis and make sure the handle is unlocked before using any Tools Plus routines.

Also see: GetEditString, GetEditLength, GetFieldString, GetFieldHandle and GetFieldLength.

GetEditLength

Determine the length of the active field's edited text.

C
```
pascal short GetEditLength (void);
```

Pascal
```
function GetEditLength: INTEGER;
```

This routine returns the length of the edited text in the active field in the current window. Zero (0) is returned if the current window does not have an active field or if the current window does not belong to your application.

The length refers to a generic block of text, and not a Pascal or C string. This is the number of characters in the actual text being edited by the user, and it does not have a Pascal string length-byte prefix or C string null-byte terminator. Its value can be up to 32767 characters if the field is not length limited. In length limited fields, this number won't exceed the field's maximum size.

Also see: GetEditString, GetEditHandle, GetFieldString, GetFieldHandle and GetFieldLength.

GetFieldString

Obtain a copy of a field's string.

C
```
pascal void GetFieldString (short Field, Str255 EditString);
```

Pascal
```
procedure GetFieldString (Field: INTEGER; var EditString: Str255);
```

Field specifies the editing field number (from 1 to 32767) that is queried in the current window.

EditString retrieves a copy of the specified field's string as a Pascal string. If the field is active, the string contains the text that existed before changes were made by the user. Only the first 255 characters are retrieved if the field's string is a C string and is larger than 255 characters. If the current window doesn't belong to your application, or if no windows are open, or if the editing field does not exist in the current window, an empty string is returned.

Also see: GetEditString, GetEditHandle, GetEditLength, GetFieldHandle and GetFieldLength.

GetFieldHandle

Obtain a handle to a field's string.

C
```
pascal Handle GetFieldHandle (short Field);
```

Pascal
```
function GetFieldHandle (Field: INTEGER): HANDLE;
```

Field specifies the editing field number (from 1 to 32767) that is queried in the current window.

This routine returns a handle to the specified field's Pascal string or C string. If the field is active, the handle points to text that existed before changes were made by the user. If the current window doesn't belong to your application, or if no windows are open, or if the editing field does not exist in the current window, a nil handle is returned. It is critically important that you do not change the text, resize, or deallocate the handle once your field has been created. If you must lock this handle, do so only on a temporary basis and make sure the handle is unlocked before using any Tools Plus routines. It is safest to paste text into the field using PasteIntoField or a similar Tools Plus routine.

Also see: GetEditString, GetEditHandle, GetEditLength, GetFieldString and GetFieldLength.

✍ **Note**: When you create a field using a nil text handle to automatically allocate a string handle, and you use the teBuffered option to give a large field high performance, Tools Plus does not allocate a Pascal or C string. It allocates a generic text handle that is the exact length of the text. It is not prefixed with a Pascal-styled length byte or null-terminated like a C string. GetFieldHandle returns this generic handle.

☞ **Warning**: When you create a field using a nil text handle to automatically allocate a string handle, and you use the teBuffered option to give a large field high performance, Tools Plus performs some memory swapping when a field is activated or deactivated. Therefore, your handle to the field's unedited text is valid until the next time the file field is activated or deactivated.

GetFieldLength

Determine the length of a field's string.

C
```
pascal short GetFieldLength (short Field);
```

Pascal
```
function GetFieldLength (Field: INTEGER): INTEGER;
```

Field specifies the editing field number (from 1 to 32767) that is queried in the current window.

This routine returns the length of the specified field's string (it excludes a Pascal string's length-byte prefix or a C string's null termination byte). If the field is active, the length is for the text that existed before changes were made by the user. Its value can be up to 32767 characters in a C string, or 255 characters in a Pascal string. In length limited fields, this number won't exceed the maximum size of the field. If the current window doesn't belong to your application, or if no windows are open, or if the editing field does not exist in the current window, zero is returned.

Also see: GetEditString, GetEditHandle, GetEditLength, GetFieldString and GetFieldHandle.

FieldIsEmpty

Determine if the specified field is empty.

C
```
pascal Boolean FieldIsEmpty (short Field);
```

Pascal
```
function FieldIsEmpty (Field: INTEGER): BOOLEAN;
```

Field specifies the editing field number (from 1 to 32767) that is queried in the current window.

The routine's value returns *true* if the field is empty (string length is zero). A non-zero string length returns a value of *false*. If the specified field is the window's active editing field (even though the window itself may not be active at the time), the function is performed on the field's *edited* text. Otherwise, the function is performed on the field's string. If the current window doesn't belong to your application, or if no windows are open, or if the editing field does not exist in the current window, FieldIsEmpty returns with a value of *true*.

SaveFieldString

Save the active field's edited text as the field's associated string.

C
```
pascal void SaveFieldString (void);
```

Pascal
```
procedure SaveFieldString;
```

SaveFieldString is used to save an active field's edited text by copying it into the field's associated string. This action occurs in the current window. If the current window doesn't belong to your application, or if no windows are open, or if a field is not active in the current window, SaveFieldString does nothing.

When SaveFieldString is called, the field's string handle (hStr) dictates the maximum number of characters that can be saved from the edited text. For example, if the field uses an Str255 (255 character Pascal string), up to 255 characters of edited text are saved in the field's associated string. These settings are established when a field is created. If the field is length limited, the edited text complies to these constraints by physically preventing the user from typing characters that would exceed the field's limit.

EditFldWindowNumber

Get the window number of the window containing your application's active editing field.

C
```
pascal short EditFldWindowNumber (void);
```

Pascal
```
function EditFldWindowNumber: INTEGER;
```

This routine returns the window number of the window containing the active editing field in your application. If your application does not have a tool bar or floating palettes, this window will either be the active window (frontmost), or it will be zero (0) when there is no active field or no windows are open. When a tool bar and/or floating palettes are used, this window can potentially be any of the active windows (tool bar, any floating palette, or the active standard window).

ActiveFieldNumber

Determine the active field number.

C
```
pascal short ActiveFieldNumber (void);
```

Pascal
```
function ActiveFieldNumber: INTEGER;
```

The routine's value returns the active field number in the current window. If the current window does not belong to your application, or if no windows are open, or if no field is active in the current window, ActiveFieldNumber returns a value of zero (0). If you want to determine which window contains your application's active field, use the EditFldWindowNumber routine.

FieldLengthLimit

Turn field length limiting on or off.

C
```
pascal void FieldLengthLimit (Boolean Limits);
```

Pascal
```
procedure FieldLengthLimit (Limit: BOOLEAN);
```

Limit specifies if subsequently created fields are length limited or not. The boolean constants "on" or "off" may be used.

Length limiting is an enhancement supported by Tools Plus. It prevents a field's edited text from exceeding a fixed length. This is done by preventing additional characters from being typed once the field has reached its limit, and by truncating text (if necessary) after a "Paste" command is executed. The user is beeped when excess characters are typed instead of accepting the key-strokes.

If a field is length limited, the limit is set to the maximum length of the field's string (referenced via the hStr handle) as described in the NewField routine.

A field takes on its limited/unlimited status when it is created by the NewField routine, depending on FieldLengthLimit's setting. When FieldLengthLimit(true) has been set, subsequently created fields are length limited. When FieldLengthLimit(false) is in effect, subsequently created fields will not be length limited. You can achieve the same thing on a field-by-field basis by adding the teLengthLimit constant when creating a field.

For the sake of consistency, all fields on a window should either be limited or unlimited. Length limiting works best, in a visual sense, when a mono-spaced (non-proportional) font is used and the field's width is long enough to contain the maximum number of characters. In this way, the user is limited to the number of characters that are visible in a field.

FieldLengthLimit is set to *false* when Tools Plus is initialized.

Programming Tips:

1 To help you keep track of which fields have adopted length limiting, use FieldLengthLimit(on) immediately before creating a set of fields that are length limited. After the fields are created, use FieldLengthLimit(off) to make subsequently created fields non-limited (as per the default).

SetFieldLengthLimit

Set field length limiting for an existing field.

C
```
pascal void SetFieldLengthLimit (short Field, short NewLimit);
```

Pascal
```
procedure SetFieldLengthLimit (Field, NewLimit: INTEGER);
```

See the FieldLengthLimit routine for details about what "length limiting" is, and related information.

Field specifies the editing field number (from 1 to 32767) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, FieldLengthLimit does nothing

NewLimit specifies the number of characters that the user can type or paste into the editing field. If the specified new limit is smaller than the number if characters that are currently in the field, or the number of characters that the user is editing in the field, then the value of *NewLimit* is automatically adjusted to account for the current number of characters in the field.

DynamicFieldHandles

Turn fields' automatic handle resizing on or off.

C
```
pascal void DynamicFieldHandles (Boolean Resize);
```

Pascal
```
procedure DynamicFieldHandles (Resize: BOOLEAN);
```

Resize specifies if subsequently created fields will have their string handles automatically resized or not. The boolean constants "on" or "off" may be used. If a field is set to resize its string handle, the handle grows and shrinks to reflect the amount of text it contains.

A field takes on its dynamic or static status when it is created by the NewField routine, depending on DynamicFieldHandles' setting. When DynamicFieldHandles(true) has been set, subsequently created fields have their string handles automatically resized. When DynamicFieldHandles(false) is in effect, subsequently created fields have fixed size string handles. You can achieve the same thing on a field-by-field basis by adding the teResizeHdl constant when creating a field.

DynamicFieldHandles is set to *false* when Tools Plus is initialized.

DisabledFieldLook

Set the appearance and behavior for disabled fields.

C
```
pascal void DisabledFieldLook (long DimFieldSpec);
```

Pascal
```
procedure DisabledFieldLook (DimFieldSpec: LONGINT);
```

Field disabling is an enhancement supported by Tools Plus. It prevents a user from changing a field's contents or activating a field. A field takes on its "disabled characteristics" when it is created by the NewField routine. For the sake of consistency, all fields in an application should have the same characteristics when disabled, so it's a good idea

to call DisabledFieldLook *once* early in your application before any fields are created.

DimFieldSpec specifies the appearance and behavior of subsequently created fields when they are disabled. The value for this 4-byte long integer is specified by adding a set of constants to obtain the desired result as illustrated below. By default, all options are off.

`teNeverDimBWText` — Do not dither disabled text on a monochrome monitor. By default, disabled text is dithered on a black and white monitor. Fine fonts such as Geneva and Helvetica become unreadable when dithered, so you may want to use this option when appropriate.

Examples of enabled and disabled fields using large and small fonts on a black and white monitor

`teNeverDimColorText` — Do not change the appearance of a disabled field's text when displayed on a color or gray scale monitor. By default, disabled text is dimmed on a color or gray scale monitor in System 7 or later, and dithered in System 6 or earlier. This is consistent with Apple's controls.

`teColSys6Text` — Dim disabled text on a color or gray scale monitor in System 6 or earlier. By default, disabled text is dithered on System 6 or earlier. Using this option makes Macs running older system versions look more like System 7. This setting has no effect on Macs running System 7 or later.

`teNeverDimBWBox` — Don't dither a disabled field's outline box (if it has one) when it is displayed on a black and white monitor. By default, the outline box is dithered using a gray pattern on a black and white monitor.

`teNeverDimColorBox` — Do not change the appearance of a disabled field's outline box (if it has one) when it is displayed on a color or gray scale monitor. By default, a disabled field's outline is dimmed on a color or gray scale monitor in System 7 or later, and dithered in System 6 or earlier. This is consistent with Apple's controls.

`teColSys6Box` — Dim a disabled field's outline on a color or gray scale monitor in System 6 or earlier. By default, disabled objects are dithered on System 6 or earlier. Using this option makes Macs running older system versions look more like System 7. This setting has no effect on Macs running System 7 or later.

`teClickBeep` — Beep when a disabled field is clicked by the user. By default, nothing happens when a disabled field is clicked. You may want to use this option if you have turned off all visual cues for disabling and you want an audible cue to indicate that the clicked field cannot be activated.

`teDfltDisabledLook` — Use this constant alone to restore all settings back to their default values (all off).

Also see: SetDisabledFieldLook.

```
CONST                                            {Behavior and Appearance Specs for disabled   }
                                                 {  fields:                                    }
               teNeverDimBWText    = $0001;      {Never dim text (B&W)                         }
               teNeverDimColorText = $0002;      {Never dim text (color)                       }
               teColSys6Text       = $0004;      {Colorize System 6 text                       }
               teNeverDimBWBox     = $0008;      {Never dim outline (B&W)                      }
               teNeverDimColorBox  = $0010;      {Never dim outline (color)                    }
               teColSys6Box        = $0020;      {Colorize Sys 6 outline                       }
               teClickBeep         = $0040;      {Beep when disabled                           }
               teDfltDisabledLook  = $0000;      {Use default settings                         }
```

Programming Tips:

1. If your application displays text that changes but is not editable (such as status or feedback information), you can use an editing field for this purpose. First you create a field without an outline box. The field's text should not dim (add teNeverDimBWText and teNeverDimColorText when specifying the disabled field look). Then use PasteIntoField to change the field's text.
2. To create a non-editable field with scroll bars, use teNeverDimBWText + teNeverDimColorText + teNeverDimBWBox + teNeverDimColorBox. The field will look normal and will be scrollable but will not be editable.
3. To help you keep track of which fields have adopted a certain appearance when disabled, use DisabledFieldLook immediately before creating a set of fields that need a non-default look. After the fields are created, set DisabledFieldLook back to the default value after by using the teDfltDisabledLook constant.

SetDisabledFieldLook

Set the appearance and behavior for a disabled field.

C
```
pascal void SetDisabledFieldLook (short Field, long DimFieldSpec);
```

Pascal
```
procedure SetDisabledFieldLook (Field: INTEGER; DimFieldSpec: LONGINT);
```

This routine is similar to DisabledFieldLook except that it sets the disabled appearance and behavior for a single field. The change is seen immediately if the field is disabled when calling this routine.

Field specifies the editing field number (from 1 to 32767) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the editing field does not exist in the current window, SetDisabledFieldLook does nothing.

DimFieldSpec specifies the appearance and behavior of subsequently created fields when they are disabled. The value for this 4-byte long integer is specified by adding a set of constants to obtain the desired result as illustrated below.

`teNeverDimBWText` — Do not dither disabled text on a monochrome monitor. By default, disabled text is dithered on a black and white monitor. Fine fonts such as Geneva and Helvetica become unreadable when dithered, so you may want to use this option when appropriate.

Large Large Small Font Small Font

Examples of enabled and disabled fields using large and small fonts on a black and white monitor

`teNeverDimColorText` — Do not change the appearance of a disabled field's text when displayed on a color or gray scale monitor. By default, disabled text is dimmed on a color or gray scale monitor in System 7 or later, and dithered in System 6 or earlier. This is consistent with Apple's controls

`teColSys6Text` — Dim disabled text on a color or gray scale monitor in System 6 or earlier. By default, disabled text is dithered on System 6 or earlier. Using this option makes Macs running older system versions look more like System 7. This setting has no effect on Macs running System 7 or later.

`teNeverDimBWBox` — Don't dither a disabled field's outline box (if it has one) when it is displayed on a black and white monitor. By default, the outline box is dithered using a gray pattern on a black and white monitor.

`teNeverDimColorBox` — Do not change the appearance of a disabled field's outline box (if it has one) when it is displayed on a color or gray scale monitor. By default, a disabled field's outline is dimmed on a color or gray scale monitor in System 7 or later, and dithered in System 6 or earlier. This is consistent with Apple's controls.

`teColSys6Box`	Dim a disabled field's outline on a color or gray scale monitor in System 6 or earlier. By default, disabled objects are dithered on System 6 or earlier. Using this option makes Macs running older system versions look more like System 7. This setting has no effect on Macs running System 7 or later.
`teClickBeep`	Beep when a disabled field is clicked by the user. By default, nothing happens when a disabled field is clicked. You may want to use this option if you have turned off all visual cues for disabling and you want an audible cue to indicate that the clicked field cannot be activated.
`teDfltDisabledLook`	Use this constant alone to restore all settings back to their default values (all off).

```
CONST                                         {Behavior and Appearance Specs for disabled   }
                                              {  fields:                                    }
            teNeverDimBWText    = $0001;      {Never dim text (B&W)                         }
            teNeverDimColorText = $0002;      {Never dim text (color)                       }
            teColSys6Text       = $0004;      {Colorize System 6 text                       }
            teNeverDimBWBox     = $0008;      {Never dim outline (B&W)                      }
            teNeverDimColorBox  = $0010;      {Never dim outline (color)                    }
            teColSys6Box        = $0020;      {Colorize Sys 6 outline                       }
            teClickBeep         = $0040;      {Beep when disabled                           }
            teDfltDisabledLook  = $0000;      {Use default settings                         }
```

PasteIntoField

Paste text into a field.

C
```
pascal void PasteIntoField (short Field, const Str255 Text, Boolean Replace);
```

Pascal
```
procedure PasteIntoField (Field: INTEGER; Text: STRING; Replace: BOOLEAN);
```

Some applications need to paste text directly into a field under their own control. An example of this is an operation that lets the user select an item from a List Box, then the selected item is pasted into a field as though the user had typed it. Use this routine judiciously, because indiscriminate pasting can be detrimental to a good user interface. Text can be pasted into an active or inactive field. After pasting into an active field, the insertion point is placed after the last character of the pasted text. If the pasting occurred in the active field within a window, then the Edit menu, if one exists, has its "Undo" item set to "Undo Paste" thereby allowing the pasting to be undone.

Field specifies the field number (from 1 to 32767) into which the text is pasted in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, PasteIntoField does nothing.

Text specifies the string that is pasted into the specified field. If an empty string (length of 0) is specified, the field's affected text is cleared. When pasting into a length limited field, text is pasted one character at a time (although very quickly) and stops when the field is full. This text is automatically filtered if the field is using a filter.

Replace specifies if the pasted text is inserted into the field or if it replaces the field's entire contents. With a value of *teInsert*, the field's selected range of characters is removed (if a selection range exists) and the new text is inserted at the insertion point. If a value of *teReplace* is used, the field's entire text is replaced with the contents of the supplied string. When pasting into an inactive field, the field's contents are replaced regardless of the value of the Replace parameter.

Also see: PastePintoField and PasteHIntoField.

```
CONST                          {Types of pasting                                        }
            teReplace = true;  {Replace field's contents with specified text            }
            teInsert  = false; {Insert specified text at the insertion point            }
```

PastePIntoField

Paste text into a field using a pointer.

C
```
pascal void PastePIntoField (short Field, Ptr Text, short TextLength,
            Boolean Replace);
```

Pascal
```
procedure PastePIntoField (Field: INTEGER; Text: PTR; TextLength: INTEGER;
            Replace: BOOLEAN);
```

PastePIntoField is similar to PasteIntoField in that it pastes text into a field, however this routine pastes text from a pointer instead of a string.

Field specifies the field number (from 1 to 32767) into which the text is pasted in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the field does not exist in the current window, PastePIntoField does nothing.

Text is a pointer to the text that is being pasted. It can also be the address of a C string or any other structure where the text begins at the first byte.

TextLength specifies the number of characters being pasted. It may be in the range of 0 to 32767 characters. If 0 is specified, the field's affected text is cleared. When pasting into a length limited field, text is pasted one character at a time (although very quickly) and stops when the field is full. This text is automatically filtered if the field is using a filter.

Replace specifies if the pasted text is inserted into the field or if it replaces the field's entire contents. With a value of *teInsert*, the field's selected range of characters is removed (if a selection range exists) and the new text is inserted at the insertion point. If a value of *teReplace* is used, the field's entire text is replaced with the contents of the supplied string. When pasting into an inactive field, the field's contents are replaced regardless of the value of the Replace parameter.

```
CONST                           {Types of pasting                                     }
               teReplace = true;   {Replace field's contents with specified text            }
               teInsert  = false;  {Insert specified text at the insertion point            }
```

PasteHIntoField

Paste text into a field using a handle.

C
```
pascal void PasteHIntoField (short Field, Handle Text, short TextLength,
            Boolean Replace);
```

Pascal
```
procedure PasteHIntoField (Field: INTEGER; Text: HANDLE; TextLength: INTEGER;
            Replace: BOOLEAN);
```

PasteHIntoField is similar to PastePIntoField except that it accepts a handle to the text instead of a pointer. Your application can pass either a locked or unlocked handle

MoveField

Move a field to a new location on the window.

C
```
pascal void MoveField (short Field, short toHoriz, short toVert);
```

Pascal
```
procedure MoveField (Field, toHoriz, toVert: INTEGER);
```

Field specifies the field number (from 1 to 32767) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Field* specifies a field that does not exist, MoveField does nothing. The change is seen immediately providing that the field is not hidden. The field's width and height are not changed.

ToHoriz is the new horizontal co-ordinate at which the left side of the field appears.

ToVert is the new vertical co-ordinate at which the top of the field appears.

Also see: SizeField and MoveSizeField.

OffsetField

Change a field's co-ordinates without affecting its image on the window.

C
```
pascal void OffsetField (short Field, short distHoriz, short distVert);
```

Pascal
```
procedure OffsetField (Field, distHoriz, distVert: INTEGER);
```

When you scroll an area that contains fields, first use ScrollRect to scroll the pixel image containing the affected objects in the window. OffsetField is used to offset a field's co-ordinates without altering its image (since ScrollRect has already done so). At this point, the field's co-ordinates match the scrolled image of the field. ObscureField or KillField can be used to hide or delete fields that are scrolled out of view.

Field specifies the field number (from 1 to 32767) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Field* specifies a field that does not exist, OffsetField does nothing.

DistHoriz and *distVert* specify the horizontal and vertical amount by which the field's co-ordinates are offset. Positive numbers are right and down. The field's co-ordinates are updated but no change is seen.

Scrolling fields

Your application can create a matrix of fields and treat them as though they were "cells" in a spread-sheet. For example, fields could be aligned to represent columns and rows as illustrated below:

Customer	Name	Telephone
12413	John Smith	

Seven rows, each with 3 fields, is a total of 21 fields. In fact, any combination of lines and columns can be used as long as no more than 32767 fields are visible at a time.

By scrolling up or down, the user is given the impression that there are actually more lines available than those that are currently visible. The scrolling process is accomplished by the following series of steps.

1. **Shift Lines**: Use the toolbox's ScrollRect routine to shift the lines to their new position. They should be shifted to the position where they will appear after scrolling, typically a multiple of a field's height.
2. **Delete Old Fields**: Use KillField to delete fields that are scrolled out of view. KillField does not affect the window's image.

3. **Shift Fields**: Use OffsetField to change co-ordinates of each remaining field that is visible after scrolling.
4. **Create New Fields**: Create new fields that are now in view due to scrolling.

Note that within the steps outlined above, you will have to decide how your application deals with the active field and its edited text, since it may be scrolled out of view during this process. See the tutorials folder for a working example of scrolled a "field list."

SizeField

Change a field's size.

C
```
pascal void SizeField (short Field, short width, short height);
```

Pascal
```
procedure SizeField (Field, width, height: INTEGER);
```

SizeField changes a field's width and/or height without altering the field's top or left co-ordinate. The change is seen immediately providing that the field is not hidden.

Field specifies the field number (from 1 to 32767) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Field* specifies a field that does not exist, SizeField does nothing.

Width and *height* specify the field's new width and height in pixels. You must specify a minimum width of 5 and a minimum height of 8 or SizeField does nothing. The *height* parameter is ignored if you are changing a single line field.

Also see: MoveField and MoveSizeField.

MoveSizeField

Change a field's co-ordinates.

C
```
pascal void MoveSizeField (short Field,
            short left, short top, short right, short bottom);
```

Pascal
```
procedure MoveSizeField (Field, left, top, right, bottom: INTEGER);
```

MoveSizeField changes any of the field's four co-ordinates. The change is seen immediately providing that the field is not hidden. This routine combines the functions of MoveField and SizeField.

Field specifies the field number (from 1 to 32767) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Field* specifies a field that does not exist, MoveSizeField does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in local co-ordinates that determines the field's size and location in the window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). You must specify a minimum width of 5 and a minimum height of 8 or MoveSizeField does nothing. The *bottom* parameter is ignored if you are changing a single line field to prevent changing the field's height.

Also see: GetFieldRect.

MoveSizeFieldRect

Change a field's co-ordinates.

C
```
pascal void MoveSizeFieldRect (short Field, const Rect *Bounds);
```

Pascal
```
procedure MoveSizeFieldRect (Field: INTEGER; Bounds: RECT);
```

MoveSizeFieldRect is identical to the MoveSizeField routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

AutoMoveSizeField

Specify how a field is automatically moved and/or resized as its window's size is changed.

C
```
pascal void AutoMoveSizeField (short Field,
            Boolean left, Boolean top, Boolean right, Boolean bottom);
```

Pascal
```
procedure AutoMoveSizeField (Field: INTEGER;
            left, top, right, bottom: BOOLEAN);
```

Field specifies the field number (from 1 to 32767) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Field* specifies a field that does not exist, AutoMoveSizeField does nothing.

The *left*, *top*, *right* and *bottom* parameters specify if that side of the field is automatically adjusted when the window's size changes. These setting are applied to the field and are used the next time the window's size changes:

left	Does the field's left side track the window's right edge?
top	Does the field's top track the window's bottom edge?
right	Does the field's right side track the window's right edge?
bottom	Does the field's bottom track the window's bottom edge?

You can think of each *false* value as locking that side of the field to a fixed co-ordinate regardless of the window's size (this is the default). Each *true* value establishes a fixed distance between that side of the field and the window's edge. For example, setting only *left* and *right* to *true* makes the field move horizontally as the window widens and narrows, but the field does not move vertically when the window's height changes.

If you are setting these values identically for a group of objects, use AutoMoveSize to define the settings then add the appropriate *x*AutoMoveSize constant (such as teAutoMoveSize for fields) to the objects' spec as they are created. The objects will adopt the settings specified by the AutoMoveSize routine.

☞ **Warning**: Make sure that you resize objects in a way that makes sense. Don't allow a window to shrink down to a size where objects become unusable or disappear altogether.

ResetFieldScrolling

Scroll the text in an editing field to its default position.

C
```
pascal void ResetFieldScrolling (short Field);
```

Pascal
```
procedure ResetFieldScrolling (Field: INTEGER);
```

By default, an editing field's text is automatically reset to its default scrolling orientation when it is deactivated, that is:
- top-left for left-aligned fields
- top-center for centered fields
- top-right for right aligned fields

Your application can override this behavior when the field is created by adding the teNoResetOnDeactivate option to the field's specification. When this is done and the user tabs or clicks in another field, the current field stays scrolled exactly as the user left it.

Field specifies the editing field number (from 1 to 32767) that is affected. If the current window doesn't belong to your application, or if no windows are open, or if the editing field does not exist in the current window, ResetFieldScrolling does nothing. This routine does not work on editing fields created with the Appearance Manager's Edit Text control (created with the teSystemBody option).

NewFieldFilter

Create a new field filter independently of any fields.

C
```
pascal short NewFieldFilter (const Str255 Chars, long FilterSpec);
```

Pascal
```
function NewFieldFilter (Chars: STRING; FilterSpec: LONGINT): INTEGER;
```

Chars specifies the character set that makes up a single, unique filter. By default, the filter is sensitive to case and diacritical marks. Later, when your application applies this filter to a field or a set of fields (using CurrentFieldFilter), it specifies if the characters in the filter are allowed or disallowed in the field. Tools Plus automatically processes the following special characters by either carrying out their correct action as is the case with the right arrow, or by filtering out the character as is the case with the escape key. You do not have to specify any of these special characters in your filter. They are ASCII characters 0-16, 21-31 and 127:

EscClearKey	HelpKey	LeftArrowKey
FKey	DeleteFwdKey	RightArrowKey
BackSpaceKey	HomeKey	UpArrowKey
TabKey	EndKey	DownArrowKey
ReturnKey	PageUpKey	
EnterKey	PageDownKey	

FilterSpec specifies several options for the filter. The value for this 4-byte long integer is specified by adding a set of constants to obtain the desired result as illustrated below. By default, all options are off.

`teIgnoreCase` — The filter automatically adds upper and lower case characters to those you specify. For example, if you specify "AbC" the field's character set it expanded to "ABCabc".

`teIgnoreDiac` — The filter automatically adds diacritically equivalent characters to those you specify. For example, if you specify "bce" the field's character set it expanded to "bcçeéèêë".

`teShiftCaseUp` — As characters are added to fields using this filter, they are converted to their upper case equivalent.

`teShiftCaseDown` — As characters are added to fields using this filter, they are converted to their lower case equivalent. Do not use this option in conjunction with teShiftCaseUp.

The routine's value returns with a unique Filter Reference Number that is in the range of 1 to 32767. If your application tries to create two identical filters, the second attempt will return the prior Filter Reference Number without creating a duplicate filter.

Although NewFieldFilter does not fragment memory, it is a good idea to create all your filters early in your application before memory is fragmented at all.

Also see: CurrentFieldFilter and SetFieldFilter to apply a filter to a field.

Note: The teIgnoreDiac option includes all diacritically equivalent characters, even though those characters may not be present in a field's font. Helvetica, for example, has the character "Ÿ" (upper case "ÿ") while Chicago does not. If this is a concern, manually specify the characters in the filter instead of using the teIgnoreDiac option.

```
CONST                                   {Filter options                                    }
               teIgnoreCase    = $0001;  {Disable case sensing                              }
               teIgnoreDiac    = $0002;  {Disable diacritical sensing                       }
               teShiftCaseUp   = $0004;  {Shift typed/pasted characters to upper case       }
               teShiftCaseDown = $0008;  {Shift typed/pasted characters to lower case       }
```

CurrentFieldFilter

Apply a filter to subsequently created editing fields.

C
```
pascal void CurrentFieldFilter (short FilterRefNum);
```

Pascal
```
procedure CurrentFieldFilter (FilterRefNum: INTEGER);
```

FilterRefNum specifies the Filter Reference Number of the filter that is used by subsequently created editing fields. If zero (0) is used, or if the specified filter does not exist, subsequently created fields are not filtered. A filter's reference number is returned when the filter is created by using the NewFieldFilter routine. Text pasted into static text fields is always unfiltered.

A field adopts a filter when it is created by the NewField routine, depending on CurrentFieldFilter's setting. Specifying a positive filter number *allows* only the characters contained in the filter's character set. Using a negative version of the same reference number *disallows* the characters contained in the filter's character set. The following example disallows the digits 0 through 9:

```
myFilter := NewFieldFilter ('0123456789', 0);    {Create new filter & return reference number    }
CurrentFieldFilter (-myFilter);                  {Disallow chars in filter specified by myFilter }
```

CurrentFieldFilter is set to 0 when Tools Plus is initialized.

Also see: NewFieldFilter to create new filters.

SetFieldFilter

Apply a filter to an editing field.

C
```
pascal void SetFieldFilter (short Field, short FilterRefNum);
```

Pascal
```
procedure SetFieldFilter (Field: INTEGER; FilterRefNum: INTEGER);
```

Field specifies the field number (from 1 to 32767) in the current window that will adopt the specified filter. If the current window doesn't belong to your application, or if no windows are open, or if the specified field number does not exist, SetFieldFilter does nothing.

FilterRefNum specifies the Filter Reference Number for the filter that is applied to the field. A filter's reference number is returned when the filter is created by using the NewFieldFilter routine. Specifying a positive number allows

only the characters in the filter. Specifying a negative number disallows the characters in the filter. The field becomes unfiltered if zero (0) is used for this parameter. If the specified Filter Reference Number does not exist, SetFieldFilter does nothing.

SetFieldFilter establishes the relationship between the field and a filter. Subsequently, the field uses that filter to filter unwanted characters as they are typed or pasted into the field. This routine does not filter characters that are already in the field when SetFieldFilter is used.

Also see: NewFieldFilter to create new filters.

SetTENoUndoThresh

Specify the minimum free memory required after "undo" services are set up.

C
```
pascal void SetTENoUndoThresh (long Threshold);
```

Pascal
```
procedure SetTENoUndoThresh (Threshold: LONGINT);
```

Threshold specifies the minimum amount of contiguous memory you want your application to have after Tools Plus's Undo/Redo services have been set up (which may consume up to 64K of memory). The services are set up just before a change is made to an active editing field.

If the largest piece of continuous memory is smaller than this specified value after the Undo/Redo services have been set up, the user is warned with a message stating "Low memory… Continue without 'Undo/Redo'?" A "Continue" button lets the user continue without the Undo/Redo services being set up (i.e., the Edit menu's "Undo…" item is disabled and set to "Can't Undo"). A "Cancel" button lets the user cancel the editing operation without making any changes.

This threshold is set to a reasonable default value when Tools Plus is initialized, so your application benefits from low-memory protection without you having to explicitly do anything. You can change the message displayed by Tools Plus as described in the Multiple Languages chapter.

SetTENoEditThresh

Specify minimum free memory below which text editing is disallowed.

C
```
pascal void SetTENoEditThresh (long Threshold);
```

Pascal
```
procedure SetTENoEditThresh (Threshold: LONGINT);
```

Threshold specifies the minimum amount of contiguous memory you want your application to have after an editing operation (such as pasting) is performed *without* Undo/Redo services. The condition is checked just before a change is made to an active editing field.

If there is not enough memory to set up the Undo/Redo services and the largest piece of continuous memory is smaller than this specified value after the edit is performed (such as a paste or typing), the user is warned with a message stating "WARNING… Not enough memory for this operation." A "Cancel" button lets the user cancel the editing operation without making any changes.

This threshold is set to a reasonable default value when Tools Plus is initialized, so your application benefits from low-memory protection without you having to explicitly do anything. You can change the message displayed by Tools Plus as described in the Multiple Languages chapter.

SetTELowMemThresh

Specify minimum free memory below which a "low memory" warning is displayed while typing.

C
```
pascal void SetTELowMemThresh (long Threshold);
```

Pascal
```
procedure SetTELowMemThresh (Threshold: LONGINT);
```

Threshold specifies the minimum amount of contiguous memory you want your application to have after the user types in a field. The condition is checked as the user types in an active field.

If the largest piece of continuous memory is smaller than this specified value after the user types a character, the user is warned with a message stating "WARNING… Low memory!" An "OK" button lets the user continue. This message is displayed every 90 seconds as long as the user continues to type while memory is low.

This threshold is set to a reasonable default value when Tools Plus is initialized, so your application benefits from low-memory protection without you having to explicitly do anything. You can change the message displayed by Tools Plus as described in the Multiple Languages chapter.

GetTEHandle

Get a handle to a field's TextEdit record.

C
```
pascal TEHandle GetTEHandle (short Field);
```

Pascal
```
function GetTEHandle (Field: INTEGER): TEHandle;
```

This routine returns a standard TEHandle to a field that was created by a Tools Plus routine. You should never need to use this routine. It is provided for advanced programmers who may have specialized needs. Always use Tools Plus routines to create and manipulate fields.

Field specifies the field number (from 1 to 32767) in the current window whose handle is being retrieved. If the current window doesn't belong to your application, or if no windows are open, or if *Field* specifies a field that does not exist, GetTEHandle returns nil.

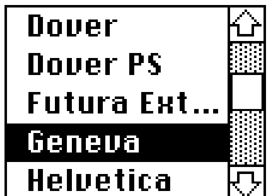
To conserve memory, Tools Plus does not always allocate a TextEdit record for each field. This is detailed at the beginning of this chapter and in the section describing the NewField routine. If a field is "buffered" with its own TextEdit record, you can get a handle to the TextEdit record at any time. If the field is not buffered, meaning it shares a single TextEdit record with other fields on the same window, then you can only obtain a handle to the TextEdit record when the field is active. This is true even if the field is deselected when its window is inactive.

☞ **Warning**: If you need to lock the handle or change its attributes, do so temporarily then restore the original settings before using any Tools Plus routines. If you alter this handle or any data that is made accessible by this handle, you do so at your own risk.

---

10 List Boxes

List boxes are a mechanism that lets the user make a selection from multiple choices. Where this interface differs from radio buttons, is that the user can optionally make multiple selections from the list, and the available choices are dynamic. Your application specifies the list box's dimensions as the visible area in which the items appear. A 1-pixel border is drawn just *outside* these co-ordinates and a scroll bar is integrated to the right of the list box, thereby consuming an additional 16 pixels. Once a list box is created, your application can define each line of text within the list. Lines can be added, changed and deleted as required. Tools Plus also supports the use of custom LDEFs.



Various selection rules can be put into place to control how the user can select lines. The simplest rule allows only one line to be selected at a time. More complex methods allow multiple lines to be selected with various limitations imposed. When a line is selected, it is highlighted, such as "Geneva" in the example to the left. When the user clicks or double-clicks a line, Tools Plus reports this to your application with a doListBox event (your application may choose to ignore double-clicks). Although your application can create blank lines in a list box, they can't be selected by the user.

List boxes are created on the current window by the NewListBox routine. Each list box is referenced by a unique list box number that can be from 1 to 511. This number is specified when the list box is created, and refers to the specific list box until that list box is deleted. Note that the list box number is related to its associated window. This means that two different windows can each have a list box numbered "1" without interfering with each other. Whenever the user clicks on a line within the list box, Tools Plus reports this to your application.

After NewListBox creates an empty list box, repeated calls to SetListBoxText will append lines to the list box, or replace an existing line's text with new text. The GetListBoxText routine is used to obtain any line's text. Individual lines are referenced by a relative line number, where the top line is line 1, the second from the top is line 2, and so on. All lines are referenced by the relative line number, even when lines are inserted or deleted.

The InsertListBoxLine routine inserts a new line between two existing lines. The DeleteListBoxLine routine deletes an existing line. ResNamesToListBox inserts resource names (such as fonts or sounds), sorted alphabetically, at a specified line. In all cases, if any other lines were selected before calling the routine, they retain their selection status.

SetListBoxLine is used to highlight a line, and is often used to set a default line the first time a list box is displayed. The first selected line is automatically scrolled into view. GetListBoxLine is a complementary routine that tells you if a specific line is selected or not. An additional routine, GetListBoxLines (ending with an "s") is used to determine the next selected line. This is useful when the selection of multiple lines is allowed because your application does not have to query each line individually.

Lines in a list box can be arranged in alphabetical order by adding or inserting lines in the correct place. The SearchListBox routine tells you where to insert a new text line to make the list alphabetic.

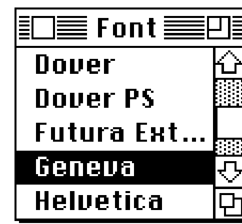
One additional routine is used to make your list boxes look professional: DrawListBox. Because list boxes are "live," the use of SetListBoxText, SetListBoxLine, InsertListBoxLine, and DeleteListBoxLine has an immediate and visible affect on your list. This can become quite unsightly when adding one line at a time to a list box of any significant length. DrawListBox turns the drawing process off prior to your application's maintenance of lines. When the task is completed, DrawListBox turns the drawing process on and instantly displays all the visible lines.

An entire list box can be deleted by using DeleteListBox. ClearListBox deletes all the lines in a list box.

When a window becomes inactive, any selected lines become deselected and the scroll bar is disabled. When the window is activated again, the selection(s) reappear and the scroll bar is enabled.

When a list box is no longer required, it is deleted by the DeleteListBox routine, which releases the memory used by the list box. This is done automatically if a window is closed. A list box can also be hidden or displayed with the ListBoxDisplay routine. List boxes can be moved to a new location with MoveListBox and have their width and/or height changed with SizeListBox. MoveSizeListBox combines both tasks by letting you specify new co-ordinates for the list box.

Auto-Positioning Options



Tools Plus can automatically position a list box if you use one or more of the following constants in place of actual co-ordinates: listLeftEdge, listTopEdge, listRightEdge and listBottomEdge. A special situation exists when you create a list box whose bottom co-ordinate is equal to the window's bottom local co-ordinate (or listBottomEdge), and the list's right co-ordinate is equal to the window's right local co-ordinate less 15 pixels (or listRightEdge): the list attaches itself to the window's bottom right corner and resizes automatically when the window's size changes. Tools Plus also shortens the scroll bar appropriately if the window has a grow box so that the list fits perfectly around the window's grow box.

Fonts

All list boxes default to using the Chicago 12pt font. When a list box is created, it can optionally adopt and remember the window's current font, size and style settings (as set by the TextFont, TextSize, and TextFace routines) by including the listUseWFont option in the spec parameter. The window's settings can then be changed without affecting the list box. Unlike regular list boxes, Tools Plus list boxes can each have a different font. You can use the GetListBoxFontSettings and SetListBoxFontSettings routines to get and set the list box's font, size and style settings.

Colors

By default, a list box is displayed using black text on a white background. You can change this by adding the listColorList constant to the list's type when it is created. When doing so, the list box stores the window's foreground and background colors and displays its text using these colors. The GetListBoxColors and SetListBoxColors routines can be used to set and retrieve a lists box's text and background colors. When you implement a list box as a control (an option under Tools Plus), the list box ignores the window's colors. Note that some controls ignore color settings, particularly those in the Appearance Manager. If you create a list box using the Appearance Manager's List Box control, it ignores color settings.

Appearance Manager Controls

The Appearance Manager, first introduced in mid 1997 with Mac OS 8, gives your application a number of 3D controls including a list box control with an integrated 3D scroll bar. All the new Appearance Manager controls are implemented as CDEFs, but unlike third party CDEF resources that must be installed in your application when it is built, the Appearance Manager's list box control is available to your application without having to install it. It is available from the system, just like regular system controls, if the Macintosh running your application has an Appearance Manager.

If you want to use the Appearance Manager's list box control, you need to make your application "Appearance Manager aware." 680x0 applications are automatically Appearance Manager aware. To make your PowerPC application Appearance Manager aware, see the Designing Your Application chapter of this manual for details in the "Using the Appearance Manager" section. Your application must also include an 'ldes' resource to let the Appearance Manager do its work. In Tools Plus, a single 'ldes' resource is shared by all list box controls. The settings in this resource are ignored because Tools Plus populates it with the correct values just before the Appearance Manager reads the resource. To include an 'ldes' resource in your application, just copy the 'ldes' resource that is supplied in the "Optional Resources" folder into the resource file you are using for your project.

See the chapters on Buttons, Scroll Bars, Editing Fields and Pop-Up Menus in this user manual for additional Appearance Manager controls.

Note: For complete information on Appearance Manager concepts, the Appearance Manager's features, and how to best use the Appearance Manager's new controls, please read the documentation pertaining to the Appearance Manager. It is available from Apple or in the latest issue of Inside Macintosh. This manual does not duplicate that material.

Note: Remember to include an ‘ldes’ resource in your project!

List Box (CDEF 22)

List Box control

Tools Plus supports the Appearance Manager’s list box control in a way that, from a programmer’s perspective, it is indistinguishable from a regular list box. This lets you use a consistent set of Tools Plus routines and programming principles to take advantage of the List Box control if it is available, or the regular List Manager list box. Tools Plus automatically implements a regular List Manager list box if the Appearance Manager is not available. Remember to include an ‘ldes’ resource in your project.

```
CONST
   kControlListBoxProc          = 352;  {List Box ProcIDs                    }
```

Creating a List Box Using a ‘CNTL’ Resource

Tools Plus offers considerable versatility in the way it supports the creation of list boxes from ‘CNTL’ resources. These features are most often used when opening a dialog (‘DLOG’ resource) that contains list boxes. In all cases, the ‘CNTL’ resource specifies a CDEF ID of 22 which produces a procID of 352 plus any variants. When you open a dialog, ‘CNTL’ resources that reference CDEF ID 22 (the list box control) create a Tools Plus list box. The translation from a ‘CNTL’ resource to a Tools Plus list box takes place as follows:

- Tools Plus starts by assuming that you want to use the Appearance Manager’s list box control (CDEF 22) and it attempts to create the control.
- If the Appearance Manager is not available, a regular List Manager list box is created. You can use the same Tools Plus routines to access the List Manager’s list box as you would a list box control.
- The list box is created using the listOnlyOne option, thereby allowing only one item to be selected by the user in the list.
- To set the appearance and behavior specifications for a list box, place the specification’s value in the ‘CNTL’ resource’s contrlRfCon field, the reference constant. A list of possible values can be found in the NewListBox description.

Note: Remember to include an ‘ldes’ resource in your application. See the “Appearance Manager Controls” section earlier in this chapter for details. Flag your ‘CNTL’ and ‘ldes’ resources as purgeable to save memory. Tools Plus makes a copy of their data.

Appearance Manager and Keyboard Focus

Before the Appearance Manager’s arrival, the only user interface element that could process keystrokes was an editing field. With the Appearance Manager present, a variety of user interface elements can process keystrokes, such as editing fields, list boxes and the clock control. Keystrokes are directed at only one user interface element at a time (and possibly at no element at all). When a user interface element processes keystrokes in such a way, it is said to have the “keyboard focus.” Only one user interface element can have the keyboard focus at a time, and it is visually indicated with a highlighted “band” around the object. Tools Plus takes care of the focus highlight, and of applying keystrokes to the element that has the keyboard focus.

The process of moving the keyboard focus between objects, either by tabbing or clicking, is identical to that of navigating between editing fields. For details, see the “Clicking and Tabbing” section in the Editing Fields chapter.

Special Considerations

Starting with Mac OS 8.5, the Appearance Manager’s List Box control (i.e., a list box created with the listSystemBody option) always draws a 3 pixel thick band around the outer edge of the box using the window’s background theme. If you want the list box to be drawn perfectly when running on Mac OS 8.5 or later, you must make sure you do one of the following:

- Leave the window's backdrop color as white, and do not apply a background theme or backdrop color to the window.
- Apply a background theme to the window
- Create a standard list box without using the listSystemBody. This list box will not get the keyboard focus.

Handling List Boxes

Once a list box is created, Tools Plus performs all the processing required within the box and its scroll bar. When a window in inactive, Tools Plus deselects items in all list boxes on that window. When the window is activated again, all list boxes regain their original state as specified by your application. Tools Plus constantly inquires about any events that have occurred, inlcuding events in a list box.

Several types of events *may* indicate that your application has to perform some action. For example, you may want to enable or disable buttons based on whether a selection has been made in the list box. Or you may ignore all list box action and use GetListBoxLine to determine the selection only after an OK button is clicked. Though various interpretations can be implemented, please adhere to the Macintosh User Interface Guidelines as outlined in Inside Macintosh. In any case, Tools Plus tells your application if any activity has occurred in a list box, or if a selection has been double clicked.

See the Event Management chapter for details pertaining to list box events.

NewListBox

Create a new list box.

C
```
pascal void NewListBox (short ListBox, short left, short top, short right,
            short bottom, long Spec);
```

Pascal
```
procedure NewListBox (ListBox, left, top, right, bottom: INTEGER;
            Spec: LONGINT);
```

ListBox specifies the list box number (from 1 to 511) that is created in the current window. Once a list box is created, it is referenced by this list box number. If a list box has been previously created in the current window using the same number, it is replaced with a new (empty) list box as specified by the parameters in the NewListBox routine (which is a good way to clear all of an existing list box's lines). If the current window doesn't belong to your application, or if no windows are open, NewListBox does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in the current window's local co-ordinates that determine the list box's size and location in the window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). A 1-pixel wide outline is drawn as a frame for the list box just outside these co-ordinates. Also, a scroll bar is created along the entire length of the list box's right side. The scroll bar is 16 pixels wide and is drawn outside the specified co-ordinates. To make a list box operate at its best, the list box's height (difference between top and bottom) should be a multiple of its font's height (font height can be determined by calling the GetFontInfo routine and adding *Ascent* + *Descent* + *Leading*). If bottom is less than top, NewListBox takes the absolute value of bottom and creates a list box that is that many lines high (i.e., if top is 30 and bottom is -8, an eight line list box is created starting downward at the top co-ordinate). You can align the list box's edge to a window's edge by using the listLeftEdge, listTopEdge, listRightEdge or listBottomEdge constants in place of the list box's left, top, right or bottom co-ordinates.

Spec specifies a list box's appearance and behavior. It is a combination of various Tools Plus options detailed below.

Appearance and Behavior Specification

Spec specifies a list box's appearance and behavior. To maintain backwards compatibility with previous versions of Tools Plus, the value for this 4-byte long integer can be calculated in either of the following ways:

(a) By adding only standard Apple constants. Apple's standard LDEF is used.

(b) By adding Tools Plus constants plus an optional custom LDEF procID. If no procID is specified, Apple's standard LDEF is used. Using this method, spec is a combination of an LDEF procID (low 16 bits) plus various Tools Plus options (high 16 bits).

Do not mix standard Apple constants with Tools Plus constants when specifying the spec. Doing so will produce unpredictable results.

Choose any of the following selection methods (when using Apple's standard LDEF)…

`0 (zero)` — This is the default selection method. If additional options are specified they override the default behavior. Line selection is affected by modifier keys as follows:

Option	The Option key is always ignored even when used in combination with other keys.
no Shift or ⌘	Any click in the list box deselects previous selections and selects the line clicked by the user. If the mouse is moved while the mouse button is held down, only the line beneath the cursor is selected.
Shift	If the Shift key is down before clicking the mouse, the selection is extended or shortened as if it were an expandable rectangle. When the mouse is first clicked, the selection is changed to include the line that was just clicked. If the mouse is dragged, the selection either extends or shortens to follow the mouse's pointer.
⌘	If the ⌘ key is down before clicking the mouse, lines are either selected or unselected, depending on the first clicked line. If the initial line was selected, it is deselected along with any other lines the mouse's pointer passes over. If the initial line was not selected, it is selected along with any other lines the mouse's pointer passes over. This is called "sense of first line."

`lOnlyOne` — Only one line can be selected at a time. Any previous selection is deselected when a new line is clicked.

`lExtendDrag` — Selections are extended without using the Shift key. All lines dragged over by the mouse are selected. It works best when used in conjunction with lNoDisjoint, lNoExtend, and lUseSense.

`lNoDisjoint` — Multiple lines can be selected, but all lines are deselected when the mouse is clicked. This occurs even if the Shift or ⌘ keys are held down, and prevents "disjointed" selections.

`lNoExtend` — The current selection is ignored. The mouse's click defines an anchor point for the new Shift selection.

`lUseSense` — If the Shift key is pressed, "sense of first line" is in effect. This means if the initial line was selected, it is deselected along with any other line the mouse passes over. If the initial line was not selected, it is selected along with any other line the mouse passes over.

Common or interesting selection methods (combinations of the above)…

`lOnlyOne` — Only one selection can be made at a time. Any previous selection is deselected. This is a Macintosh standard.

`lNoExtend + lUseSense` — A click deselects previous selections. A Shift-Click selects a deselected line, or deselects a selected line. A Shift-Click can also be dragged to perform the same action across other lines. The "Font/DA Mover" uses this method.

`lExtendDrag +`
`lNoDisjoint +`
`lNoExtend +`
`lUseSense`
	A click deselects previous selections. Dragging the click selects lines along the drag. This is a good way to let the user select multiple lines that *must* be grouped together.

`lExtendDrag +`
`lNoExtend +`
`lUseSense`
	A click selects a deselected line, or deselects a selected line. A click can be dragged to perform the same action across other lines. Shift is ignored. This selection method makes a list box behave as though it were a list of check boxes. Although this is a neat idea, it does not follow the Macintosh User Interface Guidelines.

If you need to create a list box that goes beyond Apple's standard list and/or uses a custom LDEF, use the following Tools Plus constants in place of Apple's constants.

Choose any of the following selection methods…

`listDefault`	Same as 0 (zero) when using Apple's constants.
`listOnlyOne`	Same as lOnlyOne when using Apple's constants.
`listExtendDrag`	Same as lExtendDrag when using Apple's constants.
`listNoDisjoint`	Same as lNoDisjoint when using Apple's constants.
`listNoExtend`	Same as lNoExtend when using Apple's constants.
`listUseSense`	Same as lUseSense when using Apple's constants.

Optionally choose any of the following options…

`listSystemBody`	Create the list box using the Appearance Manager's list box control. If the Appearance Manager is not available, a regular List Manager list box is created. When you create a list box with this option, the list dims when it is on an inactive window. **Note**: Only list boxes that are implemented as a control can be embedded into other controls.
`listUseWFont`	Display the list box using the window's current font, size and style settings (as set by the TextFont, TextSize, and TextFace routines). The list box stores this information for future reference. By default, list boxes are drawn using the system font (Chicago, 12 pt).
`listDimWhenInactive`	Using this option causes the list's text and frame to be dimmed when the list is inactive, such as on an inactive window. This option is automatically included when you use the listSystemBody option.
`listColorList`	Use the window's foreground color for the list's text, and the window's background color for the list's background. The list stores this information for future reference. By default, the list box is drawn using black text on a white background. This option is ignored if you use the listSystemBody option.
`listNoFrame`	Do not draw a box around the list. By default, list boxes have a 1-pixel wide frame around them. This option is useful if you are integrating the list box into a graphic element and you do not want a line between the two elements. This option is ignored if you use the listSystemBody option.
`listAutoMoveSize`	Automatically move and/or resize the list box when the window's size changes. The AutoMoveSize routine lets you specify which sides are altered. You can use the AutoMoveSizeListBox routine as an alternative to setting this option.
`listHidden`	Create a hidden list box. This kind of list box is accessible to your application but not to the user.
(LDEF procID)	If you want to use an LDEF other than the default Apple LDEF, add its procID to the spec's value. Do not specify kControlListBoxProc, because the is a CDEF (control) procID, and not one for an LDEF. If you use the listSystemBody option, it automatically calls the Appearance Manager's list box LDEF.

Also see: NewListBoxRect and NewDialogListBox.

Note: Tools Plus makes no attempt to control the placement of list boxes or to protect them once they have been created. It is your responsibility to ensure that list boxes are of sufficient size to contain their lines, and that their placement within the window is reasonable and does not conflict with other objects. Furthermore, you should not allow your application's text and drawing processes to interfere with list boxes. Windows with a "size box" should not allow list boxes to be obscured or hidden by making the window too small.

```
CONST                                        {LIST BOXES:                                 }
                                             {Standard Apple constants for backwards      }
                                             {compatibility.  Don't mix with T+ constants: }
               lUseSense   = 4;              {   Shift senses state of initial line       }
               lNoExtend   = 16;             {   Shift won't extend selection             }
               lNoDisjoint = 32;             {   Click deselects previous selections      }
               lExtendDrag = 64;             {   Drag extends without shift key           }
               lOnlyOne    = -128;           {   Prevent multiple selections              }

                                             {List box co-ordinates:                      }
               listLeftEdge     =-32768;     {  Window's left edge                        }
               listTopEdge      =-32768;     {  Window's top edge                         }
               listRightEdge    = 32767;     {  Window's right edge                       }
               listBottomEdge   = 32767;     {  Window's bottom edge                      }

                                             {Tools Plus List Box constants:              }
               listDefault      = $00000000; {   Default list box                         }
               listSystemBody   = $80000000; {   Use Appearance Manager List Box          }
               listUseWFont     = $40000000; {   Use window's font                        }
               listNoFrame      = $00040000; {   Don't draw frame around the list box     }
               listColorList    = $00080000; {   Use color settings for this list box     }
               listHidden       = $00100000; {   Create a hidden list box                 }
               listAutoMoveSize = $00200000; {   Auto-move/size as window's size chg      }
               listUseSense     = $00400000; {   Shift senses state of initial line       }
               listNoExtend     = $00800000; {   Shift won't extend selection             }
               listNoDisjoint   = $01000000; {   Click deselects previous selections      }
               listExtendDrag   = $02000000; {   Drag extends without shift key           }
               listOnlyOne      = $04000000; {   Prevent multiple selections              }
```

NewListBoxRect

Create a new list box.

C
```
pascal void NewListBoxRect (short ListBox, const Rect *Bounds,
              long Spec);
```

Pascal
```
procedure NewListBoxRect (ListBox: INTEGER; Bounds: RECT;
              Spec: LONGINT);
```

NewListBoxRect is identical to the NewListBox routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

NewDialogListBox

Create a new list box in a dialog using a dialog item's co-ordinates.

C
```
pascal void NewDialogListBox (short ListBox, long Spec);
```

Pascal
```
procedure NewDialogListBox (ListBox: INTEGER; Spec: LONGINT);
```

NewDialogListBox is identical to the NewListBox routine, except that the list box is created in a dialog (a window opened with the LoadDialog routine, or one that had a dialog list attached with the LoadDialogList routine). The list box's co-ordinates are obtained from the dialog item whose number matches the list box number.

EmbedListBoxInButton

Embed a list box into a button or into the window's root control (Appearance Manager only).

C
```
pascal void EmbedListBoxInButton (short ListBox, short ContainerButton);
```

Pascal
```
procedure EmbedListBoxInButton (ListBox, ContainerButton: INTEGER);
```

The Appearance Manager lets you embed a control into a parent control such that when the parent is hidden or disabled, all embedded controls are similarly affected. All Tools Plus routines that load a dialog item list (LoadDialog, LoadSpecDialog, LoadDialogList, etc.) automatically embed controls at all times. EmbedListBoxInButton lets you manually embed a list box into a button, or into the window's root control. Note that the term "button" does not literally mean a button control. It means any control that is implemented as a button in Tools Plus. The most likely candidate is a Group Box control. If the Appearance Manager is not available, EmbedListBoxInButton does nothing.

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, EmbedListBoxInButton does nothing. Note that the only list boxes that can be embedded are those that are drawn using a CDEF (use the listSystemBody option when creating the list box).

ContainerButton specifies the button number (from 1 to 511) into which *ListBox* is embedded. This control must exist in the current window, and it must be a "container" type control such as the Appearance Manager's Group Box. The list box must fit entirely within the container control or EmbedListBoxInButton does nothing. If a value of 0 is provided for a container button, *ListBox* is embedded into the window's root control.

Also see: EmbedListBoxInScrollBar and SetAutoEmbed.

EmbedListBoxInScrollBar

Embed a list box into a scroll bar or into the window's root control (Appearance Manager only).

C
```
pascal void EmbedListBoxInScrollBar (short ListBox,
            short ContainerScrollBar);
```

Pascal
```
procedure EmbedListBoxInScrollBar (ListBox, ContainerScrollBar: INTEGER);
```

The Appearance Manager lets you embed a control into a parent control such that when the parent is hidden or disabled, all embedded controls are similarly affected. All Tools Plus routines that load a dialog item list (LoadDialog, LoadSpecDialog, LoadDialogList, etc.) automatically embed controls at all times. EmbedListBoxInScrollBar lets you manually embed a list box into a scroll bar, or into the window's root control. Note that the term "scroll bar" does not literally mean a scroll bar control. It means any control that is implemented as a scroll bar in Tools Plus. If the Appearance Manager is not available, EmbedListBoxInScrollBar does nothing.

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, EmbedListBoxInScrollBar does nothing. Note that the only list boxes that can be embedded are those that are drawn using a CDEF (use the listSystemBody option when creating the list box).

ContainerScrollBar specifies the scroll bar number (from 1 to 511) into which *ListBox* is embedded. This control must exist in the current window, and it must be a "container" type control. The list box must fit entirely within the container control or EmbedListBoxInScrollBar does nothing. If a value of 0 is provided for a container scroll bar, *ListBox* is embedded into the window's root control.

Also see: EmbedListBoxInButton and SetAutoEmbed.

GetFreeListBoxNum

Get the first unused list box number.

C
```
pascal short GetFreeListBoxNum (void);
```

Pascal
```
function GetFreeListBoxNum: INTEGER;
```

Some developers may prefer to write code that more closely resembles a traditional Macintosh application, in that creating an object returns a reference to it such as a handle or pointer. Instead of having to assign your own list box number, GetFreeListBoxNum returns the first unused (free) list box number. Using this routine, you can assign an unused list box number to a variable, then use that variable throughout your application without concern for the true list box number.

GetFreeListBoxNum returns the first free list box number on the current window. If the current window doesn't belong to your application, if no windows are open, or if the maximum number of list boxes has already been created on the current window (no new ones can be created), GetFreeListBoxNum returns a value of zero (0).

DeleteListBox

Delete a list box.

C
```
pascal void DeleteListBox (short ListBox);
```

Pascal
```
procedure DeleteListBox (ListBox: INTEGER);
```

ListBox specifies the list box number (from 1 to 511) that is deleted from the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, DeleteListBox does nothing. Use KillListBox if you want to delete the list box without removing its image from the window.

KillListBox

Delete a list box without affecting its image on the window.

C
```
pascal void KillListBox (short ListBox);
```

Pascal
```
procedure KillListBox (ListBox: INTEGER);
```

KillListBox is identical to DeleteListBox except that it does not remove the list box's image from the window. This routine is useful for scrolling list boxes in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetListBox repositions the list box's co-ordinates without affecting its image (since ScrollRect has already moved it). KillListBox then deletes the list boxes that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

ListBoxDisplay

Hide or show a list box.

C
```
pascal void ListBoxDisplay (short ListBox, Boolean Show);
```

Pascal
```
procedure ListBoxDisplay (ListBox: INTEGER; Show: BOOLEAN);
```

ListBoxDisplay hides or shows a list box on the current window. The result is seen immediately.

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, ListBoxDisplay does nothing.

Show indicates if the list box is being hidden or displayed. The two constants that can be used for this flag are *on* and *off*.

ListBoxIsVisible

Determine if a list box is visible.

C
```
pascal Boolean ListBoxIsVisible (short ListBox);
```

Pascal
```
function ListBoxIsVisible (ListBox: INTEGER): BOOLEAN;
```

ListBoxIsVisible reports if a list box is visible on the current window, or if it is hidden.

ListBox specifies the list box number (from 1 to 511) that is queried in the current window.

This routine's value returns *true* if the list box is visible, and *false* if the list box is hidden. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, ListBoxIsVisible returns *false*. This routine takes control embedding into account, so it will return *false* if the target list box is embedded and its container control is hidden.

ObscureListBox

Hide a list box without removing its image from the window.

C
```
pascal void ObscureListBox (short ListBox);
```

Pascal
```
procedure ObscureListBox (ListBox: INTEGER);
```

ObscureListBox hides a list box on the current window without removing its image from the window. This routine is useful for scrolling list boxes in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetListBox repositions the list box's co-ordinates without affecting its image (since ScrollRect has already moved it). ObscureListBox then hides the list boxes that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

ListBox specifies the list box number (from 1 to 511) that is hidden in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, ObscureListBox does nothing.

ActivateListBox

Activate a list box to give it the keyboard focus.

C
```
pascal void ActivateListBox (short ListBox);
```

Pascal
```
procedure ActivateListBox (ListBox: INTEGER);
```

ListBox specifies the list box number (from 1 to 511) that acquires the keyboard focus in the current window. ActivateListBox does nothing under any of these conditions: the current window doesn't belong to your application, no windows are open, the list box does not exist in the current window, the list box is disabled or hidden, the list box was not implemented using the listSystemBody option, or the Appearance Manager is not available to your application.

Activating a list box allows the user to interact with the list box by typing on the keyboard. On an active window, the list box acquires the keyboard focus making it the item that automatically processes keystrokes. Visually, this is indicated by having highlighted lines. Additionally, the list box is encompassed with a highlighting keyboard focus band to indicate that it has the focus. Using ActivateListBox in an active window removes the keyboard focus from any other object that may have the focus within the same window or any other active window such as a tool bar or floating palette. This action may deactivate an active editing field.

If the list box being activated is in an active window that allows access to pull-down menus, the Edit menu's "Undo" item is changed to "Can't Undo" and is disabled. The "Cut", "Copy", "Paste", "Clear" and "Select All" items are also disabled.

Your application can activate virtually any editing field or Appearance Manager control that accepts the keyboard focus. This flexibility can lead to a confusing user interface by allowing the keyboard focus to jump between active windows. A good rule to observe is to activate a single item only on a standard window (not a tool bar or a floating palette) when the window first opens. This sets up the default keyboard focus item for that window. At all other times, activate a list box only in response to a user's actions.

Also see: HaveTabInFocus, TabToFocus, the doClickToFocus event, and ClickToFocus for other activating services.

GetListBoxRect

Get a list box's co-ordinates.

C
```
pascal void GetListBoxRect (short ListBox, Rect *Bounds);
```

Pascal
```
procedure GetListBoxRect (ListBox: INTEGER; var Bounds: RECT);
```

ListBox specifies the list box number (from 1 to 511) that is queried in the current window.

Bounds returns the list box's bounding rectangle specified in the window's local co-ordinates. These co-ordinates match those used to create the list box. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, *Bounds* returns with all co-ordinates set to zero (0).

SetListBoxText

Create a new line in a list box, or replace an existing line's text.

C
```
pascal void SetListBoxText (short ListBox, short LineNum, const Str255 Text);
```

Pascal
```
procedure SetListBoxText (ListBox, LineNum: INTEGER; Text: STRING);
```

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, SetListBoxText does nothing.

LineNum specifies the line number (from 1 to 32767) that is affected in the specified list box. The line is created if it does not already exist. If necessary, blank lines are created between the last line in the list box and the new line being created. If the line already exists, the line's text is simply replaced.

The *Text* parameter is the text that appears in the list box's line. When running on System 6 or older, text that is too long to be displayed in its entirety is truncated and appended with an ellipsis ("…"). System 7 (or later) does not do this. Instead, it first condenses character spacing, then, as a last resort, gives the appearance of truncation without altering the text.

Repeated calls of SetListBoxText should be bracketed by calls to DrawListBox. See the DrawListBox routine for details.

Programming Tips:

1. If you are going to create several list boxes, create all the empty list boxes first using NewListBox. This makes all the list boxes appear in quick succession. Next, turn list box drawing off by using DrawListBox, and fill all the list boxes as required. Lastly, turn list box drawing back on. Your list boxes will be filled in quick succession. No time is saved, but the display looks more professional.
2. If you know in advance how many lines are going to be added to a list box, add the last line *first*, even if it is blank. By doing this, all the blank lines you need will be created at the same time. Since it takes less time to replace text in an existing line than it does to append a new line, your list box will be filled much quicker.
3. If you are creating a list box of font names, be aware that some Macintoshes have some fonts in ROM. That means that calling CountResources('FOND') will include not only the number of fonts in your system, but in ROM too. Before you add a font name to your list box, check to see if it already exists in the list to avoid duplicates. You can use the ResNamesToListBox routine to find, sort and insert the resource names for you.
4. If your application is running on a system file prior to System 7, long lines of text in the list *are* actually truncated as you see them on the screen (i.e., "Long word…"). In System 7 or higher, long lines only *appear* to be truncated. When you retrieve a line by using the GetListBoxText routine under System 7 or higher, you will obtain the full string that was placed in that line, even if it appears truncated on the screen to fit in the list. The same call, when using a system file prior to System 7, retrieves the truncated text as it appears in the list.

☞ **Warning**: Apple's List Manager (and the LDEF written by Apple) is limited to 32K of data. This means that your list can't contain more than thirty-two thousand characters. Tools Plus will break this limit in a future release by writing our own list manager. Tools Plus routines will continue to work as they do now, but will have additional functionality available to them.

ResNamesToListBox

Insert resource names into a list box.

C
```
pascal void ResNamesToListBox (short ListBox, short LineNum, ResType rType);
```

Pascal
```
procedure ResNamesToListBox (ListBox, LineNum: INTEGER; rType: RESTYPE);
```

This routine finds all named resources of the specified type and inserts those names (sorted alphabetically) into a list box. Duplicated names are ignored as are ones that start with "." (period) or "%".

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, ResNamesToListBox does nothing.

LineNum specifies the line number (from 1 to 32767) where the resource names are inserted. The line is created if it does not already exist. If necessary, blank lines are created between the last line in the list box and the new line being created.

rType is the four character resource type whose names are being inserted into the list box. When running on System 6 or older, names that are too long to be displayed in their entirety are truncated and appended with an ellipsis ("…"). System 7 (or later) does not do this. Instead, it first condenses characters spacing then as a last resort, gives the appearance of truncation without altering the name. If you specify 'FOND' or 'FONT' resources, both are obtained since they are just different types of fonts.

StrToListBox

Copy a set of strings to a list box.

C
```
pascal void StrToListBox (short ListBox, Handle hRec);
```

Pascal
```
procedure StrToListBox (ListBox: INTEGER; hRec: HANDLE);
```

StrToListBox is a highly optimized routine used to copy a set of strings into a list box. It is about 30 times faster than using toolbox routines to populate the list. You can think of this routine as a "batch loader" for a list box.

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, StrToListBox does nothing. This list is cleared of entries before the new ones are added.

HRec is a handle to an indexed string structure, or commonly known as an 'STR#' record or resource. Your application can create and populate the record using Tools Plus's routines, or it can read an 'STR#' resource from a resource file. If this handle points to a resource, it is best if you flag the resource as purgeable since you won't need to keep it in memory after the strings have been loaded into the list. The following code shows you how to copy an 'STR#' resource to a list box:

```
hRec := GetResource('STR#', 128);  {Load the resource into memory                        }
HNoPurge(hRec);                    {Prevent purging while we copy strings to the list box }
StrToListBox(1, hRec);             {Copy 'STR#' resource to the list box                  }
HPurge(hRec);                      {Allow the resource to be purged                       }
ReleaseResource(hRec);             {Release the resource to save memory                   }
```

The hRec parameter can also be used to specify an 'STR#' resource ID instead of a handle to an 'STR#' structure. To do so, typecast the resource ID as a handle before passing it to the routine.

GetListBoxText

Get the text from a specific line in a list box.

C
```
pascal void GetListBoxText (short ListBox, short LineNum, Str255 Text);
```

Pascal
```
procedure GetListBoxText (ListBox, LineNum: INTEGER; var Text: Str255);
```

ListBox specifies the list box number (from 1 to 511) that is queried in the current window.

LineNum specifies the line number (from 1 to 32767) that is queried in the specified list box.

The *Text* variable is the text that appears in the list box's specified line. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, or if LineNum does not exist in the specified list box, *Text* will return as a null string.

Programming Tips:

1 If your application is running on a system file prior to System 7, long lines of text in the list *are* actually truncated as you see them on the screen (i.e., "Long word…"). In System 7 or higher, long lines only *appear* to be truncated. When you retrieve a line by using the GetListBoxText routine under System 7 or higher, you will obtain the full string that was placed in that line, even if it appears truncated on the screen to fit in the list. The same call, when using a system file prior to System 7, retrieves the truncated text as it appears in the list.

SearchListBox

Search a list box for a line that is greater than or equal to the specified text.

C
```
pascal short SearchListBox (short ListBox, const Str255 Text);
```

Pascal
```
function SearchListBox (ListBox: INTEGER; Text: STRING): INTEGER;
```

ListBox specifies the list box number (from 1 to 511) that is queried in the current window.

Text is a string that is used to find a matching line.

The routine searches all the lines in the specified list box for the string indicated by *Text*. If an exact match is found, the routine's value returns the line number containing the matching string. If a match is not found, the routine returns the first line number that contains text that is *greater* than the specified *Text*. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, a value of zero (0) is returned.

This routine is most useful if you want to create a list box with the lines in alphabetical order (such as fonts). Before your application adds a line, call SearchListBox to determine where the line should be inserted. Using the returned value, call InsertListBoxLine to insert a blank line in the list box, then use SetListBoxText to set the blank line's text. Keep in mind that the alphabetic comparison that is made between the list box's lines and the string specified by *Text* differentiates between upper and lower case letters. Also, under systems older than System 7, the list box's lines may have been truncated and suffixed by with an ellipsis ("…") if they were too wide to fit in the list box.

SetListBoxLine

Select or deselect a line in a list box.

C
```
pascal void SetListBoxLine (short ListBox, short LineNum, Boolean SetIt);
```

Pascal
```
procedure SetListBoxLine (ListBox, LineNum: INTEGER; SetIt: BOOLEAN);
```

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, SetListBoxLine does nothing.

LineNum specifies the line number (from 1 to 32767) that is affected in the specified list box. If the line number does not exist in the specified list box, SetListBoxLine does nothing. This line scrolls into view if it is being selected.

SetIt specifies if the line is to be selected or deselected. The constants *on* and *off* can be used for this purpose.

The SetListBoxLine routine should be used to set default lines within a list box immediately after it is created. Normally, you won't have to use this routine because the user's actions will select and deselect lines.

If you are going to select a number of lines using this routine, first use the DrawListBox routine to turn list drawing off, set your lines, then use DrawListBox to turn drawing on. The user sees all the lines selected at once instead of seeing each line selected individually.

```
CONST                         {list box's line state                      }
               on  = true;    {line is selected                           }
               off = false;   {line is deselected                         }
```

GetListBoxLine

Determine if a specified line in a list box is selected or deselected.

C
```
pascal Boolean GetListBoxLine (short ListBox, short LineNum);
```

Pascal
```
function GetListBoxLine (ListBox, LineNum: INTEGER): BOOLEAN;
```

ListBox specifies the list box number (from 1 to 511) that is queried in the current window.

LineNum specifies the line number (from 1 to 32767) that is queried in the specified list box.

The routine's value returns as *true* if the line is selected, or *false* if the line is not selected. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, or if LineNum does not exist in the specified list box, the routine's value returns a *false*.

See the GetListBoxLines (ending with an "s") routine.

GetListBoxLines

Find the first selected line in a list box, starting at a specified line number.

C
```
pascal short GetListBoxLines (short ListBox, short LineNum);
```

Pascal
```
function GetListBoxLines (ListBox, LineNum: INTEGER): INTEGER;
```

ListBox specifies the list box number (from 1 to 511) that is queried in the current window.

LineNum specifies the line number (from 1 to 32767) that is the first in a series to be queried in the specified list box.

The routine's value returns the line number of the first selected line starting at *LineNum*. If no selected lines were found, the routine returns a value of zero (0). If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, or if LineNum does not exist in the specified list box, the routine's value returns a zero (0).

When multiple lines can be selected in a list box, GetListBoxLines is a good way to determine which lines are currently selected. Instead of checking each line individually, use GetListBoxLines with a *LineNum* of 1 to determine the first selected line. Add 1 to the resultant value to resume the search starting at the next *LineNum*. When GetListBoxLines returns a value of zero, you know you have reached the end of the list.

A second use for this routine is to enable or disable a button depending on whether any items are selected in a list box. The "Open…" dialog box provides a good example. If a line (i.e., file name) is not selected in the list box, the Open button is disabled. As soon as a line is selected, the Open button is enabled. To do this, your application merely has to set up the correct default (or absence of one) in the list box, and correctly enable or disable a push button. When Tools Plus informs your application that some activity has taken place in the list box, call GetListBoxLines with a *LineNum* of 1 to determine if any lines were selected. If GetListBoxLines returns a non-zero value, you know a selection has been made, otherwise, no selections have been made. Based on this conclusion, you could enable or disable the push button accordingly.

InsertListBoxLine

Insert a blank line into a list box.

C
```
pascal void InsertListBoxLine (short ListBox, short LineNum);
```

Pascal
```
procedure InsertListBoxLine (ListBox, LineNum: INTEGER);
```

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, InsertListBoxLine does nothing.

LineNum specifies the line number (from 1 to 32767) that is the targeted destination for a blank line in the specified list box. The specified line number, and all the lines below it, are pushed down one line. If the line number does not exist in the specified list box, InsertListBoxLine does nothing.

Repeated calls to InsertListBoxLine should be bracketed by calls to DrawListBox. See the DrawListBox routine for details.

DeleteListBoxLine

Delete an existing line from a list box.

C
```
pascal void DeleteListBoxLine (short ListBox, short LineNum);
```

Pascal
```
procedure DeleteListBoxLine (ListBox, LineNum: INTEGER);
```

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, DeleteListBoxLine does nothing.

LineNum specifies the line number (from 1 to 32767) that is deleted. The specified line number is deleted, and all the lines below it are moved up one line. If the line number does not exist in the specified list box, DeleteListBoxLine does nothing.

If you want to clear a line (i.e., clear the existing text and leave a blank line), use SetListBoxText and specify a null string ("\p" in C, or ‘‘ in Pascal).

Repeated calls to DeleteListBoxLine should be bracketed by calls to DrawListBox. See the DrawListBox routine for details. You can delete all lines in a list box with ClearListBox.

ClearListBox

Delete all lines from a list box.

C
```
pascal void ClearListBox (short ListBox);
```

Pascal
```
procedure ClearListBox (ListBox: INTEGER);
```

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, ClearListBox does nothing.

ClearListBox deletes all lines in a list box. This routine is very quick regardless of the number of lines in the list box so you can leave list box drawing on when you use it.

ListBoxIsEnabled

Determine if a list box is enabled or disabled.

C
```
pascal Boolean ListBoxIsEnabled (short ListBox);
```

Pascal
```
function ListBoxIsEnabled (ListBox: INTEGER): BOOLEAN;
```

ListBox specifies the list box number (from 1 to 511) that is queried in the current window.

The routine's value returns *true* if the list box is enabled, and *false* if the list box is disabled. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, ListBoxIsEnabled returns *false*. ListBoxIsEnabled returns the list box's enabled state as it is currently displayed, so if the list box's window is inactive and has temporarily disabled the list box, ListBoxIsEnabled returns *false*.

SetListBoxFontSettings

Set a list box's font, size and style settings.

C
```
pascal void SetListBoxFontSettings (short ListBox,
            short theFont, short theSize, Style theStyle);
```

Pascal
```
procedure SetListBoxFontSettings (ListBox: INTEGER;
            theFont: INTEGER; theSize: INTEGER; theStyle: Style);
```

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if the list box does not exist, SetListBoxFontSettings does nothing. Otherwise, the change is seen immediately.

TheFont specifies the list box's new font. The default is Chicago, which is represented by the systemFont constant.

TheSize specifies the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle specifies the list box's new style. Special character constants defined by the Font Manager are bold, italic, underline and shadow. C programmers use the Font Manager's constants to specify a composite style, such as SetListBoxFontSettings(1, geneva, 9, bold + outline) for bold and outlined, or SetListBoxFontSettings(1, geneva, 9, 0) for plain text. Pascal programmers use the Font Manager's constants to specify a style set, such as SetListBoxFontSettings(1, geneva, 9, [bold, outline]) for bold and outlined, or SetListBoxFontSettings(1, geneva, 9, [ ]) for plain text.

A list box's font settings are set when a list box is created, so this routine is not normally used by many applications.

GetListBoxFontSettings

Get a list box's font, size and style settings.

C
```
pascal void GetListBoxFontSettings (short ListBox,
            short *theFont, short *theSize, Style *theStyle);
```

Pascal
```
procedure GetListBoxFontSettings (ListBox: INTEGER;
            var theFont: INTEGER; var theSize: INTEGER; var theStyle: Style);
```

ListBox specifies the list box number (from 1 to 511) in the current window whose font settings are being retrieved. If the current window doesn't belong to your application, if no windows are open, or if *ListBox* specifies a list box that does not exist, GetListBoxFontSettings returns default values.

TheFont is the list box's font number. The default is 0 which is represented by the systemFont constant.

TheSize is the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle is the list box's font style. The default is plain text, which is represented by 0 in C and [ ] in Pascal.

SetListBoxColors

Set a list box's colors.

C
```
pascal void SetListBoxColors (short ListBox,
            const RGBColor *TextColor, const RGBColor *BackColor);
```

Pascal
```
procedure SetListBoxColors (ListBox: INTEGER;
            TextColor: RGBColor; BackColor: RGBColor);
```

ListBox specifies the list box number (from 1 to 511) in the current window whose colors are being set. If the current window doesn't belong to your application, or if no windows are open, SetListBoxColors does nothing. Also, if *ListBox* specifies a list box that does not exist, or if Color QuickDraw is unavailable or not used, SetListBoxColors does nothing. The change is seen immediately, regardless if the list box was originally created with the listColorList option or not. If the list box is implemented using the listSystemBody option, it ignores color settings.

TextColor is the color of the list's text.

BackColor is the list box's background color upon which the text is drawn.

Normally, a list box's colors are set when this list box is created with NewListBox or NewListBoxRect, so this routine would not be used by many applications.

GetListBoxColors

Get a list box's colors.

C
```
pascal void GetListBoxColors (short ListBox,
            RGBColor *TextColor, RGBColor *BackColor);
```

Pascal
```
procedure GetListBoxColors (ListBox: INTEGER;
            var TextColor: RGBColor; var BackColor: RGBColor);
```

ListBox specifies the list box number (from 1 to 511) in the current window whose colors are being retrieved. If the current window doesn't belong to your application, or if no windows are open, or if *ListBox* specifies a list box that does not exist, GetListBoxColors returns default color values. If the list box is implemented as a control it ignores color settings.

TextColor is the color of the list's text. The default color is black.

BackColor is the list box's background color upon which the text is drawn. The default color is white.

ListBoxLineCount

Determine the number of lines in a list box.

C
```
pascal short ListBoxLineCount (short ListBox);
```

Pascal
```
function ListBoxLineCount (ListBox): INTEGER;
```

ListBox specifies the list box number (from 1 to 511) you wish to query in the current window.

The routine's value returns the number of lines in the specified list box. If the list box number does not exist, the routine returns zero.

DrawListBox

Turn list box drawing on or off (immediate update when lines are changed).

C
```
pascal void DrawListBox (short ListBox, Boolean DrawIt);
```

Pascal
```
procedure DrawListBox (ListBox: INTEGER; DrawIt: BOOLEAN);
```

When your application makes changes to a list box by adding, changing, deleting or inserting lines, the change is immediately visible. This can become quite unsightly when adding one line at a time to a list box of any significant length. Whenever changes are going to be made to more than a single line, turn drawing off *before* making any changes. After all the changes are completed, turn list drawing back on.

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the list box does not exist in the current window, DrawListBox does nothing.

DrawIt specifies if the drawing is to be turned on or off. The constants *on* and *off* may be used. When list drawing is turned off, any changes made to the lines are not displayed although changes are invisibly accumulated. When list drawing is turned back on, the list box is drawn instantly with all its lines and selections displayed.

DrawListBox has no effect on hidden list boxes since their drawing mode is always turned off while they are hidden.

```
CONST                           {List box's line drawing                  }
               on  = true;      {Text lines are drawn                     }
               off = false;     {Text lines are not drawn                 }
```

MoveListBox

Move a list box to a new location on the window.

C
```
pascal void MoveListBox (short ListBox, short toHoriz, short toVert);
```

Pascal
```
procedure MoveListBox (ListBox, toHoriz, toVert: INTEGER);
```

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *ListBox* specifies a list box that does not exist, MoveListBox does nothing. The change is seen immediately providing that the list box is not hidden. The list box's width and height are not changed.

ToHoriz is the new horizontal co-ordinate at which the left side of the list box appears.

ToVert is the new vertical co-ordinate at which the top of the list box appears.

Also see: SizeListBox and MoveSizeListBox.

OffsetListBox

Change a list box's co-ordinates without affecting its image on the window.

C
```
pascal void OffsetListBox (short ListBox,
            short distHoriz, short distVert);
```

Pascal
```
procedure OffsetListBox (ListBox, distHoriz, distVert: INTEGER);
```

When you scroll an area that contains list boxes, first use ScrollRect to scroll the pixel image containing the affected objects in the window. OffsetListBox is used to offset a list box's co-ordinates without altering its image (since ScrollRect has already done so). At this point, the list box's co-ordinates match the scrolled image of the list box. ObscureListBox or KillListBox can be used to hide or delete list boxes that are scrolled out of view.

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *ListBox* specifies a list box that does not exist, OffsetListBox does nothing.

DistHoriz and *distVert* specify the horizontal and vertical amount by which the list box's co-ordinates are offset. Positive numbers are right and down. The list box's co-ordinates are updated but no change is seen.

SizeListBox

Change a list box's size.

C
```
pascal void SizeListBox (short ListBox, short width, short height);
```

Pascal
```
procedure SizeListBox (ListBox, width, height: INTEGER);
```

SizeListBox changes a list box's width and/or height without altering the list box's top or left co-ordinate. The change is seen immediately providing that the list box is not hidden.

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *ListBox* specifies a list box that does not exist, SizeListBox does nothing.

Width and *height* specify the list box's new width and height in pixels. If either parameter is less than 1, SizeListBox does nothing.

Also see: MoveListBox and MoveSizeListBox.

MoveSizeListBox

Change a list box's co-ordinates.

C
```
pascal void MoveSizeListBox (short ListBox,
            short left, short top, short right, short bottom);
```

Pascal
```
procedure MoveSizeListBox (ListBox, left, top, right, bottom: INTEGER);
```

MoveSizeListBox changes any of the list box's four co-ordinates. The change is seen immediately providing that the list box is not hidden. This routine combines the functions of MoveListBox and SizeListBox.

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *ListBox* specifies a list box that does not exist, MoveSizeListBox does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in local co-ordinates that determines the list box's size and location in the window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). If these parameters specify an empty rectangle, MoveSizeListBox does nothing.

Also see: GetListBoxRect.

MoveSizeListBoxRect

Change a list box's co-ordinates.

C
```
pascal void MoveSizeListBoxRect (short ListBox, const Rect *Bounds);
```

Pascal
```
procedure MoveSizeListBoxRect (ListBox: INTEGER; Bounds: RECT);
```

MoveSizeListBoxRect is identical to the MoveSizeListBox routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

AutoMoveSizeListBox

Specify how a list box is automatically moved and/or resized as its window's size is changed.

C
```
pascal void AutoMoveSizeListBox (short ListBox,
            Boolean left, Boolean top, Boolean right, Boolean bottom);
```

Pascal
```
procedure AutoMoveSizeListBox (ListBox: INTEGER;
            left, top, right, bottom: BOOLEAN);
```

ListBox specifies the list box number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *ListBox* specifies a list box that does not exist, AutoMoveSizeListBox does nothing.

The *left*, *top*, *right* and *bottom* parameters specify if that side of the list box is automatically adjusted when the window's size changes. These setting are applied to the list box and are used the next time the window's size changes:

- *left* Does the list box's left side track the window's right edge?
- *top* Does the list box's top track the window's bottom edge?
- *right* Does the list box's right side track the window's right edge?
- *bottom* Does the list box's bottom track the window's bottom edge?

You can think of each *false* value as locking that side of the list box to a fixed co-ordinate regardless of the window's size (this is the default). Each *true* value establishes a fixed distance between that side of the list box and the window's edge. For example, setting only *left* and *right* to *true* makes the list box move horizontally as the window widens and narrows, but the list box does not move vertically when the window's height changes.

If you are setting these values identically for a group of objects, use AutoMoveSize to define the settings then add the appropriate *x*AutoMoveSize constant (such as listAutoMoveSize for list boxes) to the objects' spec as they are created. The objects will adopt the settings specified by the AutoMoveSize routine.

☞ **Warning**: Make sure that you resize objects in a way that makes sense. Don't allow a window to shrink down to a size where objects become unusable or disappear altogether.

GetListBoxHandle

Get a handle to a list box's list record.

C
```
pascal ListHandle GetListBoxHandle (short ListBox);
```

Pascal
```
function GetListBoxHandle (ListBox: INTEGER): ListHandle;
```

This routine returns a standard ListHandle to a list box that was created by a Tools Plus routine. You should never need to use this routine. It is provided for advanced programmers who may have specialized needs. Always use Tools Plus routines to create and manipulate list boxes. If you are using an Appearance Manager List Box control, that is a control created with the listSystemBody option, GetListBoxHandle returns a handle to the control. You can then use toolbox routines to get a handle to the list itself.

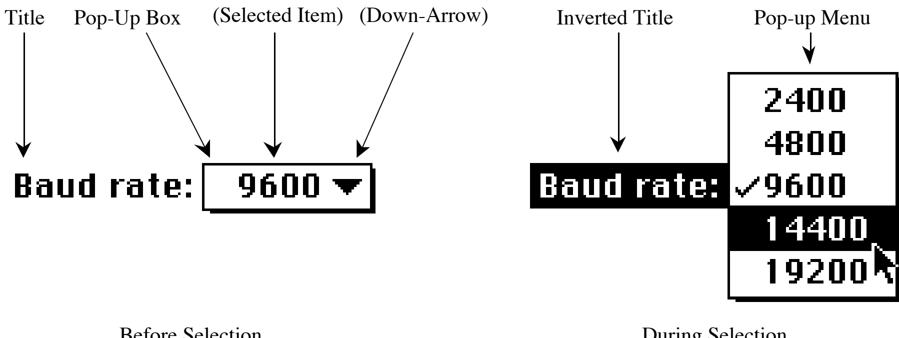
ListBox specifies the list box number (from 1 to 511) in the current window whose handle is being retrieved. If the current window doesn't belong to your application, or if no windows are open, or if *ListBox* specifies a list box that does not exist, GetListBoxHandle returns nil.

☞ **Warning**: If you need to lock the handle or change its attributes, do so temporarily then restore the original settings before using any Tools Plus routines. If you alter this handle or any data that is made accessible by this handle, you do so at your own risk. The only exception is the list box's reference constant (refCon field) which can safely be used to store any value you want.

11 Pop-Up Menus

Pop-up menus are a mechanism that lets the operator make a selection from multiple choices. Where this interface differs from a set of radio buttons, a set of check boxes, or a list box, is that a pop-up menu requires minimal space on a window by hiding most of its detail until it is required. Pop-up menus are typically used for lists of items, such as fonts. See the Macintosh User Interface Guidelines chapter of Inside Macintosh for details on the use of pop-up menus. The implementation of pop-up menus shares many similarities with pull-down menus, so you will find that this chapter has a lot of commonalty with the chapter on Menus.

A pop-up menu is typically made up of two components as illustrated below on the left: a title, and a pop-up box. The pop-up box contains the selected item, and a "down arrow" which provides the user with a visual cue that the control is a pop-up menu. When the user clicks and holds the title or the pop-up box, a list of choices is displayed in a pop-up menu, allowing the user to select one of the items. By default, Tools Plus pop-up menus allow only a single item to be selected, but you can easily override this behavior.

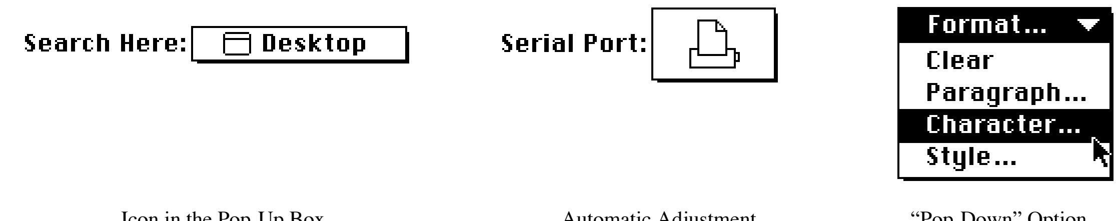


Before Selection

During Selection

Tools Plus's pop-up menus provide several options that are not available on ordinary Macintosh pop-up menus, as illustrated below. One of these options displays the selected item's icon within the pop-up box. As you may notice, you can also suppress the "down arrow" if you want. Tools Plus's pop-up menus also perform automatic adjustments to create the perfect looking pop-up menu without having to calculate font heights.

Another feature that is not available in ordinary Macintosh pop-up menus is the "pop down" option. It displays the menu's list below the pop-up menu's body. If the pop-up menu has a title, it is displayed within the control's body, otherwise the first selected item is displayed in the control's body, like a regular pop-up menu. This feature is useful in a window where space is limited and several "do it now" options are required.



Icon in the Pop-Up Box

Automatic Adjustment

"Pop-Down" Option

In this document, the term *pop-up menu* refers to the entire control; that is, the pop-up box and its contents, the name that appears to the left, and the individual items which appear during selection. The term *menu item* or *item* refers to individual items found within a pop-up menu. The item number is determined by counting from the top of the list, the first item being 1, the second being 2, etc.

A pop-up menu is created on the current window with the NewPopUp routine. Each pop-up menu is referenced by a unique pop-up menu number that can be from 1 to 511. This number is specified when the pop-up menu is created, and refers to the specific pop-up menu until it is deleted. Note that the pop-up menu number is related to its associated window. This means that two different windows can each have a pop-up menu numbered “1” without interfering with each other. Whenever the user makes a selection in the pop-up menu, Tools Plus reports this to your application. You can also create an entire pop-up menu from a ‘MENU’ resource by using the LoadPopUp routine.

The PopUpMenu routine is used to add *items* to a specific pop-up menu, or to rename existing items in a pop-up menu. ResNamesToPopUp inserts resource names (such as fonts or sounds), sorted alphabetically, at a specified item.

Pop-up menu items can also be inserted between others using the InsertPopUpItem routine. This lets your application maintain a dynamic pop-up menu that may be used, for example, for a list of available font sizes.

An entire pop-up menu can be deleted by using the RemovePopUp routine. This routine reclaims the memory used by the pop-up menu. Individual items can also be deleted using this routine.

Pop-up menu items can be renamed by using the RenamePopUp routine. This should be done judiciously, since changes to pop-up menu items may prove to be confusing to the user.

An entire pop-up menu can be enabled or disabled with the EnablePopUp routine, as can individual menu items. When an entire pop-up menu is disabled, it is dimmed and it cannot be selected. Furthermore, its items cannot be displayed. When an item is disabled, it becomes dim and cannot be selected. A pop-up menu can be hidden and displayed using PopUpDisplay.

Various other menu item-related features are supported, such as setting or removing “check marks” with the CheckPopUp routine. You can set or remove other marks with the PopUpMark routine, and determine which mark is displayed by using GetPopUpMark. You can set and retrieve an item’s icon number with PopUpIcon and GetPopUpIcon. An item’s text is retrieved with GetPopUpString, and its style is set with PopUpStyle.

Pop-up menus can be moved to a new location with MovePopUp and have their width changed with SizePopUp. MoveSizePopUp combines both tasks by letting you specify new co-ordinates for the pop-up menu.

Fonts

All pop-up menus default to using the Chicago 12pt font. When a pop-up menu is created, it can optionally adopt and remember the window’s current font, size and style settings (as set by the TextFont, TextSize, and TextFace routines) by including the popupUseWFont option. The window’s settings can then be changed without affecting the pop-up menu. You can use the GetPopUpFontSettings and SetPopUpFontSettings routines to get and set the pop-up menu’s font, size and style settings.

Colors

By default, a pop-up menu has black text on a white background. The control’s frame is also black and the control body is white. The pop-up menu’s items are displayed using black text on a white background. Optionally, each pop-up menu can adopt unique color settings as it is created. The colors for the various parts are defined by the PopUpColors routine, and are optionally adopted by pop-up menus as they are created. Pop-up menus’ colors can be changed afterwards using the SetPopUpColors routine. Conversely, the GetPopUpColors routine retrieves a pop-up menu’s color settings. If you want to get or set the colors for a single menu item, use the GetPopUpItemColors and SetPopUpItemColors routines.

When designing applications, always design them in black and white then apply color (if required) to add value to your application. Don’t add color just because you can. In the case of color pop-up menus, test your color selection thoroughly on a monitor set to 8, 4, and 2-bit color and gray scale, and black and white to ensure that your colors and window backdrop color map to usable colors. In all cases, use color very judiciously, and only if there is value in adding colors.

The Appearance Manager does not support the use of colors in pop-up menus (it supplies colors and patterns that are consistent with the user-selected theme). Initializing Tools Plus with the initPureAppearanceManager option enforces this principle by ignoring custom color information when the Appearance Manager is available.

Command Keys & Hierarchical Pop-Up Menus

Macintosh User Interface Guidelines recommend against using command keys or submenus in a pop-up menu. Tools Plus enforces this to a great degree, but for developers who insist on creating hierarchical pop-up menus, a solution is at hand. The AttachPopUpSubMenu routine lets you attach a hierarchical menu to a pop-up menu (see the Menus chapter for details about creating a hierarchical menu). If you populate your pop-up menu using a 'MENU' resource, Tools Plus recognizes the submenus and attaches them appropriately to the pop-up menu.

When you create hierarchical menus for a pop-up menu, make sure that the hierarchical menu does not contain command key equivalents because Tools Plus ignores them. Also make sure that your hierarchical menu number is in the range of 16 through 200. A hierarchical menu can be shared by numerous pop-up menus, but keep in mind that if you make a change to a shared hierarchical menu, that change shows up in all the pop-up menus in your application that use that hierarchical menu.

Creating a Pop-Up Menu Using a 'CNTL' Resource

Tools Plus offers considerable versatility in the way it supports the creation of pop-up menus from 'CNTL' resources. These features are most often used when opening a dialog ('DLOG' resource) that contains pop-up menus. In all cases, the 'CNTL' resource specifies a CDEF ID of 63, which produces a procID of 1008 plus any variants. Just after you initialize your application, use the SetDialogCNTLPopUpSpec routine to specify the default appearance and behavior specifications ("spec" parameter) for pop-up menus that are created by dialogs. A list of possible values can be found in the NewPopUp description. By default, Tools Plus simply creates a pop-up menu using the system's CDEF, thus providing you with the ease of use that is provided by Tools Plus's pop-up menu routines.

Pure System Pop-Up Menu

For "pure" pop-up menus, that being without any of the advantages of Tools Plus's pop-up menu routines, call SetDialogCNTLPopUpSpec(-1) just after you initialize your application. This causes 'CNTL' resources that reference the pop-up menu CDEF to be implemented as "buttons" instead of being implemented as Tools Plus's pop-up menus. Your application has access to the control's handle via the GetButtonHandle routine. This approach gives you the ultimate control over your pop-up menu. It also makes it the most difficult alternative in terms of programming because you must do all the toolbox coding for the pop-up menu. In this situation, set up the 'CNTL' resource's fields with values as detailed in Inside Macintosh.

Tools Plus Pop-Up Menu (CDEF 63)

A much easier alternative is to create a Tools Plus pop-up menu using the 'CNTL' resource. When you open a dialog, 'CNTL' resources that reference CDEF ID 63 (the pop-up menu) create a Tools Plus pop-up menu. The translation from a 'CNTL' resource to a Tools Plus pop-up menu takes place as follows:

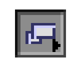
- Tools Plus first looks at the default pop-up menu appearance and behavior specifications, as set by the SetDialogCNTLPopUpSpec. You will likely set this value to something like popupSystemBody, simply telling Tools Plus to create a pop-up menu that looks like the system's pop-up menu.
- If the Macintosh running your application does not have a pop-up menu CDEF as is the case with System 6, Tools Plus will create a standard pop-up menu using its own code.
- To override the default appearance and behavior specifications for a single pop-up menu, place the replacement spec value in the 'CNTL' resource's contrlRfCon field, the reference constant. A list of possible values can be found in the NewPopUp description. A contrlRfCon value of zero (0) indicates that the default appearance and behavior specification is used, as set by the SetDialogCNTLPopUpSpec routine.
- The 'CNTL' resource's contrlMin field (control's minimum limit) is used to specify the 'MENU' resource that is used to name the pop-up menu and to populate it with items. The menu's title is used as the title for the pop-up menu. The 'MENU' resource ID must be in the range of 16000 through 31999.
- If the 'CNTL' resource's contrlMin field (control's minimum limit) is set to zero (0), the contrlTitle field (title) is used for the pop-up menu's title, and the pop-up menu is not populated with menu items. Make sure you disable a pop-up menu that has no items since the user cannot select anything in it.

Bevel Button Pop-Up Menu (CDEF 2)

The Appearance Manager's bevel button control has a number of options that can be implemented as pop-up menus. You create this kind of user interface element by using a 'CNTL' resource in a dialog. When you open the dialog, Tools Plus recognizes this kind of bevel button control as a "special case" and implements it as a pop-up menu instead of a button, thereby providing you with all the advantages and ease of use offered by Tools Plus's pop-up menus.

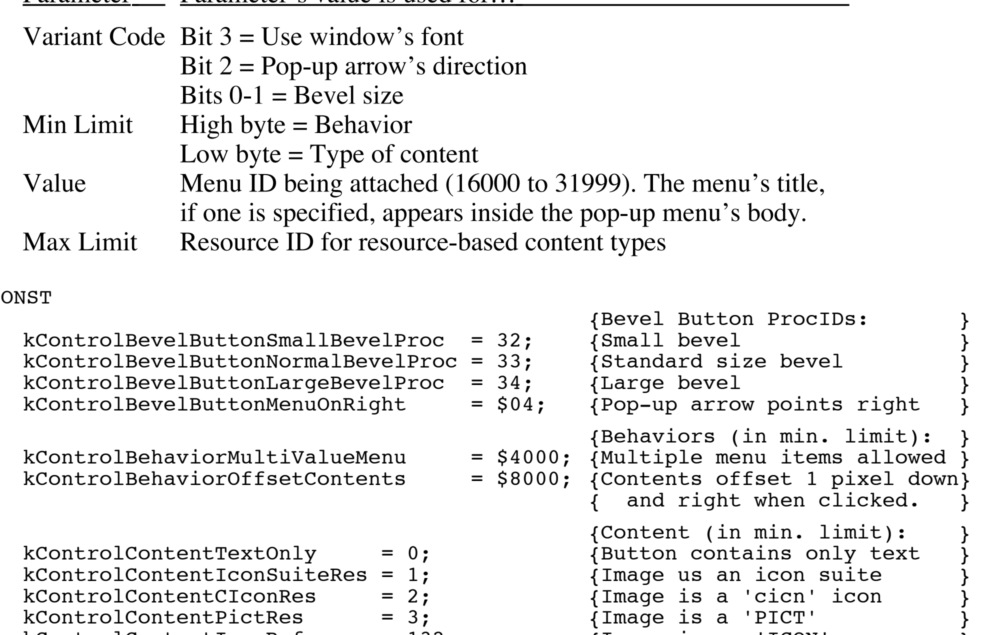
Small, Medium and Large Bevel Buttons

The bevel button is the most versatile control offered by the Appearance Manager. It allows you to specify the button's appearance, its content (picture, icon, etc.), its behavior (push button, toggle, or sticky), and its pop-up menu ID. All these capabilities are invoked by correctly setting the control's variant code, minimum limit, maximum limit, and value. You will use CDEF 2 in all cases, therefore the control's procID will be 32 plus a variant code.



Selected

Deselected



Parameter	Parameter's value is used for…
Variant Code	Bit 3 = Use window's font Bit 2 = Pop-up arrow's direction Bits 0-1 = Bevel size
Min Limit	High byte = Behavior Low byte = Type of content
Value	Menu ID being attached (16000 to 31999). The menu's title, if one is specified, appears inside the pop-up menu's body.
Max Limit	Resource ID for resource-based content types

```
CONST
                                                {Bevel Button ProcIDs:       }
   kControlBevelButtonSmallBevelProc  = 32;     {Small bevel                 }
   kControlBevelButtonNormalBevelProc = 33;     {Standard size bevel         }
   kControlBevelButtonLargeBevelProc  = 34;     {Large bevel                 }
   kControlBevelButtonMenuOnRight     = $04;    {Pop-up arrow points right   }
                                                {Behaviors (in min. limit):  }
   kControlBehaviorMultiValueMenu     = $4000;  {Multiple menu items allowed }
   kControlBehaviorOffsetContents     = $8000;  {Contents offset 1 pixel down}
                                                {  and right when clicked.   }
                                                {Content (in min. limit):    }
   kControlContentTextOnly     = 0;             {Button contains only text   }
   kControlContentIconSuiteRes = 1;             {Image us an icon suite      }
   kControlContentCIconRes     = 2;             {Image is a 'cicn' icon      }
   kControlContentPictRes      = 3;             {Image is a 'PICT'           }
   kControlContentIconRef      = 132;           {Image is an 'ICON'          }
```

Flag your 'CNTL' and 'MENU' resources as purgeable to save memory. Tools Plus makes a copy of their data. See the chapters on Buttons, Scroll Bars, Editing Fields, and List Boxes in this user manual for additional Appearance Manager controls.

Handling Pop-Up Menus

Once a pop-up menu is created, Tools Plus performs all the processing required to maintain it. When a window is inactive, Tools Plus disables all pop-up menus on that window. When the window is activated again, all the pop-up menus regain their correct status as specified by your application. Tools Plus constantly inquires about any events that have occurred, including the user clicking a pop-up menu.

See the Event Management chapter for complete details on the handling of pop-up menus.

NewPopUp

Create a new pop-up menu.

C
```
pascal void NewPopUp (short MenuNumber,
            short left, short top, short right, short bottom,
            const Str255 MenuTitle, long Spec, Boolean EnabledFlag);
```

Pascal
```
procedure NewPopUp (MenuNumber, left, top, right, bottom: INTEGER;
            MenuTitle: STRING; Spec: LONGINT; EnabledFlag: BOOLEAN);
```

This routine just creates the pop-up menu control and its title. Pop-up menu *items* are created with PopUpMenu.

MenuNumber specifies the pop-up menu number (from 1 to 511) that is created in the current window. Once a pop-up menu is created, it is referenced by this pop-up menu number. If a pop-up menu has been previously created in the current window using the same number, it is replaced with a new pop-up menu (without any items) as specified by the parameters in the NewPopUp routine. If the current window doesn't belong to your application, or if no windows are open, NewPopUp does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in the current window's local co-ordinates that determine the pop-up menu's size and location in the window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom) of the pop-up box. The pop-up box's 1-pixel border and its drop shadow are drawn outside these co-ordinates. Also, the pop-up menu's title is drawn to the left of the specified rectangle. To make a pop-up menu operate at its best, its height (difference between top and bottom) must be equivalent to the font's height (font height can be determined by calling the GetFontInfo routine and adding *Ascent* + *Descent* + *Leading*). If you make *bottom* equal to *top*, the bottom is adjusted automatically to the exact font height. If your menu is comprised entirely of icons (no text in the items), set the height of the rectangle to equal the height of the icon, and add 2.

MenuTitle is the pop-up menu's name that appears to the left of the pop-up box, or inside the pop-up box when the "pop down" option is used. You may specify a null string (‘‘) if you do not want to have an external title displayed.

Spec specifies the pop-up menu's appearance and behavior characteristics. The value for this 4-byte long integer can be specified either by adding a set of constants to obtain the desired result, or using a specially defined variant record. See the section below for details.

EnabledFlag indicates if the newly created pop-up menu is enabled or not. When a pop-up menu is disabled, it becomes dim and cannot be selected by the user, nor can its items be viewed. All pop-up menus automatically become disabled when the window containing them is inactive. When the window is activated, the pop-up menus assume their state as set by the NewPopUp routine and subsequent calls to the EnablePopUp routine. The two constants that can be used for this flag are *enabled* and *disabled*.

Appearance and Behavior

Spec specifies the pop-up menu's appearance and behavior characteristics. The value for this 4-byte long integer can be specified either by adding a set of constants to obtain the desired result, or using a specially defined variant record, as illustrated below.

Optionally choose only one of the following pop-up menu styles…

`popupSystemBody` — Use CDEF ID = 63. The pop-up menu CDEF is available in System 7 or later. You can also use a custom CDEF by setting its ID to 63. In System 7, the pop-up menu CDEF looks identical to the regular Tools Plus pop-up menu, but it has fewer features and some bugs (from Apple, not from us). If a CDEF with ID 63 can't be found as is the case in System 6, the standard Tools Plus pop-up menu is used.

The system's pop-up menu CDEF will likely look different in future versions of Mac OS. Using this option will ensure a consistent appearance with Mac OS 8, but you won't be able to use some of the options offered in Tools Plus's pop-up menus.

Note: Only pop-up menus that use a CDEF can be embedded into other controls.

Format

`popup3DBody` — Use a 3D pop-up menu that supports all Tools Plus features. This pop-up menu's appearance is designed to mimic a popular 3D look in Mac OS 8, but it is available on all system versions.

Format

Optionally choose any of the following options…

`popupUseWFont` — Use the window's font for the menu. By default, pop-up menus use the System Font (Chicago 12pt.) You may want to use a smaller font, such as Geneva 9, in windows where space is scarce.

When using this option, the window's current font, size and style settings (as set by the TextFont, TextSize, and TextFace routines) are remembered by the pop-up menu as it is created. The window's font settings (font, size, text-transfer mode, and style) can then be changed without affecting the pop-up menu.

`popupColorPopUp` — Adopt the color settings as defined by the PopUpColors routine. By default, pop-up menus have black text, frame, and items while their body and list background color are white. Colors are ignored by some pop-up menu CDEFs if you use the popupSystemBody option.

`popupHasBackground` — The pop-up menu is drawn on a complex (non-solid) background such as a picture. When this option is used, a doPreRefresh event is generated after the user uses the pop-up menu in to let your application refresh the image behind the pop-up menu. Use this option only when necessary because it is slower. This option is not supported when using popupSystemBody, the system's pop-up menu CDEF.

`popupNeverDimOutline` — Never dim the pop-up box. By default, the pop-up box is dimmed when the pop-up menu is disabled or when its parent window is inactive.

`popupNeverDimSelection` — Never dim the selected item's text (and the optional icon and "down arrow") displayed in the pop-up box. By default, the selected item is dimmed when the pop-up menu is disabled or when its parent window is inactive. This option is useful when using small fonts on a black and white monitor, since those fonts tend to look illegible when dithered.

`popupNeverDimTitle`	Never dim the external title. By default, the external title is dimmed when the pop-up menu is disabled or when its parent window is inactive. This option is useful when using small fonts on a black and white monitor, since those fonts tend to look illegible when dithered.
`popupNoArrow`	Suppress the "down arrow." By default, the "down arrow" is displayed in the pop-up box. This option is not supported when using popupSystemBody, the system's pop-up menu CDEF.
`popupMultiSelect`	Allow multiple items to be selected. By default, pop-up menus allow only a single item to be selected at a time (selecting another item deselects the original one).
`popupIconTitle`	Draw the selected item's icon in the pop-up box. By default, the selected item's icon is not drawn in the pop-up box regardless if icons are used in the pop-up menu or not. This option is not supported when using popupSystemBody, the system's pop-up menu CDEF.
`popupDropDown`	Display the list below the pop-up menu's control. If the pop-up menu has a title it is displayed within the control, otherwise the first selected item in the list is displayed in the control's body.
`popupAutoMoveSize`	Automatically move and/or resize the pop-up menu when the window's size changes. The AutoMoveSize routine lets you specify which sides are altered (the top and bottom parameters set the same to retain the pop-up menu's height). You can use the AutoMoveSizePopUp routine as an alternative to setting this option.
`popupHidden`	Create a hidden pop-up menu. This kind of pop-up menu is accessible to your application but not to the user.
`popupDefaultType`	This constant, if used alone, produces a standard pop-up menu using Chicago 12 that allows one item to be selected at a time. Adding any of the above options overrides default behavior.

So, if you want to create a pop-up menu that uses the window's current font settings instead of Chicago 12, and you wanted the selected item's icon to be displayed in the pop-up box, you should use the combined constants popupUseWFont + popupIconTitle. Alternatively, a C structure and a Pascal variant record are available to help you define the Spec in a more intuitive way, as shown below:

C

```
union TPPopUpMenuSpec {                        /*Pop-Up Menu's appearance and behavior        */
                                               /*  specifications in 2 formats…               */
   struct{                                     /* · Parsed into components:                   */
      unsigned short bit31to20: 12;            /*     (reserved bits)                         */
      unsigned short UseColor: 1;              /*    Use color settings                       */
      unsigned short bit18to16: 3;             /*     (reserved bits)                         */
      unsigned short Hidden: 1;                /*    Create a hidden Pop-Up Menu              */
      unsigned short AutoMoveSize: 1;          /*    Auto-resize as window's size chg         */
      unsigned short BodyIs3D: 1;              /*    Body is drawn in 3D style                */
      unsigned short SystemBody: 1;            /*    Use system's standard body style         */
      unsigned short bit11to10: 2;             /*     (reserved bits)                         */
      unsigned short HasBackground: 1;         /*    Pop-Up Menu is drawn over an image       */
      unsigned short NeverDimOutline: 1;       /*    Never dim the control's outline          */
      unsigned short NeverDimSelectedItem: 1;  /*    Never dim the selected item's text       */
      unsigned short NeverDimTitle: 1;         /*    Never dim the title                      */
      unsigned short NoArrow: 1;               /*    Is the "down arrow" hidden               */
      unsigned short MultipleSelections: 1;    /*    Allow multiple items to be selected      */
      unsigned short UseWindowFont: 1;         /*    Display using window's font              */
      unsigned short IconInTitle: 1;           /*    Draw icon in the control's title         */
      unsigned short DropDown: 1;              /*    Drop list down from control              */
      unsigned short bit0: 1;                  /*     (reserved bit)                          */
   } Bits;                                     /*                                             */
   long Num;                                   /* · Long equivalent                           */
};                                             /*                                             */
typedef union TPPopUpMenuSpec TPPopUpMenuSpec;
```

Pascal

```
TPPopUpMenuSpec = packed record            {Pop-Up Menu's appearance and behavior           }
                                           {  specifications in 2 formats…                  }
    case integer of                        {                                                }
     0: (                                  { · Parsed into components:                      }
       bit31, bit30, bit29, bit28: boolean; {      (reserved bits)                          }
       bit27, bit26, bit25, bit24: boolean; {      (reserved bits)                          }
       bit23, bit22, bit21, bit20: boolean; {      (reserved bits)                          }
       UseColor: boolean;                  {     Use color settings                         }
       bit18, bit17, bit16: boolean;       {      (reserved bits)                           }
       Hidden: boolean;                    {     Create a hidden pop-up menu                }
       AutoMoveSize: boolean;              {     Auto-move/size as window's size changes    }
       BodyIs3D: boolean;                  {     Body is drawn in 3D style                  }
       SystemBody: boolean;                {     Use the system's standard body style       }
       bit11, bit10: boolean;              {      (reserved bits)                           }
       HasBackground: boolean;             {     Pop-up menu is drawn over an image         }
       NeverDimOutline: boolean;           {     Never dim the control's outline            }
       NeverDimSelectedItem: boolean;      {     Never dim the selected item's text         }
       NeverDimTitle: boolean;             {     Never dim the title                        }
       NoArrow: boolean;                   {     Is the "down arrow" hidden                 }
       MultipleSelections: boolean;        {     Allow multiple items to be selected        }
       UseWindowFont: boolean;             {     Display using window's font                }
       IconInTitle: boolean;               {     Draw icon in the control's title           }
       DropDown: boolean;                  {     Drop list down from control                }
       bit0: boolean;                      {      (reserved bit)                            }
     );                                    {                                                }
     1: (                                  { · Longint equivalent:                          }
       Num: longint;                       {     Specification longint                      }
     );                                    {                                                }
   end;
```

As an example, lets create a pop-up menu that uses the window's current font and displays the icon in the title. The following code sample illustrates how this is done:

```
procedure DoItNow;
  var
    Spec: TPPopUpMenuSpec;                {Define the variable used for the Spec            }
  begin
    Spec.Num := 0;                        {Initialize all the bits to zero values           }
    Spec.UseWindowFont := true;           {Specify that the window font is to be used       }
    Spec.IconInTitle := true;             {Specify that the selected item's icon appears in }
                                          {  the pop-up box.                                }
                                          {Create the pop-up menu using the integer part of }
    NewPopUp(1, 110, 20, 209, 20, 'Day of Week:', Spec.Num, enabled);  {  the Spec.        }
```

You can use whatever you like best as the Spec, a single constant, several constants added together, a variable, or the short or 4-byte integer component of a structure or variant record.

Pop-Up Menus on Color Backgrounds

Sometimes it may be necessary to place a pop-up menu on a color surface, such as a tool bar. If you are creating a pop-up menu on a color surface, set the window's background color (by using SetBackRGB) to the color on which the pop-up menu is being created, then create the menu. You may change the window's foreground and background colors at any time without affecting pop-up menus.

Each pop-up menu remembers the color on which it is created, and uses this color when any erasing is performed by the pop-up menu. An example of this is when the user clicks and holds the pop-up menu. In such a case, the control's body temporarily disappears and is replaced by the pop-up menu's list of items.

Also see: NewPopUpRect, NewDialogPopUp and PopUpMenu.

Programming Tips:

1 If you want to create a menu that is comprised entirely of icons (without any text items), make sure all the icons have the same height, then make the pop-up menu's height equal to the icon height plus 2.

```
CONST                                            {Pop-Up Menu Behavior and Appearance Specs:   }
               popupColorPopUp        = $00080000;  {Use color settings for this pop-up menu   }
               popupHidden            = $00008000;  {Create a hidden pop-up menu?              }
               popupAutoMoveSize      = $00004000;  {Auto-resize as window's size changes      }
               popup3DBody            = $00002000;  {Draw body using 3D style                  }
```

```
popupSystemBody        = $00001000; {Draw the system's standard body style       }
popupHasBackground     = $00000200; {Pop-Up Menu is drawn over an image?         }
popupNeverDimOutline   = $00000100; {Never dim the control's outline?            }
popupNeverDimSelection = $00000080; {Never dim the selected item's text?         }
popupNeverDimTitle     = $00000040; {Never dim the title?                        }
popupNoArrow           = $00000020; {Is the "down arrow" hidden?                 }
popupMultiSelect       = $00000010; {Allow multiple items to be selected?        }
popupUseWFont          = $00000008; {Use the window's font for the menu?         }
popupIconTitle         = $00000004; {Draw icon in the control's title?           }
popupDropDown          = $00000002; {Drop list down from control                 }
popupDefaultType       = $00000000; {Default menu (sys font, 1 item, no icon)    }
```

NewPopUpRect

Create a new pop-up menu.

C
```
pascal void NewPopUpRect (short MenuNumber, const Rect *Bounds,
              const Str255 MenuTitle, long Spec, Boolean EnabledFlag);
```

Pascal
```
procedure NewPopUpRect (MenuNumber: INTEGER; Bounds: RECT;
              MenuTitle: STRING; Spec: LONGINT; EnabledFlag: BOOLEAN);
```

NewPopUpRect is identical to the NewPopUp routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

LoadPopUp

Create a new pop-up menu using a ‘MENU’ resource.

C
```
pascal void LoadPopUp (short MenuNumber, short left, short top, short right,
              short bottom, long Spec, short ResID);
```

Pascal
```
procedure LoadPopUp (MenuNumber, left, top, right, bottom: INTEGER;
              Spec: LONGINT; ResID: INTEGER);
```

LoadPopUp is identical to the NewPopUp routine, except that it uses a ‘MENU’ resource to populate the pop-up menu. The ‘MENU’ resource contains the pop-up menu’s title. If the title is disabled then the pop-up menu is disabled. The remainder of the ‘MENU’ resource specifies the pop-up menu’s items.

ResID is the ‘MENU’ resource ID number that is used to create the pop-up menu. If the menu has an ‘mctb’ color table resource, it must use the same ID number. The resource ID number must be in the range of **16000** to **31999**. These resource numbers don’t overlap the range used by menu numbers, so you can think of them as a temporary holding area for ‘MENU’ resources that have not become usable menus.

When creating pop-up menus using ‘MENU’ resources, please note the following:

- Flag your ‘MENU’ and ‘mctb’ resources as purgeable to save memory. Tools Plus makes a copy of their data.
- Submenus must be in the range of 16 to 200.
- Command key equivalents are cleared because they are not supported in pop-up menus in Tools Plus.

LoadPopUpRect

Create a new pop-up menu using a 'MENU' resource.

C
```
pascal void LoadPopUpRect (short MenuNumber, const Rect *Bounds, long Spec,
           short ResID);
```

Pascal
```
procedure LoadPopUpRect (MenuNumber: INTEGER; Bounds: RECT; Spec: LONGINT;
           ResID: INTEGER);
```

LoadPopUpRect is identical to the LoadPopUp routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

LoadDialogPopUp

Create a new pop-up menu in a dialog using a 'MENU' resource and a dialog item's co-ordinates.

C
```
pascal void LoadDialogPopUp (short MenuNumber, long Spec, short ResID);
```

Pascal
```
procedure LoadDialogPopUp (MenuNumber: INTEGER; Spec: LONGINT;
           ResID: INTEGER);
```

LoadDialogPopUp is identical to the LoadPopUp routine, except that the pop-up menu is created in a dialog (a window opened with the LoadDialog routine, or one that had a dialog list attached with the LoadDialogList routine). The pop-up menu's co-ordinates are obtained from the dialog item whose number matches the pop-up menu number.

EmbedPopUpInButton

Embed a pop-up menu into a button or into the window's root control (Appearance Manager only).

C
```
pascal void EmbedPopUpInButton (short MenuNumber, short ContainerButton);
```

Pascal
```
procedure EmbedPopUpInButton (MenuNumber, ContainerButton: INTEGER);
```

The Appearance Manager lets you embed a control into a parent control such that when the parent is hidden or disabled, all embedded controls are similarly affected. All Tools Plus routines that load a dialog item list (LoadDialog, LoadSpecDialog, LoadDialogList, etc.) automatically embed controls at all times. EmbedPopUpInButton lets you manually embed a pop-up menu into a button, or into the window's root control. Note that the term "button" does not literally mean a button control. It means any control that is implemented as a button in Tools Plus. The most likely candidate is a Group Box control. If the Appearance Manager is not available, EmbedPopUpInButton does nothing.

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the pop-up menu does not exist in the current window, EmbedPopUpInButton does nothing. Note that the only pop-up menus that can be embedded are those that are drawn using a CDEF (use the popupSystemBody option when creating the pop-up menu).

ContainerButton specifies the button number (from 1 to 511) into which *MenuNumber* is embedded. This control must exist in the current window, and it must be a "container" type control such as the Appearance Manager's Group Box. The pop-up menu must fit entirely within the container control or EmbedPopUpInButton does nothing. If a value of 0 is provided for a container button, *MenuNumber* is embedded into the window's root control.

Also see: EmbedPopUpInScrollBar and SetAutoEmbed.

EmbedPopUpInScrollBar

Embed a pop-up menu into a scroll bar or into the window's root control (Appearance Manager only).

C
```
pascal void EmbedPopUpInScrollBar (short MenuNumber,
            short ContainerScrollBar);
```

Pascal
```
procedure EmbedPopUpInScrollBar (MenuNumber, ContainerScrollBar: INTEGER);
```

The Appearance Manager lets you embed a control into a parent control such that when the parent is hidden or disabled, all embedded controls are similarly affected. All Tools Plus routines that load a dialog item list (LoadDialog, LoadSpecDialog, LoadDialogList, etc.) automatically embed controls at all times. EmbedPopUpInScrollBar lets you manually embed a pop-up menu into a scroll bar, or into the window's root control. Note that the term "scroll bar" does not literally mean a scroll bar control. It means any control that is implemented as a scroll bar in Tools Plus. If the Appearance Manager is not available, EmbedPopUpInScrollBar does nothing.

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the pop-up menu does not exist in the current window, EmbedPopUpInScrollBar does nothing. Note that the only pop-up menus that can be embedded are those that are drawn using a CDEF (use the popupSystemBody option when creating the pop-up menu).

ContainerScrollBar specifies the scroll bar number (from 1 to 511) into which *MenuNumber* is embedded. This control must exist in the current window, and it must be a "container" type control. The pop-up menu must fit entirely within the container control or EmbedPopUpInScrollBar does nothing. If a value of 0 is provided for a container scroll bar, *MenuNumber* is embedded into the window's root control.

Also see: EmbedPopUpInButton and SetAutoEmbed.

GetFreePopUpNum

Get the first unused pop-up menu number.

C
```
pascal short GetFreePopUpNum (void);
```

Pascal
```
function GetFreePopUpNum: INTEGER;
```

Some developers may prefer to write code that more closely resembles a traditional Macintosh application, in that creating an object returns a reference to it such as a handle or pointer. Instead of having to assign your own pop-up menu number, GetFreePopUpNum returns the first unused (free) pop-up menu number. Using this routine, you can assign an unused pop-up menu number to a variable, then use that variable throughout your application without concern for the true pop-up menu number.

GetFreePopUpNum returns the first free pop-up menu number on the current window. If the current window doesn't belong to your application, if no windows are open, or if the maximum number of pop-up menus has already been created on the current window (no new ones can be created), GetFreePopUpNum returns a value of zero (0).

AttachPopUpSubMenu

Attach a hierarchical menu to a pop-up menu item, or detach a hierarchical menu from a pop-up menu item.

C
```
pascal void AttachPopUpSubMenu (short MenuNumber, short ItemNumber,
            short SubMenuNumber);
```

Pascal
```
procedure AttachPopUpSubMenu (MenuNumber, ItemNumber,
            SubMenuNumber: INTEGER);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, AttachPopUpSubMenu does nothing.

ItemNumber specifies the menu item number (from 1 to 32767) that is affected. If the item number does not exist within the pop-up menu specified by MenuNumber, AttachPopUpSubMenu does nothing.

SubMenuNumber specifies the "offspring" menu number (from 16 to 200) that is attached to the pop-up menu. You can specify zero (0) to detach a submenu from a known parent pop-up menu item. If the submenu number does not exist, AttachPopUpSubMenu does nothing.

Note: When a submenu is attached to a parent pop-up menu's item, that item's "mark" (as defined by MenuMark) is cleared. Also, if an SICN icon is displayed in the item, it too is cleared. The Macintosh's Menu Manager uses these characters to make hierarchical menus work.

PopUpColors

Set the colors for new pop-up menus as they are created.

C
```
pascal PopUpColors (const RGBColor *Title, const RGBColor *Frame,
            const RGBColor *Body, const RGBColor *DfltItemText,
            const RGBColor *ListBackground);
```

Pascal
```
procedure PopUpColors (Title, Frame, Body, DfltItemText,
            ListBackground: RGBColor);
```

When new pop-up menus are created, by default they have a black frame and text and the control's body is white. The list's text is black on a white background. When you use the PopUpColors routine, new pop-up menus adopt the colors specified in this routine (providing that the pop-up menu is created with the popupColorPopUp option in the pop-up menu's spec). This is the most efficient way to color multiple pop-up menus using the same colors.

Title is the color of the pop-up menu's title, which may be external to the control or a fixed title within the control.

Frame is the pop-up menu's frame color.

Body is the pop-up menu's body color. This is the color that is used to fill the control's body.

DfltItemText is the default color used to display items in the pop-up menu's list.

ListBackground is the background color used for the pop-up menu's list.

Also see: NoPopUpColors and SetPopUpColors.

NoPopUpColors

Reset the colors for new pop-up menus to the default.

C
```
pascal void NoPopUpColors (void);
```

Pascal
```
procedure NoPopUpColors;
```

When new pop-up menus are created, by default they have a black frame and text and the control's body is white. The list's text is black on a white background. When you use the PopUpColors routine, new pop-up menus adopt the colors specified by that routine (providing that the pop-up menu is created with the popupColorPopUp option in the pop-up menu's spec).

This routine resets the settings of the PopUpColors routine to the default values (black title, frame and item text, white body and list background). It is seldom required since you can create default pop-up menus by simply excluding the popupColorPopUp constant from the pop-up menu's spec parameter.

Also see: PopUpColors.

PopUpMenu

Create a pop-up menu, add more items, or rename existing items.

C
```
pascal void PopUpMenu (short MenuNumber, short ItemNumber,
            Boolean EnabledFlag, const Str255 MenuText);
```

Pascal
```
procedure PopUpMenu (MenuNumber, ItemNumber: INTEGER;
            EnabledFlag: BOOLEAN; MenuText: STRING);
```

After a pop-up menu is created with NewPopUp or NewPopUpRect, pop-up menu items can then be added to the menu. Your application should define items in their correct order (i.e., top to bottom) in order to use the full power of this routine.

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, PopUpMenu does nothing.

ItemNumber specifies the pop-up menu's item number (from 1 to 32767) that is affected.

EnabledFlag specifies whether the menu item is enabled or disabled. The menu item can be selected only when enabled. When disabled, the pop-up menu item is dimmed and cannot be selected by the user. The two constants that can be used for this purpose are *enabled* and *disabled.* Pop-up menus and their items can be enabled and disabled by using the EnablePopUp routine.

MenuText is the pop-up menu item's name. If you specify a null string (length equal to zero), Tools Plus will insert a space to prevent anomalous behavior. When a pop-up menu item is first created, certain *metacharacters* are recognized by Tools Plus to provide special instructions to the Menu Manager. You may choose to include or exclude these characters within MenuText, however, you should be aware of their effects. Pop-up menu items can include multiple metacharacters.

✍ **Note**: The Macintosh's Menu Manager allows only the first 31 items of a pop-up menu to be disabled individually. The entire pop-up menu, however, can always be disabled.

Metacharacters

Metacharacters are symbols that tell the Menu Manager to perform special functions on a menu. They are recognized and processed only when a *menu item* is first created, and are ignored (displayed as ordinary characters) when menu items are renamed. Menu items can include multiple metacharacters or combinations of metacharacters.

Unlike the Macintosh toolbox's menu routines, Tools Plus removes the semi-colon (;) and Return character ($0D), and does not process them as metacharacters.

Metacharacter	Meaning
^	Display an icon to the left of the menu item. The number following the caret (^) should be from 1 to 255 (i.e., "^28"). The Menu Manager adds 256 to the number you state to specify a resource ID that is in the range of 257 to 511, so if you specify 28, resource ID 284 is used (28 + 256 = 284). These icon resources are read from your application. Tools Plus tries to use a 'cicn' icon if Color QuickDraw is available on the Macintosh running your application. Otherwise, it will search for an 'ICON' (black and white) icon, then an 'SICN' icon. Unlike the equivalent Macintosh toolbox routine, your menu item will remain unaffected if the specified icon can't be found (i.e., empty space is not reserved in the menu). Be aware that the Menu Manager drawing a 'cicn' icon in color will do so even if the icon was created using 8-bit colors and the monitor is set to 4-bits. This may produce unsatisfactory results. If possible, use 4-bit colors or colors that translate well into 4-bit colors.
!	Display a special mark to the left of a menu item. The single character that follows the exclamation mark (!) is displayed. The check mark is the default. (It is best to use the CheckPopUp or PopUpMark routines.)
<	The item is displayed in a special character style. The single character that follows this symbol specifies the style (Bold, Italic, Underline, Outline, or Shadow). Multiple styles can be combined, such as "<B<I" for "bold and italic." It is best to use PopUpStyle to change styles.

To create a "dividing line" between sections of related pop-up menu items, disable the item and use '-' (a minus or dash) as the MenuText value. You can use the constant mDividingLine for this purpose.

Special care should be taken to create pop-up menu items in their correct order. If any items are skipped when defining a pop-up menu item (i.e., creating item 3 without first creating 1 and 2) the missing pop-up menu items (1 and 2) are automatically created as blank, disabled items. Consequently, metacharacters will not be recognized when the PopUpMenu routine references these automatically created items; the PopUpMenu routine will simply rename the existing item.

Programming Tips:

1. If you are creating a pop-up menu that contains font names, be aware that some Macintoshes have some fonts in ROM. That means that calling CountResources('FOND') will include not only the number of fonts in your system, but in ROM too. Before you add a font name to your pop-up menu, check to see if it already exists to avoid duplicates.
2. If you need any of the metacharacters to appear in an item's text (such as an exclamation mark), first create a blank item (MenuText equals a space), then change the item's text with the PopUpMenu or RenamePopUp routine to include the desired characters. Metacharacters are displayed but not specially processed when an item's name is changed.
3. If your pop-up menu contains icons, and the menu displays the icon in the title, you must exercise some care in your design of icons. Make sure the icon is no wider than 16 pixels, and two pixels shorter than the font height you are using. For the System Font, Chicago 12 pt, your icons must be 14 pixels high (or shorter), and no more than 16 pixels wide.

```
CONST                          {Menu and Menu Item status              }
               enabled       = true;  {enable the menu/item              }
               disabled      = false; {disable the menu/item             }
               mDividingLine = '-';   {Dividing line                     }
```

InsertPopUpItem

Insert an item into an existing pop-up menu.

C
```
pascal void InsertPopUpItem (short MenuNumber, short ItemNumber,
            Boolean EnabledFlag, const Str255 MenuText);
```

Pascal
```
procedure InsertPopUpItem (MenuNumber, ItemNumber: INTEGER;
            EnabledFlag: BOOLEAN; MenuText: STRING);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, InsertPopUpItem does nothing.

ItemNumber specifies the pop-up menu's item number (from 1 to 32767) where the item is inserted. If the pop-up menu item does not exist in the specified pop-up menu, InsertPopUpItem does nothing. InsertPopUpItem will *append* one item to the end of a pop-up menu if the ItemNumber equals the current number of items plus 1.

EnabledFlag specifies whether the item is enabled or disabled. In the enabled state, the item can be selected whereas in the disabled state, the item is dimmed and cannot be selected by the user. The two constants that can be used for this purpose are *enabled* and *disabled.* An entire pop-up menu and individual pop-up menu items can be enabled and disabled by using the EnablePopUp routine.

MenuText is the name of the item. Certain *metacharacters* are recognized by Tools Plus to provide special instructions to the Menu Manager. You may choose to include or exclude these characters within MenuText, however, you should be aware of their effects. See the PopUpMenu routine for details on metacharacters.

When the item is inserted, all existing items starting at ItemNumber are pushed down one space to make room for the new item. This means that their item number will be changed. The new item is inserted at the location specified by ItemNumber. The main use for this routine is to let your application maintain a dynamic menu, such as a list of open document names.

Note: The Macintosh's Menu Manager allows only the first 31 items of a pop-up menu to be disabled individually. The entire pop-up menu, however, can always be disabled.

```
CONST                             {Menu and Menu Item status              }
          enabled       = true;   {enable the menu/item                   }
          disabled      = false;  {disable the menu/item                  }
          mDividingLine = '-';    {Dividing line                          }
```

ResNamesToPopUp

Insert resource names into a pop-up menu.

C
```
pascal void ResNamesToPopUp (short MenuNumber, short ItemNumber,
            ResType rType);
```

Pascal
```
procedure ResNamesToPopUp (MenuNumber, ItemNumber: INTEGER;
            rType: RESTYPE);
```

This routine finds all named resources of the specified type and inserts those names (sorted alphabetically) into a pop-up menu. Duplicated names are ignored as are ones that start with "." (period) or "%".

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, ResNamesToPopUp does nothing.

ItemNumber specifies the pop-up menu item number (from 1 to 32767) where the resource names are inserted. If the pop-up menu item does not exist in the specified pop-up menu, ResNamesToPopUp does nothing. This routine will *append* to the end of a pop-up menu if the ItemNumber equals the current number of items plus 1.

rType is the four character resource type whose names are being inserted into the pop-up menu.

When the resource names are inserted, all existing items starting at ItemNumber are pushed down to make room for the new items. This means that their item number will be changed. The new items are inserted starting at the location specified by ItemNumber. If you specify 'FOND' or 'FONT' resources, both are obtained since they are just different types of fonts.

Note: If the first character of a resource name is a dash (-), it is added into the menus as an option-dash (character 208) to prevent the Menu Manager from interpreting the name as a dividing line.

RemovePopUp

Delete a pop-up menu and its associated items, or delete an individual pop-up menu item.

C
```
pascal void RemovePopUp (short MenuNumber, short ItemNumber);
```

Pascal
```
procedure RemovePopUp (MenuNumber, ItemNumber: INTEGER);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, RemovePopUp does nothing.

ItemNumber specifies the pop-up menu's item number (from 1 to 32767) that is deleted. If ItemNumber is zero (0), RemovePopUp refers to the pop-up menu and all its associated items. If ItemNumber is not zero and it does not exist, RemovePopUp does nothing.

Use KillPopUp if you want to delete the pop-up menu without removing its image from the window.

Note: Use RemovePopUp to maintain a dynamic menu, such as a list of available font sizes. Do not use it to make items unavailable. Instead, disable items with EnablePopUp.

ClearPopUp

Delete all items from a pop-up menu.

C
```
pascal void ClearPopUp (short MenuNumber);
```

Pascal
```
procedure ClearPopUp (MenuNumber: INTEGER);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, ClearPopUp does nothing.

KillPopUp

Delete a pop-up menu without affecting its image on the window.

C
```
pascal void KillPopUp (short MenuNumber);
```

Pascal
```
procedure KillPopUp (MenuNumber: INTEGER);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is deleted in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, KillPopUp does nothing.

KillPopUp is similar to RemovePopUp except that it does not remove the pop-up menu's image from the window. This routine is useful for scrolling pop-up menus in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetPopUp repositions the pop-up menu's co-ordinates without affecting its image (since ScrollRect has already moved it). KillPopUp then deletes the pop-up menus that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

GetPopUpRect

Get a pop-up menu's co-ordinates.

C
```
pascal void GetPopUpRect (short MenuNumber, Rect *Bounds);
```

Pascal
```
procedure GetPopUpRect (MenuNumber: INTEGER; var Bounds: RECT);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is queried in the current window.

Bounds returns the pop-up menu's bounding rectangle specified in the window's local co-ordinates. These co-ordinates match those used to create the pop-up menu. If the current window doesn't belong to your application, or if no windows are open, or if the pop-up menu does not exist in the current window, *Bounds* returns with all co-ordinates set to zero (0).

PopUpDisplay

Hide or show a pop-up menu.

C
```
pascal void PopUpDisplay (short MenuNumber, Boolean Show);
```

Pascal
```
procedure PopUpDisplay (MenuNumber: INTEGER; Show: BOOLEAN);
```

PopUpDisplay hides or shows a pop-up menu on the current window. The result is seen immediately. Use discretion with this routine since pop-up menus should be enabled and disabled to indicate if they are accessible by the user.

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the pop-up menu does not exist in the current window, PopUpDisplay does nothing.

Show indicates if the pop-up menu is being hidden or displayed. The two constants that can be used for this flag are *on* and *off*.

PopUpIsVisible

Determine if a pop-up menu is visible.

C
```
pascal Boolean PopUpIsVisible (short MenuNumber);
```

Pascal
```
function PopUpIsVisible (MenuNumber: INTEGER): BOOLEAN;
```

PopUpIsVisible reports if a pop-up menu (or a control that is implemented as a pop-up menu) is visible on the current window, or if it is hidden.

MenuNumber specifies the pop-up menu number (from 1 to 511) that is queried in the current window.

This routine's value returns *true* if the pop-up menu is visible, and *false* if the pop-up menu is hidden. If the current window doesn't belong to your application, or if no windows are open, or if the pop-up menu does not exist in the current window, PopUpIsVisible returns *false*. This routine takes control embedding into account, so it will return *false* if the target pop-up menu is embedded and its container control is hidden.

ObscurePopUp

Hide a pop-up menu without removing its image from the window.

C
```
pascal void ObscurePopUp (short MenuNumber);
```

Pascal
```
procedure ObscurePopUp (MenuNumber: INTEGER);
```

ObscurePopUp hides a pop-up menu on the current window without removing its image from the window. This routine is useful for scrolling pop-up menus in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetPopUp repositions the pop-up menu's co-ordinates without affecting its image (since ScrollRect has already moved it). ObscurePopUp then hides the pop-up menus that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

MenuNumber specifies the pop-up menu number (from 1 to 511) that is hidden in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the pop-up menu does not exist in the current window, ObscurePopUp does nothing.

GetPopUpString

Get a pop-up menu item's text without the metacharacters.

C
```
pascal void GetPopUpString (short MenuNumber, short ItemNumber,
            Str255 MenuText);
```

Pascal
```
procedure GetPopUpString (MenuNumber, ItemNumber: INTEGER;
            var MenuText: Str255);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window.

ItemNumber specifies the pop-up menu item number (from 1 to 32767) from which the text is obtained.

MenuText specifies the pop-up menu item's name. If the specified pop-up menu does not exist in the current window, or the specified item doesn't exist, MenuText returns as a null string (length is zero). Note that the string will return as a single space (' ') if a null string was specified when the item was created (this happens automatically to prevent the Menu Manager from crashing).

RenamePopUp

Rename an existing pop-up menu item.

C
```
pascal void RenamePopUp (short MenuNumber, short ItemNumber,
            const Str255 MenuText);
```

Pascal
```
procedure RenamePopUp (MenuNumber, ItemNumber: INTEGER; MenuText: STRING);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, RenamePopUp does nothing.

ItemNumber specifies the menu item number (from 1 to 32767) which is changed. If the item number does not exist within the pop-up menu specified by MenuNumber, RenamePopUp does nothing.

MenuText specifies the pop-up menu item's new name. The item's state (enabled/disabled), style (bold, underline, etc.), icon and command key equivalent are not changed. Metacharacters are not interpreted by this routine.

RenamePopUp does not change the pop-up menu's title. If the pop-up menu's title must be changed, the affected pop-up menu must be removed with the RemovePopUp routine, then re-created as required by using the NewPopUp routine.

EnablePopUp

Enable or disable a pop-up menu or pop-up menu item.

C
```
pascal void EnablePopUp (short MenuNumber, short ItemNumber,
            Boolean EnabledFlag);
```

Pascal
```
procedure EnablePopUp (MenuNumber, ItemNumber: INTEGER;
            EnabledFlag: BOOLEAN);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, EnablePopUp does nothing.

ItemNumber specifies the menu item number (from 1 to 32767) which is enabled/disabled. A value of zero (0) affects the entire pop-up menu. If ItemNumber is not zero and the item number does not exist within the menu specified by MenuNumber, EnablePopUp does nothing.

EnabledFlag specifies whether the pop-up menu/item is enabled or disabled. In the enabled state, the menu/item can be selected. The two constants that can used for this purpose are *enabled* and *disabled.* If the ItemNumber is zero, the entire pop-up menu is disabled and the items cannot be viewed. When the pop-up menu later becomes enabled, all items in the pop-up menu assume their correct enabling/disabling as specified by your application.

Note: The Macintosh's Menu Manager allows only the first 31 items of a pop-up menu to be disabled individually. The entire pop-up menu, however, can always be disabled.

```
CONST                        {Menu and Menu Item status                  }
            enabled  = true;   {enable the item                            }
            disabled = false;  {disable the item                           }
```

PopUpIsEnabled

Determine if a pop-up menu is enabled or disabled.

C
```
pascal Boolean PopUpIsEnabled (short MenuNumber);
```

Pascal
```
function PopUpIsEnabled (MenuNumber: INTEGER): BOOLEAN;
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is queried in the current window.

The routine's value returns *true* if the pop-up menu is enabled, and *false* if the pop-up menu is disabled. If the current window doesn't belong to your application, or if no windows are open, or if the pop-up menu does not exist in the current window, PopUpIsEnabled returns *false*. PopUpIsEnabled returns the pop-up menu's enabled state as it is currently displayed, so if the pop-up menu's window is inactive and has temporarily disabled the pop-up menu, PopUpIsEnabled returns *false*.

CheckPopUp

Display or hide a check mark to the left of a menu item.

C
```
pascal void CheckPopUp (short MenuNumber, short ItemNumber, Boolean checked);
```

Pascal
```
procedure CheckPopUp (MenuNumber, ItemNumber: INTEGER; checked: BOOLEAN);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, CheckPopUp does nothing.

ItemNumber specifies the pop-up menu item number (from 1 to 32767) that is affected. If the item number does not exist within the pop-up menu specified by MenuNumber, CheckPopUp does nothing.

Checked specifies whether the pop-up menu item's check mark is displayed or hidden. The two constants that can be used for this purpose are *on* and *off*. By default, pop-up menus can have only one item selected at a time. Therefore, placing a check mark beside an item unchecks the previously checked item. See the NewPopUp routine if you want to override this behavior.

To display characters other than the standard check mark, use the PopUpMark routine.

```
CONST                          {Menu Item check mark status                }
               on  = true;     {check mark is on                           }
               off = false;    {check mark is off                          }
```

PopUpMark

Display or hide a special character to the left of a pop-up menu item's name. Use this routine instead of CheckPopUp to display or hide characters other than the standard check mark.

C
```
pascal void PopUpMark (short MenuNumber, short ItemNumber, char markChar);
```

Pascal
```
procedure PopUpMark (MenuNumber, ItemNumber: INTEGER; markChar: CHAR);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, PopUpMark does nothing.

ItemNumber specifies the pop-up menu item number (from 1 to 32767) that is affected. If the item number does not exist within the pop-up menu specified by MenuNumber, PopUpMark does nothing.

MarkChar specifies the character that is to be displayed. By default, pop-up menus can have only one item selected at a time. Therefore, placing a mark beside an item unmarks the previously marked item. See the NewPopUp routine if you want to override this behavior. The following constants are available for pop-up menu marks:

```
CONST                              {Menu Item characters              }
              AppleChar   = char($14);   {Apple character                   }
              CheckChar   = char($12);   {Check Mark character              }
              DiamondChar = char($13);   {Diamond character                 }
              DotChar     = char($A5);   {Dot (or bullet) character         }
              NoChar      = char($00);   {No character (remove a character) }
```

GetPopUpMark

Get a pop-up menu item's special character that is optionally displayed to the left of an item's name.

C

```
pascal void GetPopUpMark (short MenuNumber, short ItemNumber,
              char *markChar);
```

Pascal

```
procedure GetPopUpMark (MenuNumber, ItemNumber: INTEGER; var markChar: CHAR);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) in the current window that contains the desired item.

ItemNumber specifies the pop-up menu item number (from 1 to 32767) whose mark character is obtained.

MarkChar contains the "mark" character that is displayed to the left of the item's name. If no mark is displayed by the specified pop-up menu item, or if the specified pop-up menu or item doesn't exist, MarkChar is set to null (char(0)). The following are useful constants for testing menu marks:

```
CONST                              {Menu Item characters              }
              AppleChar   = char($14);   {Apple character                   }
              CheckChar   = char($12);   {Check Mark character              }
              DiamondChar = char($13);   {Diamond character                 }
              DotChar     = char($A5);   {Dot (or bullet) character         }
              NoChar      = char($00);   {No character (remove a character) }
```

PopUpIcon

Set a pop-up menu item's icon.

C

```
pascal void PopUpIcon (short MenuNumber, short ItemNumber,
              short IconSelector);
```

Pascal

```
procedure PopUpIcon (MenuNumber, ItemNumber, IconSelector: INTEGER);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the pop-up menu does not exist, PopUpIcon does nothing.

ItemNumber specifies the pop-up menu item number (from 1 to 32767) that is affected. If the item number does not exist within the pop-up menu specified by MenuNumber, PopUpIcon does nothing.

IconSelector identifies the icon that is used, and should be from 1 to 255. The Menu Manager adds 256 to the number you state to specify a resource ID that is in the range of 257 to 511, so if you specify 28, resource ID 284 is used (28 + 256 = 284). These icon resources are read from your application. If Color QuickDraw is available on the Macintosh running your application, a 'cicn' (color) icon is used. If a 'cicn' is not available (or Color QuickDraw is unavailable), an 'ICON' or 'SICN' is used. Use zero (0) if you don't want an icon displayed.

Unlike the equivalent Macintosh toolbox routine, your menu item will remain unaffected if the specified icon can't be found (i.e., empty space is not reserved in the menu).

Be aware that the Menu Manager drawing a 'cicn' icon in color will do so even if the icon was created using 8-bit colors and the monitor is set to 4-bits. This may produce unsatisfactory results. If possible, use 4-bit colors or colors that translate well into 4-bit colors.

GetPopUpIcon

Get a pop-up menu item's icon number.

C
```
pascal void GetPopUpIcon (short MenuNumber, short ItemNumber,
           short *IconSelector);
```

Pascal
```
procedure GetPopUpIcon (MenuNumber, ItemNumber: INTEGER;
           var IconSelector: INTEGER);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window that contains the desired menu item.

ItemNumber specifies the pop-up menu item number (from 1 to 32767) whose icon number is obtained.

IconSelector contains the item's icon number. The Menu Manager automatically adds 256 to the IconSelector you specify, so an IconSelector of 28 means that resource ID 284 is used (28 + 256 = 284). If an icon is not displayed by the specified pop-up menu item, IconSelector will be equal to zero.

PopUpStyle

Set a pop-up menu item's style.

C
```
pascal void PopUpStyle (short MenuNumber, short ItemNumber, Style theStyle);
```

Pascal
```
procedure PopUpStyle (MenuNumber, ItemNumber: INTEGER; theStyle: Style);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the pop-up menu does not exist, PopUpStyle does nothing.

ItemNumber specifies the pop-up menu item number (from 1 to 32767) that is affected. If the item number does not exist within the pop-up menu specified by MenuNumber, PopUpStyle does nothing.

TheStyle specifies the style(s) in which the pop-up menu item is to be displayed. Special character constants defined by the Font Manager are bold, italic, underline and shadow. C programmers will use the font manager's constants to specify a composite style, such as PopUpStyle(1,1, bold + outline) for bold and outlined, or PopUpStyle(1,1,0) for plain text. Pascal programmers will use the font manager's constants to specify a set, such as PopUpStyle(1,1,[bold,outline]) for bold and outlined, or PopUpStyle(1,1, [ ]) for plain text.

PopUpItemCount

Determine the number of items in a pop-up menu.

C
```
pascal short PopUpItemCount (short MenuNumber);
```

Pascal
```
function PopUpItemCount (MenuNumber): INTEGER;
```

MenuNumber specifies the pop-up menu number (from 1 to 511) you wish to query in the current window.

The routine's value returns the number of items in the specified pop-up menu. If the pop-up menu number does not exist, the routine returns zero.

GetPopUpSelection

Determine the selected item in a pop-up menu.

C
```
pascal short GetPopUpSelection (short MenuNumber);
```

Pascal
```
function GetPopUpSelection (MenuNumber: INTEGER): INTEGER;
```

MenuNumber specifies the pop-up menu number (from 1 to 511) you wish to query in the current window.

The routine's value returns the number of the pop-up menu item that is selected by having a "mark" (check mark or otherwise) beside it. If you have defined your pop-up menu to allow multiple selections, PopUpItemCount returns the number of the *first* selected item. If the pop-up menu number does not exist, the routine returns zero.

MovePopUp

Move a pop-up menu to a new location on the window.

C
```
pascal void MovePopUp (short MenuNumber, short toHoriz, short toVert);
```

Pascal
```
procedure MovePopUp (MenuNumber, toHoriz, toVert: INTEGER);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *MenuNumber* specifies a pop-up menu that does not exist, MovePopUp does nothing. The change is seen immediately providing that the pop-up menu is not hidden. The pop-up menu's width and height are not changed.

ToHoriz is the new horizontal co-ordinate at which the left side of the pop-up menu appears.

ToVert is the new vertical co-ordinate at which the top of the pop-up menu appears.

Also see: SizePopUp and MoveSizePopUp.

OffsetPopUp

Change a pop-up menu's co-ordinates without affecting its image on the window.

C
```
pascal void OffsetPopUp (short MenuNumber,
            short distHoriz, short distVert);
```

Pascal
```
procedure OffsetPopUp (MenuNumber, distHoriz, distVert: INTEGER);
```

When you scroll an area that contains pop-up menus, first use ScrollRect to scroll the pixel image containing the affected objects in the window. OffsetPopUp is used to offset a pop-up menu's co-ordinates without altering its image (since ScrollRect has already done so). At this point, the pop-up menu's co-ordinates match the scrolled image of the pop-up menu. ObscurePopUp or KillPopUp can be used to hide or delete pop-up menus that are scrolled out of view.

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *MenuNumber* specifies a pop-up menu that does not exist, OffsetPopUp does nothing.

DistHoriz and *distVert* specify the horizontal and vertical amount by which the pop-up menu's co-ordinates are offset. Positive numbers are right and down. The pop-up menu's co-ordinates are updated but no change is seen.

SizePopUp

Change a pop-up menu's size.

C
```
pascal void SizePopUp (short MenuNumber, short width);
```

Pascal
```
procedure SizePopUp (MenuNumber, width: INTEGER);
```

SizePopUp changes a pop-up menu's width. The height cannot be changed. The change is seen immediately providing that the pop-up menu is not hidden.

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *MenuNumber* specifies a pop-up menu that does not exist, SizePopUp does nothing.

Width specifies the pop-up menu's new width in pixels. If the parameter is less than 1, SizePopUp does nothing.

Also see: MovePopUp and MoveSizePopUp.

MoveSizePopUp

Change a pop-up menu's co-ordinates.

C
```
pascal void MoveSizePopUp (short MenuNumber,
           short left, short top, short right, short bottom);
```

Pascal
```
procedure MoveSizePopUp (MenuNumber, left, top, right, bottom: INTEGER);
```

MoveSizePopUp changes any of the pop-up menu's four co-ordinates. The change is seen immediately providing that the pop-up menu is not hidden. This routine combines the functions of MovePopUp and SizePopUp.

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *MenuNumber* specifies a pop-up menu that does not exist, MoveSizePopUp does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in local co-ordinates that determines the pop-up menu's size and location in the window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). If these parameters specify an empty rectangle, MoveSizePopUp does nothing. Note that the bottom co-ordinate is ignored since the pop-up menu's height cannot be changed.

Also see: GetPopUpRect.

MoveSizePopUpRect

Change a pop-up menu's co-ordinates.

C
```
pascal void MoveSizePopUpRect (short MenuNumber, const Rect *Bounds);
```

Pascal
```
procedure MoveSizePopUpRect (MenuNumber: INTEGER; Bounds: RECT);
```

MoveSizePopUpRect is identical to the MoveSizePopUp routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

AutoMoveSizePopUp

Specify how a pop-up menu is automatically moved and/or resized as its window's size is changed.

C
```
pascal void AutoMoveSizePopUp (short MenuNumber,
            Boolean left, Boolean top, Boolean right);
```

Pascal
```
procedure AutoMoveSizePopUp (MenuNumber: INTEGER; left, top, right: BOOLEAN);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *MenuNumber* specifies a pop-up menu that does not exist, AutoMoveSizePopUp does nothing.

The *left*, *top* and *right* parameters specify if that side of the pop-up menu is automatically adjusted when the window's size changes. These setting are applied to the pop-up menu and are used the next time the window's size changes:

- *left* Does the pop-up menu's left side track the window's right edge?
- *top* Do the pop-up menu's top and bottom track the window's bottom edge?
- *right* Does the pop-up menu's right side track the window's right edge?

Notice that *top* is used to make both the top and bottom of the menu track the window's bottom. This ensures that the pop-up menu's height does not change.

You can think of each *false* value as locking that side of the pop-up menu to a fixed co-ordinate regardless of the window's size (this is the default). Each *true* value establishes a fixed distance between that side of the pop-up menu and the window's edge. For example, setting only *left* and *right* to *true* makes the pop-up menu move horizontally as the window widens and narrows, but the pop-up menu does not move vertically when the window's height changes.

If you are setting these values identically for a group of objects, use AutoMoveSize to define the settings then add the appropriate *x*AutoMoveSize constant (such as popupAutoMoveSize for pop-up menus) to the objects' spec as they are created. The objects will adopt the settings specified by the AutoMoveSize routine.

☞ **Warning**: Make sure that you resize objects in a way that makes sense. Don't allow a window to shrink down to a size where objects become unusable or disappear altogether.

SetPopUpFontSettings

Set a pop-up menu's font, size and style settings.

C
```
pascal void SetPopUpFontSettings (short MenuNumber,
            short theFont, short theSize, Style theStyle);
```

Pascal
```
procedure SetPopUpFontSettings (MenuNumber: INTEGER;
            theFont: INTEGER; theSize: INTEGER; theStyle: Style);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if the pop-up menu does not exist, SetPopUpFontSettings does nothing. Otherwise, the change is seen immediately.

TheFont specifies the pop-up menu's new font. The default is Chicago, which is represented by the systemFont constant. This font is used to display the pop-up menu's title and all the items in the list.

TheSize specifies the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case. This size is used to display the pop-up menu's title and all the items in the list.

TheStyle specifies the pop-up menu's new style. Special character constants defined by the Font Manager are bold, italic, underline and shadow. C programmers use the Font Manager's constants to specify a composite style, such as SetPopUpFontSettings(1, geneva, 9, bold + outline) for bold and outlined, or SetPopUpFontSettings(1, geneva, 9, 0) for plain text. Pascal programmers use the Font Manager's constants to specify a style set, such as

SetPopUpFontSettings(1, geneva, 9, [bold, outline]) for bold and outlined, or SetPopUpFontSettings(1, geneva, 9, [ ]) for plain text. This style applies only to the pop-up menu's title. Items in the pop-up menu's list are styled individually.

A pop-up menu's font settings are set when a pop-up menu is created, so this routine is not normally used by many applications.

☞ **Warning**: Apple's pop-up menu CDEFs are notorious for misbehaving if you change their font family or font size (either one affects the font's height). If you are using a CDEF for your pop-up menu, make sure you thoroughly test the results of using the SetPopUpFontSettings routine. These issues do not affect Tools Plus's standard or 3D pop-up menus.

GetPopUpFontSettings

Get a pop-up menu's font, size and style settings.

C
```
pascal void GetPopUpFontSettings (short MenuNumber,
            short *theFont, short *theSize, Style *theStyle);
```

Pascal
```
procedure GetPopUpFontSettings (MenuNumber: INTEGER;
            var theFont: INTEGER; var theSize: INTEGER; var theStyle: Style);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) in the current window whose font settings are being retrieved. If the current window doesn't belong to your application, if no windows are open, or if *MenuNumber* specifies a pop-up menu that does not exist, GetPopUpFontSettings returns default values.

TheFont is the pop-up menu's font number. The default is 0 which is represented by the systemFont constant.

TheSize is the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle is the field's font style. The default is plain text, which is represented by 0 in C and [ ] in Pascal.

SetPopUpColors

Set a pop-up menu's colors.

C
```
pascal void SetPopUpColors (short MenuNumber, const RGBColor *Title,
            const RGBColor *Frame, const RGBColor *Body,
            const RGBColor *DfltItemText, const RGBColor *ListBackground);
```

Pascal
```
procedure SetPopUpColors (MenuNumber: INTEGER;
            Title, Frame, Body, DfltItemText, ListBackground: RGBColor);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) in the current window whose colors are being set. If the current window doesn't belong to your application, or if no windows are open, SetPopUpColors does nothing. Also, if *MenuNumber* specifies a pop-up menu that does not exist, SetPopUpColors does nothing. The change is seen immediately, regardless if the pop-up menu was originally created with the popupColorPopUp option or not.

Title is the color of the pop-up menu's title, which may be external to the control or a fixed title within the control.

Frame is the pop-up menu's frame color.

Body is the pop-up menu's body color. This is the color that is used to fill the control's body.

DfltItemText is the default color used to display items in the pop-up menu's list.

ListBackground is the background color used for the pop-up menu's list.

Note: Some pop-up menu CDEFs may not respond to all the settings provided by this routine. This is the case with System 7's CDEF 63 and may be the case with third party CDEFs as well.

Also see: PopUpColors and GetPopUpColors.

GetPopUpColors

Get a pop-up menu's colors.

C
```
pascal void GetPopUpColors (short MenuNumber, RGBColor *Title,
            RGBColor *Frame, RGBColor *Body, RGBColor *DfltItemText,
            RGBColor *ListBackground);
```

Pascal
```
procedure GetPopUpColors (MenuNumber: INTEGER; var Title: RGBColor;
            var Frame: RGBColor; var Body: RGBColor;
            var DfltItemText: RGBColor; var ListBackground: RGBColor);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) in the current window whose colors are being retrieved. If the current window doesn't belong to your application, or if no windows are open, or if *MenuNumber* specifies a pop-up menu that does not exist, GetPopUpColors returns default color values.

Title is the color of the pop-up menu's title, which may be external to the control or a fixed title within the control.

Frame is the pop-up menu's frame color.

Body is the pop-up menu's body color. This is the color that is used to fill the control's body.

DfltItemText is the default color used to display items in the pop-up menu's list.

ListBackground is the background color used for the pop-up menu's list.

Also see: PopUpColors and SetPopUpColors.

SetPopUpItemColors

Set a pop-up menu item's colors.

C
```
pascal void SetPopUpItemColors (short MenuNumber, short ItemNumber,
            const RGBColor *MarkColor, const RGBColor *ItemColor);
```

Pascal
```
procedure SetPopUpItemColors (MenuNumber, ItemNumber: INTEGER;
            MarkColor, ItemColor: RGBColor);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, SetPopUpItemColors does nothing.

ItemNumber specifies the pop-up menu's item number (from 1 to 32767) that is affected. If *ItemNumber* does not exist, SetPopUpItemColors does nothing. If the pop-up menu displays the currently selected item and you are changing the colors for that item, the change is seen immediately.

MarkColor is the color used to draw the specified menu item's mark character.

ItemColor is the color used to draw the specified menu item's text, command key, and submenu character.

Also see: PopUpColors and GetPopUpItemColors.

GetPopUpItemColors

Get a pop-up menu item's colors.

C
```
pascal void GetPopUpItemColors (short MenuNumber, short ItemNumber,
            RGBColor *MarkColor, RGBColor *ItemColor);
```

Pascal
```
procedure GetPopUpItemColors (MenuNumber, ItemNumber: INTEGER;
            var MarkColor: RGBColor; var ItemColor: RGBColor);
```

MenuNumber specifies the pop-up menu number (from 1 to 511) in the current window whose colors are being retrieved. If the current window doesn't belong to your application, or if the specified pop-up menu doesn't exist, GetPopUpItemColors returns default color values.

ItemNumber specifies the pop-up menu's item number (from 1 to 32767) whose colors are being retrieved. If *ItemNumber* does not exist, GetPopUpItemColors returns default color values.

MarkColor is the color used to draw the specified menu item's mark character.

ItemColor is the color used to draw the specified menu item's text, command key, and submenu character.

Also see: PopUpColors and SetPopUpItemColors.

GetPopUpHandle

Get a handle to a pop-up menu's control or menu record.

C
```
pascal Handle GetPopUpHandle (short MenuNumber);
```

Pascal
```
function GetPopUpHandle (MenuNumber: INTEGER): Handle;
```

This routine returns a standard ControlHandle to a pop-up menu control that was created by a Tools Plus routine if the popupSystemBody option was used when creating the pop-up menu. If the popupSystemBody option was not used, a handle to a menu record is returned. You should never need to use this routine. It is provided for advanced programmers who may have specialized needs. Always use Tools Plus routines to create and manipulate pop-up menus.

MenuNumber specifies the pop-up menu number (from 1 to 511) in the current window whose handle is being retrieved. If the current window doesn't belong to your application, or if no windows are open, or if *MenuNumber* specifies a pop-up menu that does not exist, GetPopUpHandle returns nil.

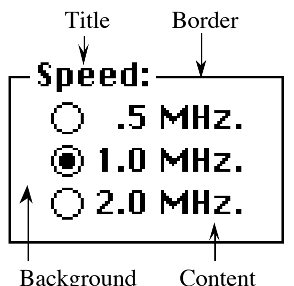
☞ **Warning**: If you need to lock the handle or change its attributes, do so temporarily then restore the original settings before using any Tools Plus routines. If you alter this handle or any data that is made accessible by this handle, you do so at your own risk. The only exception is the control's reference constant (contrlRfCon field) which can safely be set using the toolbox's SetControlReference routine, and retrieved using the toolbox GetControlReference routine.

12 Panels

Panels and group boxes are user interface elements designed to give the user a visual cue that multiple objects are related in some way. They can also be used purely as a cosmetic enhancement to give a window a more contemporary 3D look as seen in many of today's applications. Group boxes are just a panel with a title, so within this manual the terms "panel" and "group box" can be used interchangeably.

If your application is dependent upon the Appearance Manager, you should use the Appearance Manager's Group Box control through your application in place of Tools Plus's panels. The Group Box control is not as versatile as a Tools Plus panel, but it gives your application a user interface that is consistent with the Appearance Manager's themes. See the Buttons chapter for information on implementing the Appearance Manager's Group Box control in your application.

Tools Plus's panels do more than just make your user interface look better. Panels map perfectly between windows of varying depths on color and gray scale monitors as well as monochrome monitors (1-bit black and white). This means that you can create a panel just like any other Tools Plus user interface element, and it takes care of itself and looks its best at all times. Panels also enhance radio buttons and picture buttons by making them behave like a related group. When you place buttons or picture buttons inside a panel you can optionally deselect the other buttons in the group when a button is selected.



A panel is comprised of several parts all of which can be tailored to suit your application's needs. Any of the parts can be omitted to create a desired affect.

The *border* defines the panel's perimeter. An optional shadow extends inwards from these co-ordinates to make the panel appear to be either inset into the window or raised from the window. A channel option produces a 1-pixel wide groove that is cut into the window or elevated from the window. A variety of styles are available.

If a *title* is included in the panel, the panel takes on the appearance of a group box. The title can be set near the left or right edge of the border, or it can be centered. Various fonts, font sizes and styles can be used for the title. The text can also be inset or raised using soft or heavy shadows. You can use the GetPanelFontSettings and SetPanelFontSettings routines to get and set the panel's font, size and style settings.

The panel's *background* is always erased before drawing the panel or any of its parts. The background includes the region occupied by the title and it can optionally include the interior of the panel as well.

A panel's content is specified by your application. Typically this is either a set of radio buttons created with the NewButton routine, or a set of picture buttons created with the NewPictButton routine. When you create a panel you can set an option that deselects all other buttons inside the panel's when one of them is selected.

Panels are created on the current window by the NewPanel routine. Each panel is referenced by a unique panel number that can be from 1 to 511. This number is specified when the panel is created, and refers to the specific panel until that panel is deleted. Note that the panel number is related to its associated window. This means that two different windows can each have a panel numbered "1" without interfering with each other.

Panels can be moved to a new location with MovePanel and have their width and/or height changed with SizePanel. MoveSizePanel combines both tasks by letting you specify new co-ordinates for the panel.

When a panel is no longer required, it is deleted by the DeletePanel routine, which releases the memory used by that panel. This is done automatically if a window is closed. Panels can be hidden or displayed with the PanelDisplay routine. Hiding, displaying or moving panels does not affect objects you place inside the panel.

Color Tables

Panels provide a lot of versatility in the way that colors are used. Various options are offered to make efficient use of memory and to ease programming. By default each panel points to a global *standard color table* for panels that specifies the following colors:

Text	Title's color
Background	Color filled behind title and optionally inside the panel
Border	Color of panel's border
Hilite	Color used to draw highlights on the title and panels
Shadow	Used for drawing the panel's shadows
Text Shadow	Used for drawing shadows for the title
Heavy Text Shadow	Used for drawing heavy shadows for the title

The standard color table is initialized to a set of light grays that are consistent with Macintosh user interface guidelines. Your application can get and set these colors using GetStandardPanelColors and SetStandardPanelColors. If you want most or all panels to share a common set of colors that are different from the standard color table, change these settings during your application's initialization routine.

Although the standard color table includes a background color, panels assume that their background is the same as their window's backdrop color. This lets you use the standard color table on a variety of window colors without having to change the color table or use custom colors. A panel can optionally use the standard color table's background color instead of the window's backdrop for the panel background. Using the standard color table is the most memory efficient option since panels only refer to the global color table and do not make their own copy.

A second option is using the *custom color table* for panels. The custom color table is similar to the standard color table in that your application can get and set the colors using GetCustomPanelColors and SetCustomPanelColors. When you create a panel and instruct it to use the custom color table, the panel makes a copy of the custom color table for its own use, thereby letting you set custom colors for several panels at a time then change the custom color table without affecting any panels. Although there is no performance penalty, a panel that uses a custom color table consumes an additional 42 bytes of memory to store a copy of the custom colors. The custom color table is initialized to a set of darker grays. They produce an attractive interface but they do not follow the Macintosh tradition of "light, unobstructive colors."

A final set of options let you use the window's foreground color for the panel's text or border, and/or use the window's background color for the panel's background. These options cause the panel to create their own copy of the standard color table or custom color table, then override the specified entries in its own color table.

NewPanel

Create a new panel.

C
```
pascal void NewPanel (short Panel, short left, short top, short right,
            short bottom, const Str255 Title, long Spec, short ShadowWidth);
```

Pascal
```
procedure NewPanel (Panel, left, top, right, bottom: INTEGER; Title: STRING;
            Spec: LONGINT; ShadowWidth: INTEGER);
```

Panel specifies the panel number (from 1 to 511) that is created in the current window. Once a panel is created, it is referenced by this panel number. If a panel has been previously created in the current window using the same number, it is replaced with a new panel as specified by the parameters in the NewPanel routine. If the current window doesn't belong to your application, or if no windows are open, NewPanel does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in local co-ordinates that determines the panel's size and location in the window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). If you want buttons or picture buttons to be automatically deselected, their co-ordinates must lie within this rectangle. You can create a line by specifying a rectangle that is 1 pixel wide or high.

The *Title* parameter is the panel's title. The title is drawn overlaying the rectangle's top line and therefore extends beyond the boundaries specified by the panel's rectangle. You can have a panel without a title by specifying an empty string.

Spec specifies a panel's appearance and behavior. It is a combination of various Tools Plus options detailed later in this section.

ShadowWidth specifies how wide the panel's shadow is. The shadow usually extends in from the panel's rectangle. Use zero (0) when creating a group box. You can then specify you want a raised or inset 1-pixel wide channel, in which case highlights and shadows are drawn inside and outside the panel's rectangle.

Appearance and Behavior Specification

Spec specifies a panel's appearance and behavior. The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result. The constants defining the available options are as follows:

Optionally choose any of the following options…

`panFillBack` — Fill the panel's content area with its background color. By default, only the title's background is filled and the panel's interior is hollow allowing objects behind it to show through.

`panOutlined` — Always draw an outline around the panel. This is done automatically for group boxes when you specify a shadow width of zero (0). This outline puts more emphasis on a raised or inset panel.
The outline is drawn using the border color on monitors of all depths. Do not use this option if you want a border displayed only on a monitor depth of 4 bits or less.

without outline (default)

with outline

`panBlackBorder` — This option is identical to the panOutlined option, except that the border is black instead of using the border color which is usually gray.

`panOutline4bit` — Draw an outline around the panel only when it is displayed on a monitor set to 4-bits or less. This is done automatically for group boxes when you specify a shadow width of zero (0). This outline puts more emphasis on a raised or inset panel.
The outline is drawn using the border color. Do not use this option if you want a border displayed only on a monitor depth of 2 bits or less.

`panBWGrayBorder` — Draw an outline around the panel using a gray pattern to produce a dotted outline. This occurs only when the panel is drawn on a black and white (1-bit) monitor. By default, group boxes have a black outline on monochrome monitors, and ordinary panels have no outline so they disappear because both the shadow and highlight are mapped to white. Use this option if you do not want your panels to disappear on monochrome monitors.

`panUseWFont` — Display the panel's title using the window's current font, size and style settings (as set by the TextFont, TextSize, and TextFace routines). The panel stores this information for future reference. By default, all panels are drawn using the system font (Chicago, 12 pt).

`panCustomColors` — Make a copy of the custom color table and use those colors when drawing the panel. By default, panels are drawn using a shared standard color table. Making a copy of the custom color table consumes an additional 42 bytes but allows the panel to have its own color scheme.

`panColorText`	Use the window's foreground color for the panel's title. By default, the panel's title is drawn using the text color in the standard color table or in the custom color table if you've elected to use the custom table. When you use this option, a copy of the specified color table is made for this panel and its text color entry is replaced with the window's foreground color. This option is best used for overriding the text color used in the standard or custom color table. If you find yourself using this option frequently, consider changing the standard or custom color table.
`panColorBorder`	Use the window's foreground color for the panel's border. By default, the panel's border is drawn using the border color in the standard color table or in the custom color table if you've elected to use the custom table. When you use this option, a copy of the specified color table is made for this panel and its border color entry is replaced with the window's foreground color. This option is best used for overriding the border color used in the standard or custom color table. If you find yourself using this option frequently, consider changing the standard or custom color table.
`panColorBack`	Use the window's background color for the panel's background. By default, the panel's background is drawn using the window's backdrop color. When you use this option, a copy of the specified color table is made for this panel and its background color entry is replaced with the window's background color. This option is best used for overriding the background color used in the standard or custom color table. If you find yourself using this option frequently, consider changing the standard or custom color table. If you want to use the color table's background color instead of the window's backdrop color for the panel's background, use the panNoBackdrop option.
`panNoBackdrop`	Use the color table's background color for the panel's background. By default, the panel's background is drawn using the window's backdrop color.
`panAutoDeselect`	Automatically deselect other buttons and picture buttons inside the panel when a button is selected. This option is ideal for radio button groups or groups of picture buttons that behave like radio buttons. When your application gets a doButton or doPictButton event, all it has to do is select the clicked button and all other buttons inside the panel will be deselected. It is important that the buttons and/or picture buttons are completely enclosed by the panel's rectangle otherwise they are not considered to be inside the panel. Hidden buttons and picture buttons are not affected.
`panAutoMoveSize`	Automatically move and/or resize the panel when the window's size changes. The AutoMoveSize routine lets you specify which sides are altered. You can use the AutoMoveSizePanel routine as an alternative to setting this option.
`panHidden`	Create a hidden panel. This kind of panel is accessible to your application but not to the user.

Optionally choose only one of the following title-position options…

`panLeftTitle`	Position the panel's title near the left side. This is the default and it does not need to be explicitly included.
`panRightTitle`	Position the panel's title near the right side. By default, the panel's title is positioned near the left side.
`panCenterTitle`	Position the panel's title in the center. By default, the panel's title is positioned near the left side.

Optionally choose only one of the following title 3D-style options…

`panPlainTitle`	Display the panel's title without using 3D enhancements. This is the default and it does not need to be explicitly included.
`panRaiseTitle`	Display the panel's title as being raised from the window. Soft shadows are used. This option works very well with light colored backgrounds, and is a subtle effect when used on a darker background.
`panRaiseTitleDark`	Display the panel's title as being raised from the window. Heavy shadows are used. This option works very well with darker colored backgrounds but may be too dramatic on lighter colored backgrounds.
`panInsetTitle`	Display the panel's title as being inset into the window. Soft shadows are used. This option works very well with light colored backgrounds, and is a subtle effect when used on a darker background.
`panInsetTitleDark`	Display the panel's title as being inset into the window. Heavy shadows are used. This option works very well with darker colored backgrounds but may be too dramatic on lighter colored backgrounds.

Optionally choose only one of the following shadow style options…

`panNoShadow`	Draw the panel without any shadows. This option is used most often for plain group boxes without a 3D effect. This is the default and it does not need to be explicitly included.
`panRaiseShadow`	Draw the panel as being raised from the window. If you specify this option in conjunction with a shadow width of zero (0), a 1-pixel wide raised channel is created, a suitable look for a group box.
`panInsetShadow`	Draw the panel as being inset into the window. If you specify this option in conjunction with a shadow width of zero (0), a 1-pixel wide inset channel is created, the default look for a 3D group box.

Optionally choose only one of the following round-corner options…

`panRoundCorner1`
`panRoundCorner2`
`panRoundCorner3`
⇓
`panRoundCorner29`
`panRoundCorner30`
`panRoundCorner31`
	Draw the panel as a round-corner rectangle using an oval size that corresponds to the specified value. By default, panels have square corners. The round-corner rectangle is created with the toolbox's FrameRoundRect routine using an oval height and width of the specified value. If you are using a gray pattern outline on monochrome monitors (panBWGrayBorder option), the corners may not look very tidy when displayed on a 1-bit monitor.

Choose only one of the following popular combinations as a base for a spec…

`panGroupBox`	Standard group box without any 3D effects. This constant is simply a combination of the following options: panLeftTitle + panBlackBorder + panNoShadow + panAutoDeselect.
`pan3DGroupBox`	Standard 3D group box. This constant is simply a combination of the following options: panLeftTitle + panInsetShadow + panRaiseTitle + panAutoDeselect.

Also see: NewPanelRect and NewDialogPanel.

```
CONST                                           {Panel appearance/behavior specifications:     }
               panFillBack       = $00000001;   {Fill panel with background color              }
               panOutlined       = $00000002;   {Always draw outline around panel              }
               panOutline4bit    = $00000004;   {Draw outline on 4-bit monitors                }
               panBlackBorder    = $00000008;   {Draw black border instead of gray             }
               panBWGrayBorder   = $00000010;   {Draw gray pattern border on B&W monitor       }
               panUseWFont       = $00000020;   {Draw title using window's font                }
               panCustomColors   = $00000040;   {Use custom colors instead of standard ones    }
```

```
panColorText      = $00000080;   {Use foreground color for text              }
panColorBorder    = $00000100;   {Use foreground color for border            }
panColorBack      = $00000200;   {Use background color for background        }
panNoBackdrop     = $00000400;   {Use standard or custom background color    }
                                 {  instead of window's backdrop color       }
panAutoDeselect   = $00000800;   {Auto-deselect buttons in this group        }
                                 {Title alignment:                           }
panLeftTitle      = $00001000;   {  Left (default)                           }
panRightTitle     = $00002000;   {  Right                                    }
panCenterTitle    = $00003000;   {  Center                                   }
                                 {Title's 3D styling:                        }
panPlainTitle     = $00000000;   {  Plain title (no 3D effect)               }
panRaiseTitle     = $00010000;   {  Raised with light shadow                 }
panRaiseTitleDark = $00020000;   {  Raised with heavy shadow                 }
panInsetTitle     = $00030000;   {  Inset with light shadow                  }
panInsetTitleDark = $00040000;   {  Inset with heavy shadow                  }
                                 {Shadow styling:                            }
panNoShadow       = $00000000;   {  No shadow (default)                      }
panRaiseShadow    = $00080000;   {  Panel raises from window                 }
panInsetShadow    = $00100000;   {  Panel sinks into window                  }
                                 {Round-corner Width:                        }
panRoundCorner1 = $01000000;     {  RoundRect oval size of 1 pixel           }
panRoundCorner2 = $02000000;     {  RoundRect oval size of 2 pixels          }
panRoundCorner3 = $03000000;     {  RoundRect oval size of 3 pixels          }
panRoundCorner4 = $04000000;     {  RoundRect oval size of 4 pixels          }
panRoundCorner5 = $05000000;     {  RoundRect oval size of 5 pixels          }
panRoundCorner6 = $06000000;     {  RoundRect oval size of 6 pixels          }
panRoundCorner7 = $07000000;     {  RoundRect oval size of 7 pixels          }
panRoundCorner8 = $08000000;     {  RoundRect oval size of 8 pixels          }
panRoundCorner9 = $09000000;     {  RoundRect oval size of 9 pixels          }
panRoundCorner10 = $0A000000;    {  RoundRect oval size of 10 pixels         }
panRoundCorner11 = $0B000000;    {  RoundRect oval size of 11 pixels         }
panRoundCorner12 = $0C000000;    {  RoundRect oval size of 12 pixels         }
panRoundCorner13 = $0D000000;    {  RoundRect oval size of 13 pixels         }
panRoundCorner14 = $0E000000;    {  RoundRect oval size of 14 pixels         }
panRoundCorner15 = $0F000000;    {  RoundRect oval size of 15 pixels         }
panRoundCorner16 = $10000000;    {  RoundRect oval size of 16 pixels         }
panRoundCorner17 = $11000000;    {  RoundRect oval size of 17 pixels         }
panRoundCorner18 = $12000000;    {  RoundRect oval size of 18 pixels         }
panRoundCorner19 = $13000000;    {  RoundRect oval size of 19 pixels         }
panRoundCorner20 = $14000000;    {  RoundRect oval size of 20 pixels         }
panRoundCorner21 = $15000000;    {  RoundRect oval size of 21 pixels         }
panRoundCorner22 = $16000000;    {  RoundRect oval size of 22 pixels         }
panRoundCorner23 = $17000000;    {  RoundRect oval size of 23 pixels         }
panRoundCorner24 = $18000000;    {  RoundRect oval size of 24 pixels         }
panRoundCorner25 = $19000000;    {  RoundRect oval size of 25 pixels         }
panRoundCorner26 = $1A000000;    {  RoundRect oval size of 26 pixels         }
panRoundCorner27 = $1B000000;    {  RoundRect oval size of 27 pixels         }
panRoundCorner28 = $1C000000;    {  RoundRect oval size of 28 pixels         }
panRoundCorner29 = $1D000000;    {  RoundRect oval size of 29 pixels         }
panRoundCorner30 = $1E000000;    {  RoundRect oval size of 30 pixels         }
panRoundCorner31 = $1F000000;    {  RoundRect oval size of 31 pixels         }
panAutoMoveSize  = $40000000;    {Auto-move/size as window's size changes    }
panHidden        = $80000000;    {Panel is hidden                            }
                                 {Popular combinations:                      }
                                 {  Standard group box                       }
panGroupBox      = panLeftTitle + panBlackBorder + panNoShadow + panAutoDeselect;
                                 {  3D group box                             }
pan3DGroupBox    = panLeftTitle + panInsetShadow + panRaiseTitle + panAutoDeselect;
```

Programming Tips:

1. You can suppress the panel's border by specifying a rectangle whose bottom and top have the same vertical co-ordinate (an empty rectangle). This lets you use a panel to draw 3D text that can be used for titles. Tools Plus automatically refreshes these titles.
2. You can use panels to create vertical and/or horizontal lines that are automatically refreshed. For a vertical line, make the panel's right co-ordinate equal the left plus one. For a horizontal line, make the panel's bottom co-ordinate equal the top plus one.

NewPanelRect

Create a new panel.

C
```
pascal void NewPanelRect (short Panel, const Rect *Bounds,
            const Str255 Title, long Spec, short ShadowWidth);
```

Pascal
```
procedure NewPanelRect (Panel: INTEGER; Bounds: RECT;
            Title: STRING; Spec: LONGINT; ShadowWidth: INTEGER);
```

NewPanelRect is identical to the NewPanel routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

NewDialogPanel

Create a new panel in a dialog using a dialog item's co-ordinates.

C
```
pascal void NewDialogPanel (short Panel, const Str255 Title, long Spec,
            short ShadowWidth);
```

Pascal
```
procedure NewDialogPanel (Panel: INTEGER; Title: STRING; Spec: LONGINT;
            ShadowWidth: INTEGER);
```

NewDialogPanel is identical to the NewPanel routine, except that the panel is created in a dialog (a window opened with the LoadDialog routine, or one that had a dialog list attached with the LoadDialogList routine). The panel's co-ordinates are obtained from the dialog item whose number matches the panel number.

GetFreePanelNum

Get the first unused panel number.

C
```
pascal short GetFreePanelNum (void);
```

Pascal
```
function GetFreePanelNum: INTEGER;
```

Some developers may prefer to write code that more closely resembles a traditional Macintosh application, in that creating an object returns a reference to it such as a handle or pointer. Instead of having to assign your own panel number, GetFreePanelNum returns the first unused (free) panel number. Using this routine, you can assign an unused panel number to a variable, then use that variable throughout your application without concern for the true panel number.

GetFreePanelNum returns the first free panel number on the current window. If the current window doesn't belong to your application, if no windows are open, or if the maximum number of panels has already been created on the current window (no new ones can be created), GetFreePanelNum returns a value of zero (0).

SetStandardPanelColors

Set the standard color table's colors.

C
```
pascal void SetStandardPanelColors (const RGBColor *Text,
            const RGBColor *Background, const RGBColor *Border,
            const RGBColor *Hilite, const RGBColor *Shadow,
            const RGBColor *TextShadow, const RGBColor *TextHeavyShadow);
```

Pascal
```
procedure SetStandardPanelColors (Text, Background, Border, Hilite, Shadow,
            TextShadow, TextHeavyShadow: RGBColor);
```

The standard color table for panels is used by all panels as a default set of colors. It is initialized to a set of light grays that are consistent with Macintosh user interface guidelines. If you want to change the standard color table for panels, do so early in your application, typically during the application initialization routine. Changing these colors does not automatically force existing panels that use these colors to be redrawn.

Text is the color of the panel's title.

Background is the color of the panel's background. By default, the panel uses the window's backdrop color but it may be instructed to use this background color.

Border is the color of the panel's border.

Hilite is the color that is used to draw highlights on both the title and the panel. This color is usually white.

Shadow is the color used to draw the panel's shadows

TextShadow is the color used to draw soft shadows on the title.

TextHeavyShadow is the color used to draw heavy shadows on the title.

Also see: GetStandardPanelColors, SetCustomPanelColors and GetCustomPanelColors.

GetStandardPanelColors

Get the standard color table's colors.

C
```
pascal void GetStandardPanelColors (RGBColor *Text, RGBColor *Background,
            RGBColor *Border, RGBColor *Hilite, RGBColor *Shadow,
            RGBColor *TextShadow, RGBColor *TextHeavyShadow);
```

Pascal
```
procedure GetStandardPanelColors (var Text: RGBColor;
            var Background: RGBColor; var Border: RGBColor;
            var Hilite: RGBColor; var Shadow: RGBColor;
            var TextShadow: RGBColor; var TextHeavyShadow: RGBColor);
```

This routine gets the colors from the standard color table for panels. These colors are used by all panels as a default set of colors. If you want to change a few of the colors in the table, use GetStandardPanelColors to obtain all the colors in the table, then use SetStandardPanelColors to update the table with new colors, some of which could be the original colors obtained by the "get" operation.

Text is the color of the panel's title.

Background is the color of the panel's background. By default, the panel uses the window's backdrop color but it may be instructed to use this background color.

Border is the color of the panel's border.

Hilite is the color that is used to draw highlights on both the title and the panel. This color is usually white.

Shadow is the color used to draw the panel's shadows

TextShadow is the color used to draw soft shadows on the title.

TextHeavyShadow is the color used to draw heavy shadows on the title.

Also see: SetStandardPanelColors, GetCustomPanelColors and GetCustomPanelColors.

SetCustomPanelColors

Set the custom color table's colors.

C
```
pascal void SetCustomPanelColors (const RGBColor *Text,
            const RGBColor *Background, const RGBColor *Border,
            const RGBColor *Hilite, const RGBColor *Shadow,
            const RGBColor *TextShadow, const RGBColor *TextHeavyShadow);
```

Pascal
```
procedure SetCustomPanelColors (Text, Background, Border, Hilite, Shadow,
            TextShadow, TextHeavyShadow: RGBColor);
```

The custom color table for panels is optionally used by panels that require a custom color table instead of using the default standard color table. A copy of this color table is made for each panel that uses it. The custom color table is initialized to a set of darker grays. They produce an attractive interface but they do not follow the Macintosh tradition of "light, unobstructive colors." If you have a standard color theme for panels, use the standard color table to define those colors. If you want to assign a different set of colors to a number of panels, use this routine to set the custom panel color and they will be adopted by new panels as they are created.

Text is the color of the panel's title.

Background is the color of the panel's background. By default, the panel uses the window's backdrop color but it may be instructed to use this background color.

Border is the color of the panel's border.

Hilite is the color that is used to draw highlights on both the title and the panel. This color is usually white.

Shadow is the color used to draw the panel's shadows

TextShadow is the color used to draw soft shadows on the title.

TextHeavyShadow is the color used to draw heavy shadows on the title.

Also see: SetStandardPanelColors, GetStandardPanelColors and GetCustomPanelColors.

GetCustomPanelColors

Get the custom color table's colors.

C
```
pascal void GetCustomPanelColors (RGBColor *Text, RGBColor *Background,
            RGBColor *Border, RGBColor *Hilite, RGBColor *Shadow,
            RGBColor *TextShadow, RGBColor *TextHeavyShadow);
```

Pascal
```
procedure GetCustomPanelColors (var Text: RGBColor;
            var Background: RGBColor; var Border: RGBColor;
            var Hilite: RGBColor; var Shadow: RGBColor;
            var TextShadow: RGBColor; var TextHeavyShadow: RGBColor);
```

This routine gets the colors from the custom color table for panels. These colors are optionally used by panels for a customized set of colors. If you want to change a few of the colors in the table, use GetCustomPanelColors to obtain all the colors in the table, then use SetCustomPanelColors to update the table with new colors, some of which could be the original colors obtained by the "get" operation.

Text is the color of the panel's title.

Background is the color of the panel's background. By default, the panel uses the window's backdrop color but it may be instructed to use this background color.

Border is the color of the panel's border.

Hilite is the color that is used to draw highlights on both the title and the panel. This color is usually white.

Shadow is the color used to draw the panel's shadows

TextShadow is the color used to draw soft shadows on the title.

TextHeavyShadow is the color used to draw heavy shadows on the title.

Also see: SetStandardPanelColors, GetStandardPanelColors and SetCustomPanelColors.

DeletePanel

Delete a panel.

C
```
pascal void DeletePanel (short Panel);
```

Pascal
```
procedure DeletePanel (Panel: INTEGER);
```

Panel specifies the panel number (from 1 to 511) that is deleted from the current window. If the current window doesn't belong to your application, or if no windows are open, or if the panel does not exist in the current window, DeletePanel does nothing. Use KillPanel if you want to delete the panel without removing its image from the window.

KillPanel

Delete a panel without affecting its image on the window.

C
```
pascal void KillPanel (short Panel);
```

Pascal
```
procedure KillPanel (Panel: INTEGER);
```

KillPanel is identical to DeletePanel except that it does not remove the panel's image from the window. This routine is useful for scrolling panels in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetPanel repositions the panel's co-ordinates without affecting its image (since ScrollRect has already moved it). KillPanel then deletes the panels that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

PanelDisplay

Hide or show a panel.

C
```
pascal void PanelDisplay (short Panel, Boolean Show);
```

Pascal
```
procedure PanelDisplay (Panel: INTEGER; Show: BOOLEAN);
```

PanelDisplay hides or shows a panel on the current window. The result is seen immediately.

Panel specifies the panel number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the panel does not exist in the current window, PanelDisplay does nothing.

Show indicates if the panel is being hidden or displayed. The two constants that can be used for this flag are *on* and *off*.

PanelIsVisible

Determine if a panel is visible.

C
```
pascal Boolean PanelIsVisible (short Panel);
```

Pascal
```
function PanelIsVisible (Panel: INTEGER): BOOLEAN;
```

PanelIsVisible reports if a panel is visible on the current window, or if it is hidden.

Panel specifies the panel number (from 1 to 511) that is queried in the current window.

This routine's value returns *true* if the panel is visible, and *false* if the panel is hidden. If the current window doesn't belong to your application, or if no windows are open, or if the panel does not exist in the current window, PanelIsVisible returns *false*.

ObscurePanel

Hide a panel without removing its image from the window.

C
```
pascal void ObscurePanel (short Panel);
```

Pascal
```
procedure ObscurePanel (Panel: INTEGER);
```

ObscurePanel hides a panel on the current window without removing its image from the window. This routine is useful for scrolling panels in an area within a window (i.e., not the entire window). ScrollRect is used to scroll the images in the affected area. OffsetPanel repositions the panel's co-ordinates without affecting its image (since ScrollRect has already moved it). ObscurePanel then hides the panels that are scrolled out of view without affecting their image (ScrollRect has already scrolled them out of view).

Panel specifies the panel number (from 1 to 511) that is hidden in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the panel does not exist in the current window, ObscurePanel does nothing.

GetPanelRect

Get a panel's co-ordinates.

C
```
pascal void GetPanelRect (short Panel, Rect *Bounds);
```

Pascal
```
procedure GetPanelRect (Panel: INTEGER; var Bounds: RECT);
```

Panel specifies the panel number (from 1 to 511) that is queried in the current window.

Bounds returns the panel's bounding rectangle specified in the window's local co-ordinates. These co-ordinates match those used to create the panel. If the current window doesn't belong to your application, or if no windows are open, or if the panel does not exist in the current window, *Bounds* returns with all co-ordinates set to zero (0).

MovePanel

Move a panel to a new location on the window.

C
```
pascal void MovePanel (short Panel, short toHoriz, short toVert);
```

Pascal
```
procedure MovePanel (Panel, toHoriz, toVert: INTEGER);
```

Panel specifies the panel number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Panel* specifies a panel that does not exist, MovePanel does nothing. The change is seen immediately providing that the panel is not hidden. The panel's width and height are not changed.

ToHoriz is the new horizontal co-ordinate at which the left side of the panel appears.

ToVert is the new vertical co-ordinate at which the top of the panel appears.

Also see: SizePanel and MoveSizePanel.

OffsetPanel

Change a panel's co-ordinates without affecting its image on the window.

C
```
pascal void OffsetPanel (short Panel, short distHoriz, short distVert);
```

Pascal
```
procedure OffsetPanel (Panel, distHoriz, distVert: INTEGER);
```

When you scroll an area that contains panels, first use ScrollRect to scroll the pixel image containing the affected objects in the window. OffsetPanel is used to offset a panel's co-ordinates without altering its image (since ScrollRect has already done so). At this point, the panel's co-ordinates match the scrolled image of the panel. ObscurePanel or KillPanel can be used to hide or delete panels that are scrolled out of view.

Panel specifies the panel number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Panel* specifies a panel that does not exist, OffsetPanel does nothing.

DistHoriz and *distVert* specify the horizontal and vertical amount by which the panel's co-ordinates are offset. Positive numbers are right and down. The panel's co-ordinates are updated but no change is seen.

SizePanel

Change a panel's size.

C
```
pascal void SizePanel (short Panel, short width, short height);
```

Pascal
```
procedure SizePanel (Panel, width, height: INTEGER);
```

SizePanel changes a panel's width and/or height without altering the panel's top or left co-ordinate. The change is seen immediately providing that the panel is not hidden.

Panel specifies the panel number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Panel* specifies a panel that does not exist, SizePanel does nothing.

Width and *height* specify the panel's new width and height in pixels. If either parameter is less than 0, SizeButton does nothing.

Also see: MovePanel and MoveSizePanel.

MoveSizePanel

Change a panel's co-ordinates.

C
```
pascal void MoveSizePanel (short Panel,
            short left, short top, short right, short bottom);
```

Pascal
```
procedure MoveSizePanel (Panel, left, top, right, bottom: INTEGER);
```

MoveSizePanel changes any of the panel's four co-ordinates. The change is seen immediately providing that the panel is not hidden. This routine combines the functions of MovePanel and SizePanel.

Panel specifies the panel number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Panel* specifies a panel that does not exist, MoveSizePanel does nothing.

Left, *top*, *right*, and *bottom* define a rectangle in local co-ordinates that determines the panel's size and location in the window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). If these parameters specify a width or height that is less than zero (0), MoveSizePanel does nothing.

Also see: GetPanelRect.

MoveSizePanelRect

Change a panel's co-ordinates.

C
```
pascal void MoveSizePanelRect (short Panel, const Rect *Bounds);
```

Pascal
```
procedure MoveSizePanelRect (Panel: INTEGER; Bounds: RECT);
```

MoveSizePanelRect is identical to the MoveSizePanel routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

AutoMoveSizePanel

Specify how a panel is automatically moved and/or resized as its window's size is changed.

C
```
pascal void AutoMoveSizePanel (short Panel,
            Boolean left, Boolean top, Boolean right, Boolean bottom);
```

Pascal
```
procedure AutoMoveSizePanel (Panel: INTEGER;
            left, top, right, bottom: BOOLEAN);
```

Panel specifies the panel number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if *Panel* specifies a panel that does not exist, AutoMoveSizePanel does nothing.

The *left*, *top*, *right* and *bottom* parameters specify if that side of the panel is automatically adjusted when the window's size changes. These setting are applied to the panel and are used the next time the window's size changes:

left	Does the panel's left side track the window's right edge?
top	Does the panel's top track the window's bottom edge?
right	Does the panel's right side track the window's right edge?
bottom	Does the panel's bottom track the window's bottom edge?

You can think of each *false* value as locking that side of the panel to a fixed co-ordinate regardless of the window's size (this is the default). Each *true* value establishes a fixed distance between that side of the panel and the window's edge. For example, setting only *left* and *right* to *true* makes the panel move horizontally as the window widens and narrows, but the panel does not move vertically when the window's height changes.

If you are setting these values identically for a group of objects, use AutoMoveSize to define the settings then add the appropriate *x*AutoMoveSize constant (such as panAutoMoveSize for panels) to the objects' spec as they are created. The objects will adopt the settings specified by the AutoMoveSize routine.

☞ **Warning**: Make sure that you resize objects in a way that makes sense. Don't allow a window to shrink down to a size where objects become unusable or disappear altogether.

SetPanelFontSettings

Set a panel's font, size and style settings.

C
```
pascal void SetPanelFontSettings (short Panel,
            short theFont, short theSize, Style theStyle);
```

Pascal
```
procedure SetPanelFontSettings (Panel: INTEGER;
            theFont: INTEGER; theSize: INTEGER; theStyle: Style);
```

Panel specifies the panel number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, if no windows are open, or if the panel does not exist, SetPanelFontSettings does nothing. Otherwise, the change is seen immediately.

TheFont specifies the panel's new font. The default is Chicago, which is represented by the systemFont constant.

TheSize specifies the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle specifies the panel's new style. Special character constants defined by the Font Manager are bold, italic, underline and shadow. C programmers use the Font Manager's constants to specify a composite style, such as SetPanelFontSettings(1, geneva, 9, bold + outline) for bold and outlined, or SetPanelFontSettings(1, geneva, 9, 0) for plain text. Pascal programmers use the Font Manager's constants to specify a style set, such as SetPanelFontSettings(1, geneva, 9, [bold, outline]) for bold and outlined, or SetPanelFontSettings(1, geneva, 9, [ ]) for plain text.

A panel's font settings are set when a panel is created, so this routine is not normally used by many applications.

GetPanelFontSettings

Get a panel's font, size and style settings.

C
```
pascal void GetPanelFontSettings (short Panel,
            short *theFont, short *theSize, Style *theStyle);
```

Pascal
```
procedure GetPanelFontSettings (Panel: INTEGER;
            var theFont: INTEGER; var theSize: INTEGER; var theStyle: Style);
```

Panel specifies the panel number (from 1 to 511) in the current window whose font settings are being retrieved. If the current window doesn't belong to your application, if no windows are open, or if *Panel* specifies a panel that does not exist, GetPanelFontSettings returns default values.

TheFont is the panel's font number. The default is 0 which is represented by the systemFont constant.

TheSize is the font's size. The default is 0, which represents the default font size used by the system font, or 12pt in this case.

TheStyle is the panel's font style. The default is plain text, which is represented by 0 in C and [ ] in Pascal.

SetPanelColors

Set a panel's colors.

C
```
pascal void SetPanelColors (short Panel, const RGBColor *Text,
            const RGBColor *Background, const RGBColor *Border,
            const RGBColor *Hilite, const RGBColor *Shadow,
            const RGBColor *TextShadow);
```

Pascal
```
procedure SetPanelColors (Panel: INTEGER; Text, Background, Border, Hilite,
            Shadow, TextShadow: RGBColor);
```

Panel specifies the panel number (from 1 to 511) in the current window whose colors are being set. If the current window doesn't belong to your application, or if no windows are open, SetPanelColors does nothing. Also, if *Panel* specifies a panel that does not exist, SetPanelColors does nothing. The change is seen immediately. SetPanelColors automatically creates a custom color table if required for the panel.

Text is the color of the panel's title.

Background is the color of the panel's background. By default, the panel uses the window's backdrop color but it may be instructed to use this background color.

Border is the color of the panel's border.

Hilite is the color that is used to draw highlights on both the title and the panel. This color is usually white.

Shadow is the color used to draw the panel's shadows

TextShadow is the color used to draw soft shadows on the title.

TextHeavyShadow is the color used to draw heavy shadows on the title.

Also see: GetPanelColors.

GetPanelColors

Get a panel's colors.

C
```
pascal void GetPanelColors (short Panel, RGBColor *Text,
            RGBColor *Background, RGBColor *Border, RGBColor *Hilite,
            RGBColor *Shadow, RGBColor *TextShadow);
```

Pascal
```
procedure GetPanelColors (Panel: INTEGER; var Text: RGBColor;
            var Background: RGBColor; var Border: RGBColor;
            var Hilite: RGBColor; var Shadow: RGBColor;
            var TextShadow: RGBColor);
```

Panel specifies the panel number (from 1 to 511) in the current window whose colors are being retrieved. If the current window doesn't belong to your application, or if no windows are open, or if *Panel* specifies a panel that does not exist, GetPanelColors returns color values from the standard color table.

Text is the color of the panel's title.

Background is the color of the panel's background. By default, the panel uses the window's backdrop color but it may be instructed to use this background color.

Border is the color of the panel's border.

Hilite is the color that is used to draw highlights on both the title and the panel. This color is usually white.

Shadow is the color used to draw the panel's shadows

TextShadow is the color used to draw soft shadows on the title.

TextHeavyShadow is the color used to draw heavy shadows on the title.

Also see: SetPanelColors.

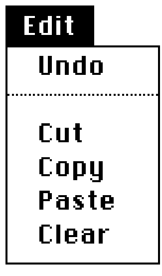
---

13 Menus

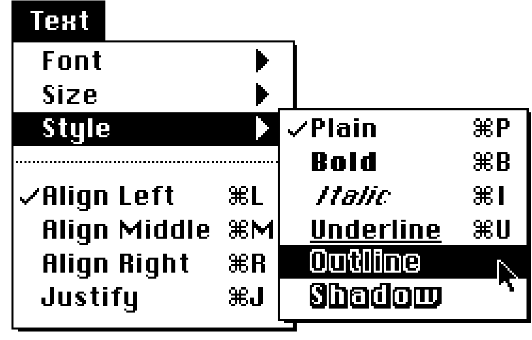
Tools Plus lets your application implement and support menus with considerably less effort than using the Macintosh toolbox's routines. The fully automated Apple menu () lets you launch, activate, and interact with desk accessories without having to write any code. Tools Plus also integrates the edit menu's Undo, Cut, Copy, Paste, and Clear commands with editing fields, desk accessories and other applications so you don't have to write any code to Cut, Copy, Paste, or Clear text, or to undo/redo your last operation.

There are two different kinds of menus your application can use, pull-down menus and hierarchical menus, as illustrated below. When implementing menus, your application should adhere to Macintosh User Interface Guidelines as expressed throughout the series of Inside Macintosh manuals.

In this document, the term *menu* refers to the entire menu object; that is, [1] the entire pull-down menu and the name that appears in the menu bar, or [2] an entire hierarchical menu. The term *menu item* or *item* refers to individual items found within a menu. The item number is determined by counting from the top of the list, the first item being 1, the second being 2, etc.



Pull-Down menu



Hierarchical menu

Menus can be created and maintained on an item by item basis within your application. You can also create an entire menu from a 'MENU' resource by using the LoadMenu routine. The LoadMenuBar routine reads an 'MBAR' menu bar resource that lists all the menus in a menu bar, and creates those menus.

Your application can create the Apple menu () with the AppleMenu routine. This routine creates the Apple menu, inserts an optional "About…" item (first item) that is used to invoke your application's "about box," and populates the menu with a list of desk accessories.

A pull-down menu is created by the Menu routine. First, your application specifies the menu's name (which appears in the menu bar) and a *menu number* that can be from 1 to 15. The menu number refers to that specific menu until the menu is deleted. The Menu routine is then used to add *items* to a specific menu.

A hierarchical menu is also created by using the Menu routine. A name does not have to be specified for a hierarchical menu because this type of menu does not appear in the menu bar. Hierarchical menu numbers can be from 16 to 200. The menu number refers to that specific hierarchical menu until it is deleted. The Menu routine is then used to add *items* to a specific hierarchical menu. You attach a hierarchical menu to an item in a pull-down menu or a hierarchical menu by using the AttachMenu routine. When a hierarchical menu is attached to another menu, it is often called a *submenu*. In this relationship where a menu has one or more submenus, the owner of the submenus is often called the *parent* menu, and the submenus are sometimes referred to as *offspring* or *child* menus.

Menu items can also be inserted between others using the InsertMenuItm routine. This lets your application maintain a dynamic menu that may be used, for example, for a list of open document names. ResNamesToMenu inserts resource names (such as fonts or sounds), sorted alphabetically, at a specified item.

An entire menu can be deleted by using the RemoveMenu routine. This routine reclaims the memory used by the menu. Individual items can also be deleted using this routine. You can delete all menus in a menu bar with the RemoveMenus routine, or all menus and hierarchical menus with the RemoveAllMenus routine. Your application can temporarily hide the menu bar then redisplay it with MenuBarDisplay.

Menu items can be renamed by using the RenameItem routine. This should be done judiciously, since changes to menus and/or menu items may prove to be confusing to the user.

An entire menu can be enabled or disabled with the EnableMenu routine, as can individual menu items. When an entire menu is disabled, it is dimmed along with all its associated items, and it cannot be selected. When an item is disabled, it becomes dim and cannot be selected.

Changes made to the menu bar by adding, deleting, enabling or disabling pull-down menus (not individual items), appear in the menu bar as soon as your application calls ProcessEvents, ProcessToolboxEvent, or Process1EventWhileBusy. Menu bar changes you make from inside your event handler routine will be seen when your event handler finishes executing. You can use the UpdateMenuBar routine if you want the changes to appear right away, such as when your application is starting up and it may be several seconds before you call ProcessEvents.

Various other menu item-related features are supported, such as setting or removing “check marks” with the CheckMenu routine. You can set or remove other marks with the MenuMark routine, and determine which mark is displayed by using GetMenuMark. You can set and retrieve an item’s Command-key character with MenuCmd and GetMenuCmd. The same applies for icons that are displayed in menus, which are set and retrieved with MenuIcon and GetMenuIcon. An item’s text is retrieved with GetMenuString, and its style is set with MenuStyle.

Note: For the sake of consistency with other applications, your application *should* have the following menus even if they are not accessed:
- Apple menu ()
- File menu (first menu). The last item should be “Quit”
- Edit menu (second menu) with the standard items as defined later is this chapter.

Menus in Plug-Ins

If you are writing a plug-in, the host application will have created its menus before calling your plug-in. Do not create new menus, or delete or modify the host application’s menus. By default, opening a modal window will disallow access to the host application’s pull-down menus. If the host application’s Edit menu follows the standards defined in this manual, you can use the wAllowEditMenu option when opening your modal window.

Colors

By default, menus have black text on a white background. The menu bar is white and the titles in the menu bar are black. You can get and set the default menu colors for your application using GetMenuBarColors and SetMenuBarColors. These defaults apply to all pull-down and hierarchical menus in your application. You can also get and set the colors for a specific menu by using he GetMenuColors and SetMenuColors routines. If you want to get or set the colors for a single menu item, use the GetMenuItemColors and SetMenuItemColors routines. In all cases, use color very judiciously, and only if there is value in adding colors.

The Appearance Manager does not support the use of colors in menus (it supplies colors and patterns that are consistent with the user-selected theme). Initializing Tools Plus with the initPureAppearanceManager option enforces this principle by ignoring custom color information when the Appearance Manager is available.

Menus Accessed by MultiFinder and System 7 or higher

This section describes how MultiFinder and System 7 or higher automatically interact with your application’s menus if your application does *not* support high-level events (also called Apple Events). You should set the appropriate bit in your application’s ‘SIZE’ resource to indicate if it supports Apple Events or not. See the Completing Your Application chapter for details about the ‘SIZE’ resource.

Both MultiFinder (running under System 5 or 6) and System 7 or higher can automatically interact with your application through its menus. If your application is running while the user double-clicks (or select-opens) one of your application's documents from the Finder, the affected document is automatically opened by your application. Also, if the user selects the Special menu's Restart or Shut Down command while your application is running, your application is instructed to quit.

In both these cases, the system *simulates* the selection of a menu item. You must have a File menu with an item named "Open…" (including the ellipsis, created using Option-;), and the last item named "Quit". In the case of opening a document, Tools Plus reports a doMenu event to your application indicating that the File menu's "Open…" command was selected, in which case your application would do whatever is appropriate, likely display an SFGetFile dialog to let the user choose which file to open. The system fools your application into thinking that the double-clicked file was selected from an SFGetFile dialog (which is not actually displayed). When the user selects Restart or Shut Down, Tools Plus reports a doMenu event to your application indicating that the File menu's "Quit" command was selected, in which case your application would do whatever is appropriate, such as asking the user if open documents should be saved before quitting.

If your application does not [1] open files by using the File menu's "Open…" command, or [2] quit by using the File menu's "Quit" command, see the Completing Your Application chapter where you can remap these functions to other menu items.

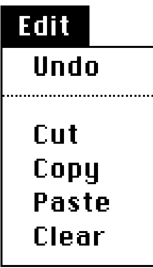
See the Completing Your Application chapter. The section on "bundles" defines the association between your application and the files it creates, while the section on 'mstr' resources details remapping the "Open…" and "Quit" functions to menus and menu items that are differently named.

Edit Menu

The Edit menu has a high degree of consistency across all Macintosh applications in appearance and function. Your application should have an Edit menu if your application:

- has two or more menus in the menu bar (excluding the Apple menu ())
- uses editing fields
- supports desk accessories and runs under systems prior to System 7

It is good form to include an Edit menu in your application even if it doesn't support its functions, just for the user's familiarity.



Your application's second menu must be called "Edit." The Edit menu must contain the items "Undo", "Cut", "Copy", "Paste" and "Clear" as the first five items in the listed order. A dividing line must exist between "Undo" and "Cut". Tools Plus automatically enables and disables items in the Edit menu as described in the Editing Fields chapter under *The Edit Menu* whenever an editing field is active. It also automatically changes the "Undo" item to "Can't Undo" when:

- a window that allows access to pull-down menus is opened or closed
- an inactive window is activated
- an editing field is deactivated or activated (including selecting a new editing field)
- a desk accessory performs a Cut or Copy operation which affects the clipboard

Your application is responsible for maintaining the Edit menu whenever it is *not interacting with an active editing field*. For example, if your application performs a function that can't be undone, it should set the "Undo" item to "Can't Undo." Also, if it performs an operation that *can* be undone, it should change the "Undo" item to whatever is appropriate, such as "Undo Merging." The other items in the Edit menu, namely "Cut", "Copy", "Paste", and "Clear" should be enabled/disabled as required by your application. Your application can have additional items in the Edit menu after the Clear command.

If you are localizing your application for a language other than English, you will likely want to use menu titles that are different from those stated here. You may feel free to do so. See the Multiple Languages chapter for details on how to make the "Undo" menu display its status in the language(s) of your choice.

Select All

An item commonly found in many applications' Edit menu is "Select All" which is used to select all objects or text. Tools Plus supports this feature by letting you create a "Select All" item in the Edit menu anywhere after the "Clear" item. By using SetSelectAllItem, your application informs Tools Plus that a "Select All" item exists in the Edit menu. Tools Plus then enables/disables the item appropriately and selects all text in the active field when this menu item is used.

Note: Do not rename the Edit menu's "Undo" item, or change the enabled/disabled state of the Edit menu's "Undo", "Cut", "Copy", "Paste", or "Clear" items if the active window allows access to pull-down menus and it contains an active editing field. Tools Plus maintains these menu items automatically. If you are going to make such a change, preface it by a call to GetEditString (to retrieve the active field's edited text for validation by your application), a call to SaveFieldString (to save the edited text as the field's associated string), then to DeactivateField (to deactivate the window's active field and end Tools Plus automatic maintenance of the Edit menu).

Menus and Editing Fields

If your application is going to use editing fields, an "Edit" menu will greatly assist in text editing. The Edit menu automatically performs all text transfer operations between the active editing field and the Clipboard. This includes:

- Cutting the selected text from the active editing field and placing it on the Clipboard
- Copying the active editing field's selected text to the Clipboard
- Pasting the Clipboard's contents into the active editing field
- Undoing the last operation, including the Edit menu's Cut, Copy, Paste, and Clear operations, as well as typing and using the Delete (or Backspace) key
- Undoing the last Undo operation (i.e. Redo)

This functionality is detailed in the Editing Fields chapter of this manual.

When a window that allows access to pull-down menus is activated, or when an editing field is activated on the active window, the Edit menu is automatically updated to reflect the operations that can be performed on the active field. Any editing done in the field automatically updates the Edit menu's items. If you deactivate such a window containing an active editing field, either by closing the window or by activating another window that allows access to pull-down menus without an active field, all five Edit menu items are disabled.

Opening and closing a window that *disallows* access to pull-down menus does not affect the Edit menu, since menus cannot be accessed while that window is active.

Apple Menu and Desk Accessories

You can give your application the ability to use desk accessories (DAs) by installing the Apple menu (). Although desk accessory activity is handled entirely by Tools Plus, your application must adhere to some very strict rules. Your application must have at least two menus to ensure compatibility with System 5 and System 6. The first menu is called "File", and must contain "Quit" as the last item. The second menu must be the "Edit" menu, as described above. The edit menu must exist as described, even if your application does not support these commands. Lastly, you must ensure that sufficient free space exists in the menu bar to accommodate additional menus, since some DAs create their own menus when running under Finder in System 5 and System 6. Complying with these requirements will ensure that your application is compatible with desk accessories in all system versions.

Desk accessories affect the Edit menu as described later in this chapter in the "Menus and Desk Accessories" section.

See the Event Management chapter For more details about desk accessory interaction.

Menus and Desk Accessories (Using Finder under System 5 or 6)

This section applies only when your application is running under Finder (not MultiFinder) in System 5 or System 6. Tools Plus automatically maintains menus according to Macintosh User Interface Guidelines (described below).

When a desk accessory is activated, Tools Plus does the following things:

- your application's menus are memorized so they can be restored to their original state when your application becomes active again
- all items in your application's File menu are disabled, except for the last item, "Quit"
- the Edit menu's Undo item is renamed to "Undo". This is done in view that the item may have read "Undo Typing" or some other specific action that does not pertain to a desk accessory
- the Edit menu's "Undo", "Cut", "Copy", "Paste" and "Clear" items are all enabled for the desk accessory's use
- all other Edit menu items are disabled
- all other menus in your application are disabled.

When the desk accessory becomes inactive, either by being closed or by activating a window in your application, your application's menus are restored to their original state.

Menus and Desk Accessories (Using MultiFinder or System 7 or higher)

This section applies only when your application is running under MultiFinder (not Finder) in System 5 or System 6, or System 7 or higher. Each application has its own menu bar including desk accessories. When your application is active its menu bar is displayed. When another application (including System 5 or System 6's MultiFinder, and System 7's or higher Finder), or a desk accessory is active, the other application's menu bar is displayed. So if your application will run only on MultiFinder in System 5 or 6, and/or System 7 or higher you can be certain that desk accessories will not be adding menus to your menu bar and that they will not be interacting with your menus.

Help Menu and Applications Menu

System 7 or higher automatically installs two menus that appear as icons on the right side of your application's menu bar. They are the Help menu and the Applications menu. All functions in the Applications menu are handled automatically, so the only access Tools Plus provides to this menu is to enable it or disable it. It is unlikely you will ever need to use this feature.

The Help menu is installed in your application's menu bar when your application creates the Apple menu in System 7 or higher. The Help menu contains several items that are automatically handled by the system. The number of these items and their function depends on which version of the System you are running. It is not important that you know what they are. If your application does not add any functionality to the default Help menu, you should never access the Help menu with any menu management routines and Tools Plus will never report any events from the Help menu.

If your application needs to add its own items to the help menu, first use the MenuItemCount routine to determine the number of items in the Help menu (the mHelpMenu constant provides a convenient way to identify the Help menu's menu number). The first time your application accesses the Help menu through any Tools Plus menu routine, a dividing line is automatically appended to end of the help menu (this is done by the Macintosh's toolbox and cannot be overridden). You may then add and maintain items in the Help menu using Tools Plus's menu routines. Tools Plus does not allow you to affect the items that are automatically installed and handled by the system. When the user selects an item from the Help menu, Tools Plus reports a doMenu event telling your application the menu number (mHelpMenu) and item number selected by the user. Your application will only be informed when the use selects items you have installed in the Help menu, and not when the user selects any of the system-installed items.

If your application is running under System 6 or earlier, menu routines referring to the Help menu or Applications menu will not do anything.

Command Key Equivalents

Menus can be invoked from the keyboard by setting up a Command key (⌘-) equivalent. The character you specify for a Command key equivalent will likely be a letter, however, other characters may be used. When the user holds down the Command key and types a letter, whether in upper or lower case, the equivalent menu item is invoked as if the menu had been pulled down and the required item selected. The appropriate name in the menu bar is highlighted while the operation takes place.

The system responds equally to shifted or non-shifted characters. For example, ⌘-C and ⌘-c both invoke the same menu item. Consequently, ⌘-+ is read by the system as ⌘-=. For consistency between applications, upper case letters should be used.

⌘-Shift-number combinations are not Command key equivalents, because they are processed as a separate type of event (this is why ⌘-Shift-1 and 2 eject the internal and external floppy disk). Although unshifted ⌘-number equivalents can be used, they should be used judiciously to avoid confusion.

Note: Several Command key equivalents are reserved for specific purposes, as listed below:

File menu		Edit menu		Style menu	
⌘-N	(New)	⌘-Z	(Undo)	⌘-P	(Plain, less common)
⌘-O	(Open…)	⌘-X	(Cut)	⌘-B	(Bold)
⌘-W	(Close)	⌘-C	(Copy)	⌘-I	(Italic)
⌘-S	(Save)	⌘-V	(Paste)	⌘-U	(Underline)
⌘-P	(Print…)	⌘-A	(Select All)		
⌘-Q	(Quit)				

Note: The Macintosh’s Menu Manager has some inherent properties that Tools Plus cannot override:
• menu items that have submenus cannot have a command key, nor can they display an SICN icon
• menu items that display an SICN icon cannot have a command key.

Planning for Balloon Help

The Help Manager implementatiion of Balloon Help has a limitation you should consider when designing your application: Balloon Help for menus cannot be implemented dynamically. You must create an ‘hmnu’ resource, and Tools Plus will attach it to the ‘MENU’ resource with the same resource ID.

Handling Menus

Tools Plus takes care of maintaining the Edit menu when an editing field is active on a window that allows access to pull-down menus. When a menu item is selected by the user, either by using the mouse or by typing a Command key equivalent, the corresponding menu name in the menu bar is highlighted and Tools Plus reports a doMenu event with the menu number and item number selected by the user. After your application has responded to the selection, call MenuHilite(0) to un-highlight the menu bar.

Hierarchical menus are handled identically to pull-down menus, except when they are selected by the user, the name of their *ultimate parent* is highlighted in the menu bar. If the user types a Command key equivalent for a hierarchical menu that is *not* attached to a pull-down menu, Tools Plus reports a doKeyDown or doAutoKey event for that keyboard event.

If the Edit menu’s Undo, Cut, Copy, Paste, or Clear are selected while an editing field is active, Tools Plus automatically processes the command and an event is not reported to your application.

If your application does not respond to Apple Events and is running under System 5 or 6’s MultiFinder (not Finder) or System 7 or higher, the user can double-click one of your application’s documents while your application is running. This generates a doMenu event that is equivalent to the File menu’s “Open…” item in your application. The user can also select the Special menu’s Restart or Shut Down commands, which generates a doMenu event that is equivalent to the File menu’s “Quit” item, instructing your application to quit. See the Event Management chapter for complete details on the handling of menus.

AppleMenu

Create the “Apple” menu.

C
```
pascal void AppleMenu (const Str255 AboutName);
```

Pascal
```
procedure AppleMenu (AboutName: STRING);
```

This routine creates the Apple menu (), inserts an optional “About…” item that is used to invoke your application’s “about box,” and populates the menu with a list of desk accessories. AppleMenu should be called before defining any other menus.

AboutName specifies the title of the Apple menu’s “About” item, such as “About my application...” This item is first in the Apple menu, followed by a dividing line, then by the desk accessories listed in alphabetic order. If AboutName is a null string, only the desk accessory names will appear in the Apple menu.

This routine does not immediately display the Apple menu in the menu bar to prevent the menu bar from flickering each time a new menu is added. Instead, all changes to the menu bar appear simultaneously when your application calls starts processing events, or when it finishes executing an event handler routine. Call UpdateMenuBar if you need the changes to appear right away.

See “Desk Accessories” at the beginning of *Menus* for desk accessory menu requirements.

Menu

Create a menu (pull-down or hierarchical), add more menu items, or rename existing menu items.

C
```
pascal void Menu (short MenuNumber, short ItemNumber, Boolean EnabledFlag,
            const Str255 MenuText);
```

Pascal
```
procedure Menu (MenuNumber, ItemNumber: INTEGER; EnabledFlag: BOOLEAN;
            MenuText: STRING);
```

When a pull-down menu is first created, the menu’s name (which appears in the menu bar) must be defined first. This is done by calling the Menu routine with *ItemNumber* having a value of zero (0). Menu items can then be added to the menu. Your application should define menu items in their correct order (i.e., top to bottom) in order to use the full power of this routine. A hierarchical menu is created the same way as a pull-down menu, except that you don’t need to create a menu name because hierarchical menus never appear in the menu bar.

MenuNumber specifies the menu number that is affected. The numbers **1** through **15** are reserved for pull-down menus, and the numbers **16** through **200** are reserved for hierarchical menus. Once a menu is created, it is referenced by this menu number. Menu names for pull-down menus are displayed in your application’s menu bar with the Apple () menu (if one was created) in the leftmost position, followed by the other pull-down menus in the ascending order of their *MenuNumber*. Use mHelpMenu to work with the Help menu.

ItemNumber specifies the menu’s item number (from 1 to 32767) that is affected. If *ItemNumber* is zero (0) for a pull-down menu, the Menu routine refers to the menu’s name in the menu bar. The first two menus (usually File and Edit) are limited to 31 items. You cannot change the items installed by the system in the Help menu.

EnabledFlag specifies whether the menu or menu item is enabled or disabled. The menu or menu item can be selected only when enabled. When disabled, the menu or menu item is dimmed and cannot be selected by the user. The two constants that can be used for this purpose are *enabled* and *disabled.* If the *ItemNumber* is zero, the action applies to all items within the specified menu. When the menu later becomes enabled, all items in the menu assume their correct enabling/disabling as specified by your application. Menus and menu items can be enabled and disabled by using the EnableMenu routine.

MenuText is the menu's name that appears in the menu bar (if *ItemNumber* is zero), or the menu item's name (if *ItemNumber* is not zero). When a menu item is first created, certain *metacharacters* are recognized by Tools Plus to provide special instructions to the Menu Manager. You may choose to include or exclude these characters within *MenuText*, however, you should be aware of their effects. A menu's name (which appears in the menu bar) should not have any metacharacters. Menu items can include multiple metacharacters.

Note: The Macintosh's Menu Manager allows only the first 31 items of a menu to be disabled individually. The entire menu, however, can always be disabled.

Metacharacters

Metacharacters are symbols that tell the Menu Manager to perform special functions on a menu. They are recognized and processed only when a *menu item* is first created, and are ignored (displayed as ordinary characters) when menu items are renamed. A menu's name (which appears in the menu bar) should not have any metacharacters. Menu items can include multiple metacharacters or combinations of metacharacters.

Unlike the Macintosh toolbox's menu routines, Tools Plus removes the semi-colon (;) and Return character ($0D), and does not process them as metacharacters.

Metacharacter	Meaning
^	Display an icon to the left of the menu item. The number following the caret (^) should be from 1 to 255 (i.e., "^28"). The Menu Manager adds 256 to the number you state to specify a resource ID that is in the range of 257 to 511, so if you specify 28, resource ID 284 is used (28 + 256 = 284). These icon resources are read from your application. Tools Plus tries to use a 'cicn' icon if Color QuickDraw is available on the Macintosh running your application. Otherwise, it will search for an 'ICON' (black and white) icon, then an 'SICN' icon. Unlike the equivalent Macintosh toolbox routine, your menu item will remain unaffected if the specified icon can't be found (i.e., empty space is not reserved in the menu). Be aware that the Menu Manager drawing a 'cicn' icon in color will do so even if the icon was created using 8-bit colors and the monitor is set to 4-bits. This may produce unsatisfactory results. If possible, use 4-bit colors or colors that translate well into 4-bit colors.
!	Display a special mark to the left of a menu item. The single character that follows the exclamation mark (!) is displayed. The check mark is the default. (It is best to use the CheckMenu or MenuMark routines.)
<	The item is displayed in a special character style. The single character that follows this symbol specifies the style (Bold, Italic, Underline, Outline, or Shadow). Multiple styles can be combined, such as "<B<I" for "bold and italic." It is best to use MenuStyle to change styles.
/	The slash (/) indicates that a menu item may be invoked by using a keyboard equivalent. The single character that follows the slash specifies the Command key equivalent to a menu item. For consistency, use upper case letters (shifted characters are ignored, so ⌘-+ is interpreted as ⌘-=). Do not use Control characters, since older Macintosh keyboards such as those found on the Macintosh Plus don't have a Control key. Tools Plus prevents you from using characters $1B through $1F (Control-[, Control-\, Control-], Control-Up Arrow, and Control--), because these characters are reserved by Apple for the system's use.

Note: The Macintosh's Menu Manager has some inherent properties that Tools Plus cannot override:
- menu items that have submenus cannot have a command key, nor can they display an SICN icon
- menu items that display an SICN icon cannot have a command key.

To create a "dividing line" between sections of related menu items, disable the menu item and use '-' (a minus or dash) as the MenuText value. You can use the constant mDividingLine for this purpose.

Special care should be taken to define the menu's name in the menu bar first, then to define menu items in their correct sequential order. If a pull-down menu item is defined without first defining the menu's name, the name in the menu bar is automatically defined as "Menu *x*" where *x* is the menu number. Also, if any menu items are skipped when defining a menu item (i.e., creating menu item 3 without first creating 1 and 2) the missing menu items (1 and 2) are automatically created as blank, disabled items. Consequently, metacharacters will not be recognized when the Menu routine references these automatically created items; the Menu routine will simply rename the existing item.

A menu's name (which appears in the menu bar) cannot be changed. If the menu's name must be changed, the affected menu must be removed with the RemoveMenu routine, then re-created as required by using the Menu routine.

This routine does not immediately display the newly added menu in the menu bar to prevent the menu bar from flickering each time a new menu is added. Instead, all changes to the menu bar appear simultaneously when your application calls starts processing events, or when it finishes executing an event handler routine. Use UpdateMenuBar if you need the changes to appear right away.

Programming Tips:

1 If you want to add resource names to a menu, such as fonts, use the ResNamesToMenu routine.

2 Tools Plus assumes that menu number two is the Edit menu. If your application does not have an Edit menu, make sure your application does not have a menu numbered two.

3 If you need any of the metacharacters to appear in a menu's text (such as an exclamation mark), first create a blank menu item (MenuText equals a space), then change the item's text with the Menu or RenameItem routine to include the desired characters. Metacharacters are displayed but not specially processed when a menu item's name is changed.

Warning: The first two menus (usually File and Edit) are limited to 31 items each. This limit has been imposed to ensure that an application running under Finder (System 5 and 6) can adhere to Macintosh User Interface Guidelines.

```
CONST                                      {Menu constants:                        }
               mFileMenu         =  1;     {File menu number                       }
               mEditMenu         =  2;     {Edit menu number                       }
               mHelpMenu         = -2;     {Help menu number (System 7.0 or higher)  }
               mDividingLine     = '-';    {Dividing line                          }

               enabled           = true;   {Enable the menu/item                   }
               disabled          = false;  {Disable the menu/item                  }
```

LoadMenu

Create a menu using a 'MENU' resource.

C
```
pascal void LoadMenu (short MenuNumber, short ResID);
```

Pascal
```
procedure LoadMenu (MenuNumber, ResID: INTEGER);
```

LoadMenu lets you create a menu along with all its items using a 'MENU' resource. If the 'MENU' resource contains references to hierarchical menus, those submenus are also loaded and attached to the menu. LoadMenu also loads the corresponding 'mctb' menu color table resource if it is available. You can create both the 'MENU' resource and its related 'mctb' resource using a resource editor such as Apple's ResEdit.

MenuNumber specifies the menu number that is created. The numbers **1** through **15** are reserved for pull-down menus, and the numbers **16** through **200** are reserved for hierarchical menus. Once a menu is created, it is referenced by this menu number. Menu names for pull-down menus are displayed in your application's menu bar with the Apple () menu (if one was created) in the leftmost position, followed by the other pull-down menus in the ascending order of their *MenuNumber*. Use the mAppleMenu constant to specify that the Apple menu is being created from a resource. If a menu already exists that is using the same menu number, it is deleted before the new menu is created.

ResID is the 'MENU' resource ID number that is used to create the menu. If the menu has an 'mctb' color table resource, it must use the same ID number. The resource ID number can be the same as the menu number, that being **1** through **15** for pull-down menus and **16** through **200** for hierarchical menus. In the case of the Apple menu, the resource number can be zero (0) as well. If your application creates only one set of menus and stays with them, it is best to number your ResID the same as your MenuNumber. If your application deletes and creates menus during execution, use ResIDs that are in the range of **16000** to **31999**. These resource numbers don't overlap the range used by menu numbers, so you can think of them as a temporary holding area for 'MENU' resources that have not become

usable menus.

When creating menus using ‘MENU’ resources, please note the following:

- Flag your ‘MENU’ and ‘mctb’ resources as purgeable to save memory. Tools Plus makes a copy of their data.
- The Apple, File and Edit menu must be the first menus in your menu bar.
- The File and Edit menus use menu numbers 1 and 2. The constants mFileMenu and mEditMenu are available.
- The File and Edit menu are limited to 31 items. Subsequent items are not installed.
- Hierarchical menus lose their title when they are installed. Tools Plus uses this field for a reference to the parent menu.
- If your menu has a reference to a submenu, that submenu resource ID must be in the range of **16** through **200**, just like hierarchical menu numbers. If the reference to the submenu’s resource ID is outside this range, the submenu is not loaded.
- If your Edit menu has a “Select All” item, use the SetSelectAllItem routine to tell Tools Plus which item it is after using LoadMenu.
- You can create menus using resources and/or programatically with the Menu routine.
- Macintosh’s Menu Manager automatically loads ‘mctb’ resource with and ID of zero (0) when your application starts and applies the resource’s setting to all menus in your application. For this reason, if you create an Apple menu with a ‘MENU’ resource ID of zero (0), LoadMenu will not load the ‘mctb’ resource with an ID of zero (0).

LoadMenuBar

Create a set of menus using an ‘MBAR’ resource.

C
```
pascal void LoadMenuBar (short ResID);
```

Pascal
```
procedure LoadMenuBar (ResID: INTEGER);
```

ResID is the ‘MBAR’ resource ID number that is used to create the set of menus. You can create the ‘MBAR’ resource using a resource editor such as Apple’s ResEdit. The ‘MBAR’ resource lists resource IDs for ‘MENU’ resources that are used to create the menu bar’s pull-down menus. See the LoadMenu routine for details about ‘MENU’ resources and their ID numbering requirements.

LoadMenuBar first calls RemoveAllMenus to remove all existing pull-down menus from the menu bar and all hierarchical menus. It then reads sequentially through the ‘MBAR’ resource to determine the ‘MENU’ resource IDs, and loads those resources to make them into menus using the LoadMenu routine. Pull-down menu numbers are assigned sequentially starting at one (1) and incrementing by one.

If your application has only one menu bar, it is best to number your ‘MENU’ resource IDs the same as your menu numbers. For example, use 0 for your Apple menu, 1 for your File menu, 2 for your Edit menu, and so on. Subsequent menu bars must use ‘MENU’ resource IDs that are in the range of **16000** to **31999** for all pull-down menus other than the Apple menu. These resource numbers don’t overlap the range used by menu numbers (1 through 15), so you can think of them as a temporary holding area for ‘MENU’ resources that have not become usable menus yet. ‘MENU’ resource IDs for submenus must be in the range of **16** to **200**.

LoadMenuBar recognizes the Apple menu as a special case and does not include it as a sequentially numbered menu. It is most convenient to always use ‘MENU’ ID 0 for the Apple menu. Flag ‘MBAR’ resources as purgeable to conserve memory. The following are examples of typical ‘MENU’ resource ID specified in an ‘MBAR’ resource:

‘MBAR’ ID = 128 *App’s first menu*	‘MBAR’ ID = 129 *Second menu bar*	‘MBAR’ ID = 130 *Third menu bar*
0 = Apple 1 = File 2 = Edit 3 = LogOn	0 = Apple 16000 = File 16001 = Edit 16002 = Styles 16003 = Colors 16004 = Models	0 = Apple 1 = File 2 = Edit 16010 = Security 16011 = Users

SetSelectAllItem

Identify the Edit menu's "Select All" item.

C
```
pascal void SetSelectAllItem (short ItemNumber);
```

Pascal
```
procedure SetSelectAllItem (ItemNumber: INTEGER);
```

If you want to include a "Select All" item in your applications Edit menu, first create the item using the Menu routine then use SetSelectAllItem to identify the item number in the Edit menu. Tools Plus then enables/disables the item appropriately and selects all text in the active field when this menu item is used.

ItemNumber specifies the menu item number (from 7 to 31) where the "Select All" item is located in the Edit menu. If the menu item does not exist in the Edit menu, or if the Edit menu does not exist, SetSelectAllItem does nothing. Use "0" if you want Tools Plus to ignore a previously set item.

SetSelectAllItem is automatically invoked if you create an Edit menu and it has an item named "Select All". You'll need to use this routine if you have a "Select All" item that is named differently, which may be the case if your application is localized to a non-English language.

GetFreeMenuNum

Get the first unused pull-down menu number.

C
```
pascal short GetFreeMenuNum (void);
```

Pascal
```
function GetFreeMenuNum: INTEGER;
```

Some developers may prefer to write code that more closely resembles a traditional Macintosh application, in that creating an object returns a reference to it such as a handle or pointer. Instead of having to assign your own pull-down menu number, GetFreeMenuNum returns the first unused (free) menu number. Using this routine, you can assign an unused menu number to a variable, then use that variable throughout your application without concern for the true menu number.

If the maximum number of pull-down menus has already been created (no new ones can be created), GetFreeMenuNum returns a value of zero (0).

GetFreeHMenuNum

Get the first unused hierarchical menu number.

C
```
pascal short GetFreeHMenuNum (void);
```

Pascal
```
function GetFreeHMenuNum: INTEGER;
```

This routine is identical to GetFreeMenuNum, except that it returns a hierarchical menu number.

AttachMenu

Attach a hierarchical menu to a menu item, or detach a hierarchical menu from a menu item.

C
```
pascal void AttachMenu (short MenuNumber, short ItemNumber,
            short SubMenuNumber);
```

Pascal
```
procedure AttachMenu (MenuNumber, ItemNumber, SubMenuNumber: INTEGER);
```

A hierarchical menu can be attached to a pull-down menu or to another hierarchical menu. When attaching a hierarchical menu to a "parent" (or if you prefer to view it the other way, providing a "parent" menu with an "offspring" hierarchical menu), you must specify the parent menu's menu number and item number, and the submenu's (offspring) menu number.

You can detach a submenu from its parent in two different ways:
[1] you can state that the "parent menu has no offspring": AttachMenu (3, 14, none)
[2] you can state that the "hierarchical menu has no parent": AttachMenu (0, 0, 105)
Notice that you can use the constant *none* to make your code more readable.

MenuNumber specifies the "parent" menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) to which the hierarchical menu is attached. You can specify zero (0) to detach a parent menu from a known submenu. If the menu number does not exist, AttachMenu does nothing. You cannot attach hierarchical menus to the Apple Menu or System 7's (or higher) Help menu or Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) to which the hierarchical menu is attached. You can specify zero (0) to detach a parent menu from a known submenu. If the menu item does not exist in the specified menu, AttachMenu does nothing.

SubMenuNumber specifies the "offspring" menu number (from 16 to 200) which is attached to the parent menu. You can specify zero (0) to detach a known parent menu from its submenu. If the menu number does not exist, AttachMenu does nothing.

✍ **Note**: AttachMenu ensures that a hierarchical menu is attached to only one parent menu item at a time by automatically detaching it from the old parent menu. Tools Plus will not allow a hierarchical menu to be attached to itself, or to result in a "cyclical hierarchy" in which the parent menus eventually lead back to a submenu. AttachMenu will beep if you attempt to define a cyclical hierarchy.

✍ **Note**: When a submenu is attached to a parent menu's item, that item's Command key and "mark" (as defined by MenuMark) are cleared. Also, if an SICN icon is displayed in the item, it too is cleared. The Macintosh's Menu Manager uses these characters to make hierarchical menus work.

InsertMenuItm

Insert a menu item into an existing menu.

C
```
pascal void InsertMenuItm (short MenuNumber, short ItemNumber,
            Boolean EnabledFlag, const Str255 MenuText);
```

Pascal
```
procedure InsertMenuItm (MenuNumber, ItemNumber: INTEGER;
            EnabledFlag: BOOLEAN; MenuText: STRING);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) where the menu item is created. If the menu does not exist, InsertMenuItm does nothing. Use mHelpMenu to work with the Help menu. You cannot insert items into the Apple Menu or System 7's (or higher) Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) where the item is inserted. If the menu item does not exist in the specified menu, InsertMenuItm does nothing. InsertMenuItm will *append* one item to the end of a menu if the ItemNumber equals the current number of items plus 1.

EnabledFlag specifies whether the item is enabled or disabled. In the enabled state, the item can be selected whereas in the disabled state, the item is dimmed and cannot be selected by the user. The two constants that can be used for this purpose are *enabled* and *disabled.* Menus and menu items can be enabled and disabled by using the EnableMenu routine.

MenuText is the name of the item. Certain *metacharacters* are recognized by Tools Plus to provide special instructions to the Menu Manager. You may choose to include or exclude these characters within MenuText, however, you should be aware of their effects. See the Menu routine for details on metacharacters.

When the item is inserted, all existing items starting at *ItemNumber* are pushed down one space to make room for the new item. This means that their item number will be changed. The new item is inserted at the location specified by *ItemNumber*. The main use for this routine is to let your application maintain a dynamic menu, such as a list of open document names.

Note: The Macintosh's Menu Manager allows only the first 31 items of a menu to be disabled individually. The entire menu, however, can always be disabled.

If a menu item that owns a submenu is disabled, either by disabling the item individually or by disabling the entire menu, the submenu cannot be viewed by the user.

Warning: The first two menus (File and Edit) are limited to 31 items each. This limit has been imposed to ensure that an application running under Finder (System 5 and 6) can adhere to Macintosh User Interface Guidelines.

```
CONST                                    {Menu and Menu Item status              }
               mHelpMenu         = -2;    {Help menu number (System 7.0 or higher)  }
               mDividingLine     = '-';   {Dividing line                            }
               enabled           = true;  {enable the menu/item                     }
               disabled          = false; {disable the menu/item                    }
```

ResNamesToMenu

Insert resource names into a menu.

C
```
pascal void ResNamesToMenu (short MenuNumber, short ItemNumber,
              ResType rType);
```

Pascal
```
procedure ResNamesToMenu (MenuNumber, ItemNumber: INTEGER;
              rType: RESTYPE);
```

This routine finds all named resources of the specified type and inserts those names (sorted alphabetically) into a menu. Duplicated names are ignored as are ones that start with "." (period) or "%".

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that is affected. If the menu does not exist, ResNamesToMenu does nothing. Use mHelpMenu to work with the Help menu. You cannot perform this operation on the Apple Menu or System 7's (or higher) Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) where the resource names are inserted. If the menu item does not exist in the specified menu, ResNamesToMenu does nothing. ResNamesToMenu will *append* to the end of a menu if the *ItemNumber* equals the current number of items plus 1. You cannot change the items installed by the system in the Help menu.

rType is the four character resource type whose names are being inserted into the menu. If you specify 'FOND' or 'FONT' resources, both are obtained since they are just different types of fonts.

When the resource names are inserted, all existing items starting at *ItemNumber* are pushed down to make room for the new items. This means that their item number will be changed. The new items are inserted starting at the location specified by *ItemNumber*.

Note: If the first character of a resource name is a dash (-), it is added into the menus as an option-dash (character 208) to prevent the Menu Manager from interpreting the name as a dividing line.

Warning: The first two menus (File and Edit) are limited to 31 items each. This limit has been imposed to ensure that an application running under Finder (System 5 and 6) can adhere to Macintosh User Interface Guidelines.

RemoveMenu

Delete a menu and its associated items, or delete an individual menu item.

C
```
pascal void RemoveMenu (short MenuNumber, short ItemNumber);
```

Pascal
```
procedure RemoveMenu (MenuNumber, ItemNumber: INTEGER);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that is affected. If the menu does not exist, RemoveMenu does nothing. Use mHelpMenu to work with the Help menu. You cannot remove the Apple Menu or System 7's (or higher) Help menu or Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) that is deleted. If *ItemNumber* is zero (0), RemoveMenu refers to the menu's name in the menu bar and all its associated items. If the menu item is not zero and it does not exist, RemoveMenu does nothing. You cannot remove the items automatically installed by System 7 or higher in the Help menu.

If a menu is deleted along with its associated items (i.e., when *ItemNumber* = 0), any menus with higher menu numbers is shifted to the left to fill in the space occupied by the deleted menu. Their menu numbers, however, remain unchanged. If the affected menu has submenus, those submenus are automatically detached but *not* deleted. If the affected menu is a submenu, it is automatically detached from its parent menu.

This routine does not immediately update the menu bar when a menu is deleted (to prevent the menu bar from flickering each time a change is made). Instead, all changes to the menu bar appear simultaneously when your application calls starts processing events, or when it finishes executing an event handler routine. Use UpdateMenuBar if you need the changes to appear right away.

Note: Use RemoveMenu to maintain a dynamic menu, such as a list of open windows. Do not use it to make items unavailable. Instead, disable items with EnableMenu

Warning: If you delete the File or Edit menu, make sure you replace it immediately. Failure to do so will result not only in user interface inconsistencies, but may also make your application unstable.

RemoveMenus

Delete all menus in the menu bar.

C
```
pascal void RemoveMenus (void);
```

Pascal
```
procedure RemoveMenus;
```

This routine simply calls RemoveMenu for each menu in the menu bar. Hierarchical menus, the Apple menu, and System 7's (or higher) Help menu and Applications menus are not affected. The menu bar is updated when your application finishes executing an event handler routine. Use UpdateMenuBar if you need the changes to appear right away.

RemoveAllMenus

Delete all menus in the menu bar and hierarchical menus.

C
```
pascal void RemoveAllMenus (void);
```

Pascal
```
procedure RemoveAllMenus;
```

This routine simply calls RemoveMenu for each menu in the menu bar and all hierarchical menus. The Apple menu, and System 7's (or higher) Help menu and Applications menus are not affected. The menu bar is updated when your application finishes executing an event handler routine. Use UpdateMenuBar if you need the changes to appear right away.

UpdateMenuBar

Display the menu bar with the changes made by the AppleMenu routine, Menu routine, RemoveMenu routine, or EnableMenu routine (when enabling/disabling an entire menu).

C
```
pascal void UpdateMenuBar (void);
```

Pascal
```
procedure UpdateMenuBar;
```

When any changes are made that affect the menu bar, all changes are displayed simultaneously when your application finishes executing an event handler routine (to prevent the menu bar from flickering with each change). Without this feature, several updates performed in a row such as creating several new pull-down menus at the beginning of a program or enabling/disabling several menus, would produce an annoying flicker with each change.

Normally, your application will not need to call UpdateMenuBar because the changes will be displayed when your application finishes executing an event handler routine. This routine is available for situations where you want to display the menu bar right away, such as when your application is starting up and you want to see menus as your application initializes and displays a splash screen.

UpdateMenuBar automatically resets a highlighted menu in the menu bar.

MenuBarDisplay

Hide or show the menu bar.

C
```
pascal void MenuBarDisplay (Boolean Show);
```

Pascal
```
procedure MenuBarDisplay (Show: BOOLEAN);
```

Multimedia and presentation applications sometimes need to hide the menu bar and make use of the entire monitor. MenuBarDisplay lets you temporarily hide your application's menu bar while your application is active. As a fail-safe precaution, the menu bar is automatically displayed when your application is suspended. It returns to the state set by your application when your application is activated. Even though the menu bar may be hidden, all command keys still work and command keys for menus will generate doMenu events.

Show indicates if the menu bar is being hidden or displayed. The two constants that can be used for this flag are *on* and *off*.

☞ **Warning**: Some development environments may act up if you try stepping through your program while the menu bar is hidden. Make sure you show the menu bar before quitting your application.

Programming Tips:

1 Hiding or showing a menu bar does not move windows. Your application must move windows if required.

2 The tool bar is automatically repositioned such that it appears below the menu bar if it is displayed, or at the top of the main monitor if the menu bar is hidden.

3 Provide the user with some way to display the menu bar if required.

GetMenuBarColors

Get the default menu colors for your application.

C
```
pascal void GetMenuBarColors (RGBColor *TitleColor, RGBColor *BackColor,
            RGBColor *ItemColor, RGBColor *BarColor);
```

Pascal
```
procedure GetMenuBarColors (var TitleColor: RGBColor;
            var BackColor: RGBColor; var ItemColor: RGBColor;
            var BarColor: RGBColor);
```

GetMenuBarColors obtains the default menu colors used by your application. These defaults can be overridden by setting the colors for specific menus, or specific menu items.

TitleColor is the color of pull-down menus' titles as they appear in the menu bar. The default is black.

BackColor is the background color of pull-down menus and hierarchical menus. The default is white.

ItemColor is the color that is used to display the menu items' text in pull-down menus and hierarchical menus. The default is black.

BarColor is the menu bar's color. The default is white.

SetMenuBarColors

Set the default menu colors for your application.

C
```
pascal void SetMenuBarColors (const RGBColor *TitleColor,
            const RGBColor *BackColor, const RGBColor *ItemColor,
            const RGBColor *BarColor);
```

Pascal
```
procedure SetMenuBarColors (TitleColor, BackColor, ItemColor,
            BarColor: RGBColor);
```

SetMenuBarColors sets the default menu colors used by your application. These defaults can be overridden by setting the colors for specific menus, or specific menu items.

TitleColor is the color of pull-down menus' titles as they appear in the menu bar. The default is black.

BackColor is the background color of pull-down menus and hierarchical menus. The default is white.

ItemColor is the color that is used to display the menu items' text in pull-down menus and hierarchical menus. The default is black.

BarColor is the menu bar's color. The default is white.

GetMenuColors

Get a menu's colors.

C
```
pascal void GetMenuColors (short MenuNumber, RGBColor *TitleColor,
            RGBColor *BackColor, RGBColor *ItemColor);
```

Pascal
```
procedure GetMenuColors (MenuNumber: INTEGER; var TitleColor: RGBColor;
            var BackColor: RGBColor; var ItemColor: RGBColor);
```

GetMenuColors obtains a menu's colors. These settings override colors set by SetMenuBarColors, your application's default menu colors. These settings can be overridden by setting the colors for specific menu items.

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that is being queried. Use mHelpMenu to work with the Help menu, or mAppleMenu to work with the Apple menu. You cannot perform this operation on System 7's (or higher) Applications menu. If the menu does not exist, GetMenuColors returns default colors.

TitleColor is the specified menu's title color as it appears in the menu bar. The default is black.

BackColor is the specified menu's background color. The default is white.

ItemColor is the color that is used to display the menu items' text in the specified menu. The default is black.

SetMenuColors

Set a menu's colors.

C
```
pascal void SetMenuColors (short MenuNumber, const RGBColor *TitleColor,
            const RGBColor *BackColor, const RGBColor *ItemColor);
```

Pascal
```
procedure SetMenuColors (MenuNumber: INTEGER;
            TitleColor, BackColor, ItemColor: RGBColor);
```

SetMenuColors sets a menu's colors. These settings override colors set by SetMenuBarColors, your application's default menu colors. These settings can be overridden by setting the colors for specific menu items.

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that is affected. Use mHelpMenu to work with the Help menu, or mAppleMenu to work with the Apple menu. You cannot perform this operation on System 7's (or higher) Applications menu. If the menu does not exist, SetMenuColors does nothing.

TitleColor specifies the menu's title color as it appears in the menu bar. The default is black.

BackColor specifies the menu's background color. The default is white.

ItemColor specifies the color that is used to display the menu items' text in the menu. The default is black.

GetMenuItemColors

Get a menu item's colors.

C
```
pascal void GetMenuItemColors (short MenuNumber, short ItemNumber,
            RGBColor *MarkColor, RGBColor *ItemColor);
```

Pascal
```
procedure GetMenuItemColors (MenuNumber, ItemNumber: INTEGER;
            var MarkColor: RGBColor; var ItemColor: RGBColor);
```

GetMenuItemColors obtains a menu item's colors. These settings override colors set by SetMenuBarColors, your application's default menu colors, or SetMenuColors, the defaults for a specific menu.

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that is being queried. Use mHelpMenu to work with the Help menu item, or mAppleMenu to work with the Apple menu item. You cannot perform this operation on System 7's (or higher) Applications menu. If the menu does not exist, GetMenuItemColors returns default colors.

ItemNumber specifies the menu item number (from 1 to 32767) that is queried. If the menu item does not exist, GetMenuItemColors returns default colors.

MarkColor is the color that is used to display the menu item's mark character. The default is black.

ItemColor is the color that is used to display the menu item's text. The default is black.

SetMenuItemColors

Set a menu item's colors.

C
```
pascal void SetMenuItemColors (short MenuNumber, short ItemNumber,
            const RGBColor *MarkColor, const RGBColor *ItemColor);
```

Pascal
```
procedure SetMenuItemColors (MenuNumber, ItemNumber: INTEGER;
            MarkColor, ItemColor: RGBColor);
```

SetMenuItemColors sets a menu item's colors. These settings override colors set by SetMenuBarColors, your application's default menu colors, or SetMenuColors, the defaults for a specific menu.

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that is affected. Use mHelpMenu to work with the Help menu item, or mAppleMenu to work with the Apple menu item. You cannot perform this operation on System 7's (or higher) Applications menu. If the menu does not exist, SetMenuItemColors does nothing.

ItemNumber specifies the menu item number (from 1 to 32767) that is affected. If the menu item does not exist, SetMenuItemColors does nothing.

MarkColor is the color that is used to display the menu item's mark character. The default is black.

ItemColor is the color that is used to display the menu item's text. The default is black.

GetMenuString

Get a menu item's text without the metacharacters.

C
```
pascal void GetMenuString (short MenuNumber, short ItemNumber,
            Str255 MenuText);
```

Pascal
```
procedure GetMenuString (MenuNumber, ItemNumber: INTEGER;
            var MenuText: Str255);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) containing the required menu item. Use mHelpMenu to work with the Help menu. You cannot perform this operation on the Apple Menu or System 7's (or higher) Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) from which the text is obtained.

MenuText specifies the menu item's name. If the specified menu number or item number doesn't exist, *MenuText* returns as a null string (length is zero). Note that the string will return as a single space (' ') if a null string was specified when the item was created (this happens automatically to prevent the Menu Manager from crashing).

RenameItem

Rename an existing menu item.

C
```
pascal void RenameItem (short MenuNumber, short ItemNumber,
            const Str255 MenuText);
```

Pascal
```
procedure RenameItem (MenuNumber, ItemNumber: INTEGER; MenuText: STRING);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) in which the menu item is changed. If the menu number does not exist, RenameItem does nothing. Use the mHelpMenu or mAppleMenu constants to work with the Help menu or Apple menu respectively. You cannot perform this operation on the System 7's (or higher) Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) which is changed. If the item number does not exist within the menu specified by MenuNumber, RenameItem does nothing. You cannot change the items installed by the system in the Help menu. You can only rename the "About…" item in the Apple menu.

MenuText specifies the menu item's new name. *MenuText* may be blank, but should never be a null string. The item's state (enabled/disabled), style (bold, underline, etc.), icon and Command key equivalent are not changed. Metacharacters are not interpreted by this routine.

RenameItem does not change the menu's name in the menu bar. If the menu's name must be changed, the affected menu must be removed with the RemoveMenu routine, then re-created as required by using the Menu routine.

EnableMenu

Enable or disable a menu or menu item.

C
```
pascal void EnableMenu (short MenuNumber, short ItemNumber,
          Boolean EnabledFlag);
```

Pascal
```
procedure EnableMenu (MenuNumber, ItemNumber: INTEGER; EnabledFlag: BOOLEAN);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) in which the enabling/disabling will take place. If the menu number does not exist, EnableMenu does nothing.

ItemNumber specifies the menu item number (from 1 to 32767) which is enabled/disabled. A value of zero (0) affects the entire menu. If *ItemNumber* is not zero and the item number does not exist within the menu specified by MenuNumber, EnableMenu does nothing. You cannot disable individual items in the Apple menu or System 7's (or higher) Applications menu, nor can you disable the items installed by the system in the Help menu. In all cases, however, you can disable the entire menu.

EnabledFlag specifies whether the menu/item is enabled or disabled. In the enabled state, the menu/item can be selected. In the disabled state, the menu/item is dimmed and cannot be selected by the user. The two constants that can be used for this purpose are *enabled* and *disabled*. If the *ItemNumber* is zero, the disabling of the menu applies to all items within that menu. When the menu later becomes enabled, all items in the menu assume their correct enabling/disabling as specified by your application. In a disabled pull-down menu, the name in the menu bar is dimmed along with all its items. In a hierarchical menu, all the items in the submenu are dimmed and cannot be selected.

This routine does not immediately update the menu bar if any of the pull-down menus are enabled/disabled by using an ItemNumber equal to zero (0). This is done to prevent the menu bar from flickering each time a menu is enabled or disabled. Instead, all changes to the menu bar appear simultaneously when your application finishes executing an event handler routine. Use UpdateMenuBar if you need the changes to appear right away.

✍ **Note**: The Macintosh's Menu Manager allows only the first 31 items of a menu to be disabled individually. The entire menu, however, can always be disabled.

If a menu item that owns a submenu is disabled, either by disabling the item individually or by disabling the entire menu, the submenu cannot be viewed by the user.

```
CONST                                {Menu constants:                        }
          mFileMenu         =  1;    {File menu number                       }
          mEditMenu         =  2;    {Edit menu number                       }
          mAppleMenu        = -1;    {Apple menu number                      }
          mHelpMenu         = -2;    {Help menu number (System 7.0 or higher)  }
          mApplicationsMenu = -3;    {Applications menu num (Sys 7 or higher)  }
          mDividingLine     = '-';   {Dividing line                          }

          enabled           = true;  {Enable the menu/item                   }
          disabled          = false; {Disable the menu/item                  }
```

CheckMenu

Display or hide a check mark to the left of a menu item.

C
```
pascal void CheckMenu (short MenuNumber, short ItemNumber, Boolean checked);
```

Pascal
```
procedure CheckMenu (MenuNumber, ItemNumber: INTEGER; checked: BOOLEAN);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that is affected. If the menu number does not exist, CheckMenu does nothing. Use mHelpMenu to work with the Help menu. You cannot perform this operation on the Apple Menu or System 7's (or higher) Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) that is affected. If the item number does not exist within the menu specified by *MenuNumber*, or if this item owns a submenu, CheckMenu does nothing. You cannot change the items installed by the system in the Help menu.

Checked specifies whether the menu item's check mark is displayed or hidden. The two constants that can be used for this purpose are *on* and *off*.

To display characters other than the standard check mark, use the MenuMark routine.

```
CONST                           {Menu Item check mark status                }
               on  = true;      {check mark is on                           }
               off = false;     {check mark is off                          }
```

MenuMark

Display or hide a special character to the left of a menu item's name. Use this routine instead of CheckMenu to display or hide characters other than the standard check mark.

C
```
pascal void MenuMark (short MenuNumber, short ItemNumber, char markChar);
```

Pascal
```
procedure MenuMark (MenuNumber, ItemNumber: INTEGER; markChar: CHAR);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that is affected. If the menu number does not exist, MenuMark does nothing. Use mHelpMenu to work with the Help menu. You cannot perform this operation on the Apple Menu or System 7's (or higher) Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) that is affected. If the item number does not exist within the menu specified by *MenuNumber*, or if this item owns a submenu, MenuMark does nothing.

MarkChar specifies the character that is to be displayed. The following constants are available for menu marks:

```
CONST                                     {Menu Item characters                }
               AppleChar   = char($14);   {Apple character                     }
               CheckChar   = char($12);   {Check Mark character                }
               DiamondChar = char($13);   {Diamond character                   }
               DotChar     = char($A5);   {Dot (or bullet) character           }
               NoChar      = char($00);   {No character (remove a character)   }
```

GetMenuMark

Get a menu item's special character that is optionally displayed to the left of an item's name.

C
```
pascal void GetMenuMark (short MenuNumber, short ItemNumber, char *markChar);
```

Pascal
```
procedure GetMenuMark (MenuNumber, ItemNumber: INTEGER; var markChar: CHAR);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that contains the desired menu item. Use mHelpMenu to work with the Help menu. You cannot perform this operation on the Apple Menu or System 7's (or higher) Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) whose mark character is obtained.

MarkChar contains the "mark" character that is displayed to the left of the item's name. If no mark is displayed by the specified menu item, or if the specified menu or menu item doesn't exist, *markChar* is set to null (char(0)). The following are useful constants for testing menu marks:

```
CONST                                     {Menu Item characters                }
               AppleChar   = char($14);   {Apple character                     }
               CheckChar   = char($12);   {Check Mark character                }
               DiamondChar = char($13);   {Diamond character                   }
               DotChar     = char($A5);   {Dot (or bullet) character           }
               NoChar      = char($00);   {No character (remove a character)   }
```

MenuCmd

Set a menu item's Command-key keyboard equivalent.

C
```
pascal void MenuCmd (short MenuNumber, short ItemNumber, char cmdChar);
```

Pascal
```
procedure MenuCmd (MenuNumber, ItemNumber: INTEGER; cmdChar: CHAR);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that is affected. If the menu number does not exist, MenuCmd does nothing. You cannot perform this operation on the Apple Menu or System 7's (or higher) Help menu or Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) that is affected. If the item number does not exist within the menu specified by *MenuNumber*, or if this item owns a submenu, MenuCmd does nothing. You cannot change the items installed by the system in the Help menu.

CmdChar specifies the menu item's Command-key keyboard equivalent. For consistency, use upper case letters (shifted characters are ignored, so ⌘-+ is interpreted as ⌘-=). Do not use Control characters, since older Macintosh keyboards such as those found on the Macintosh Plus don't have a Control key. Tools Plus prevents you from using characters $1B through $1F (Control-[, Control-\, Control-], Control-Up Arrow, and Control--), because these characters are reserved by Apple for the system's use.

Note: A command key cannot be assigned to a menu item if it has a submenu or if it displays an SICN icon. This is a limitation imposed by the Macintosh's Menu Manager.

GetMenuCmd

Get a menu item's Command-key keyboard equivalent.

C
```
pascal void GetMenuCmd (short MenuNumber, short ItemNumber, char *cmdChar);
```

Pascal
```
procedure GetMenuCmd (MenuNumber, ItemNumber: INTEGER; var cmdChar: CHAR);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that contains the desired menu item. You cannot perform this operation on the Apple Menu or System 7's (or higher) Help menu or Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) whose Command-key character is obtained.

CmdChar contains the item's Command-key character. If the menu item doesn't have a Command-key equivalent, or if the specified menu or menu item doesn't exist, cmdChar is set to null (char(0)). This is the case if the menu item owns a submenu.

MenuIcon

Set a menu item's icon.

C
```
pascal void MenuIcon (short MenuNumber, short ItemNumber,
            short IconSelector);
```

Pascal
```
procedure MenuIcon (MenuNumber, ItemNumber, IconSelector: INTEGER);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that is affected. If the menu number does not exist, MenuIcon does nothing. Use mHelpMenu to work with the Help menu. You cannot perform this operation on the Apple Menu or System 7's (or higher) Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) that is affected. If the item number does not exist within the menu specified by *MenuNumber*, MenuIcon does nothing. You cannot change the items installed by the system in the Help menu.

IconSelector identifies the icon that is used, and should be from 1 to 255. The Menu Manager adds 256 to the number you state to specify a resource ID that is in the range of 257 to 511, so if you specify 28, resource ID 284 is used (28 + 256 = 284). These icon resources are read from your application. If Color QuickDraw is available on the Macintosh running your application, a ‘cicn’ (color) icon is used. If a ‘cicn’ is not available (or Color QuickDraw is unavailable), an ‘ICON’ or ‘SICN’ is used. Use zero (0) if you don’t want an icon displayed.

Unlike the equivalent Macintosh toolbox routine, your menu item will remain unaffected if the specified icon can’t be found (i.e., empty space is not reserved in the menu).

Be aware that the Menu Manager drawing a ‘cicn’ icon in color will do so even if the icon was created using 8-bit colors and the monitor is set to 4-bits. This may produce unsatisfactory results. If possible, use 4-bit colors or colors that translate well into 4-bit colors.

Note: Due to limitations imposed by the Macintosh’s Menu Manager, assigning an ‘SICN’ icon to a menu item clears the command key for that item. Also, an ‘SICN’ icon can’t be assigned to a menu item that owns a submenu.

GetMenuIcon

Get a menu item’s icon number.

C

```
pascal void GetMenuIcon (short MenuNumber, short ItemNumber,
             short *IconSelector);
```

Pascal

```
procedure GetMenuIcon (MenuNumber, ItemNumber: INTEGER;
             var IconSelector: INTEGER);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that contains the desired menu item. Use mHelpMenu to work with the Help menu. You cannot perform this operation on the Apple Menu or System 7’s (or higher) Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) whose icon number is obtained.

IconSelector contains the item’s icon number. The Menu Manager automatically adds 256 to the *IconSelector* you specify, so an *IconSelector* of 28 means that resource ID 284 is used (28 + 256 = 284). If an icon is not displayed by the specified menu item, *IconSelector* is equal to zero.

MenuStyle

Set a menu item’s style.

C

```
pascal void MenuStyle (short MenuNumber, short ItemNumber, Style theStyle);
```

Pascal

```
procedure MenuStyle (MenuNumber, ItemNumber: INTEGER; theStyle: Style);
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) that is affected. If the menu number does not exist, MenuStyle does nothing. Use mHelpMenu to work with the Help menu. You cannot perform this operation on the Apple Menu or System 7’s (or higher) Applications menu.

ItemNumber specifies the menu item number (from 1 to 32767) that is affected. If the item number does not exist within the menu specified by *MenuNumber*, MenuStyle does nothing. You cannot change the items installed by the system in the Help menu.

TheStyle specifies the style(s) in which the menu item is to be displayed. Special character constants defined by the Font Manager are bold, italic, underline and shadow. C programmers will use the font manager's constants to specify a composite style, such as MenuStyle(1,1, bold + outline) for bold and outlined, or MenuStyle(1,1,0) for plain text. Pascal programmers will use the font manager's constants to specify a set, such as MenuStyle(1,1,[bold,outline]) for bold and outlined, or MenuStyle(1,1, [ ]) for plain text.

MenuItemCount

Determine the number of items in a menu.

C
```
pascal short MenuItemCount (short MenuNumber);
```

Pascal
```
function MenuItemCount (MenuNumber: INTEGER): INTEGER;
```

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) you wish to query.

The routine's value returns the number of menu items in the specified menu. If the menu number does not exist, the routine returns zero.

```
CONST                                   {Menu constants:                           }
               mFileMenu         =  1;    {File menu number                          }
               mEditMenu         =  2;    {Edit menu number                          }
               mAppleMenu        = -1;    {Apple menu number                         }
               mHelpMenu         = -2;    {Help menu number (System 7.0 or higher)   }
               mApplicationsMenu = -3;    {Applications menu num (Sys 7 or higher)   }
```

GetParentMenu

Determine a menu's "parent" menu number.

C
```
pascal void GetParentMenu (short *MenuNumber, short *ItemNumber,
              short SubMenuNumber);
```

Pascal
```
procedure GetParentMenu (var MenuNumber: INTEGER; var ItemNumber: INTEGER;
              SubMenuNumber: INTEGER);
```

A hierarchical menu can be attached to a pull-down menu or to another hierarchical menu. When attaching a hierarchical menu to a "parent" (or if you prefer to view it the other way, providing a "parent" menu with an "offspring" hierarchical menu), the owner of the submenu is called a "parent" menu, and this routine is used to determine a parent menu and item number for a known submenu.

MenuNumber contains the "parent" menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) which owns the specified submenu. If the specified submenu does not exist, or if it is not attached to a parent menu, *MenuNumber* is set to zero (0).

ItemNumber contains the menu item number (from 1 to 32767) which owns the specified submenu. If the specified submenu does not exist, or if it is not attached to a parent menu, *ItemNumber* is set to zero (0).

SubMenuNumber specifies a hierarchical menu number (from 16 to 200) being queried. If the specified menu does not exist, *MenuNumber* and *ItemNumber* will both return with zero (0) values. You *can* specify a pull-down menu number (from 1 to 15), but its parent menu and item number will always return with a value of zero (0).

GetSubMenu

Determine a menu item's submenu number.

C
```
pascal void GetSubMenu (short MenuNumber, short ItemNumber,
            short *SubMenuNumber);
```

Pascal
```
procedure GetSubMenu (MenuNumber, ItemNumber: INTEGER;
            var SubMenuNumber: INTEGER);
```

A hierarchical menu can be attached to a pull-down menu or to another hierarchical menu. When attaching a hierarchical menu to a "parent" (or if you prefer to view it the other way, providing a "parent" menu with an "offspring" hierarchical menu), the owned hierarchical menu is called a "submenu," and this routine is used to determine a submenu for a known parent menu and item number.

MenuNumber specifies the "parent" menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) being queried.

ItemNumber specifies the menu item number (from 1 to 32767) being queried.

SubMenuNumber returns with a hierarchical menu number (from 16 to 200) which is the submenu for the specified *MenuNumber* and *ItemNumber*. If the menu specified by *MenuNumber* does not exist, or if the specified *ItemNumber* does not exist within the menu, *SubMenuNumber* returns with a value of zero (0). *SubMenuNumber* also returns with a value of zero (0) if the specified menu and item do not own a submenu.

MenuHilite

Highlight a menu in the menu bar, or remove the current highlight from the menu bar.

C
```
pascal void MenuHilite (short MenuNumber);
```

Pascal
```
procedure MenuHilite (MenuNumber: INTEGER);
```

MenuNumber specifies the pull-down menu number (from 1 to 15) that is highlighted. Since only one menu can be highlighted at a time, any other highlighted menu is automatically unhighlighted. Specifying a MenuNumber of zero (0) unhighlights the currently highlighted menu. If the menu number does not exist and is not zero (0), MenuHilite does nothing.

GetMenuHandleFromMemory

Get a handle to a menu that is currently in memory.

C
```
pascal MenuHandle GetMenuHandleFromMemory (short MenuNumber);
```

Pascal
```
function GetMenuHandleFromMemory (MenuNumber: INTEGER): MenuHandle;
```

This routine returns a standard MenuHandle to a pull-down or hierarchical menu that is currently in memory. It does not load the menu from disk if it is not already in memory. You should never need to use this routine. It is provided for advanced programmers who may have specialized needs. Always use Tools Plus routines to create and manipulate menus.

MenuNumber specifies the menu number (from 1 to 15 for pull-down menus, and 16 to 200 for hierarchical menus) you wish to query. Alternatively you can specify one of three menu constants mAppleMenu, mHelpMenu or mApplicationsMenu.

The routine returns a handle to the menu. If the specified menu is not in memory then the routine returns with a value of nil.

```
CONST                                      {Menu constants:                           }
               mAppleMenu        = -1;     {Apple menu number                         }
               mHelpMenu         = -2;     {Help menu number (System 7.0 or higher)   }
               mApplicationsMenu = -3;     {Applications menu num (Sys 7 or higher)   }
```

☞ **Warning**: Do not attach or detach hierarchical menus manually. Always use Tools Plus routines to do this. If you need to lock the handle or change its attributes, do so temporarily then restore the original settings before using any Tools Plus routines. If you alter this handle or any data that is made accessible by this handle, you do so at your own risk.

14 Cursors

Tools Plus gives your application the ability to change the cursor's shape as required. The five standard Macintosh cursors are immediately available for use, though you may create your own custom cursors (including color cursors) by using a resource editor such as Apple's ResEdit. Cursors are identified by a number (0 through 4 are reserved for the standard cursors), and are best referenced by their constants, as indicated below:

arrowCursor iBeamCursor crossCursor plusCursor watchCursor

A cursor's shape can be changed as required by using the CursorShape routine.

Three additional features have been added which make the cursor an integral part of Tools Plus:

- The cursor changes shape automatically depending on its position on the screen and its orientation to the active window(s), including the tool bar and floating palettes.
- Tools Plus recognizes that the watchCursor is an indicator of a lengthy process, and acts accordingly.
- When the watchCursor is displayed, you can have an cursor animation similar to the Finder's spinning wrist watch.

Color Cursors

Color cursors are fully supported regardless if the Macintosh running your application has (or uses) Color QuickDraw or not. If you want to use a color cursor, create the required 'crsr' (color cursors) resources *instead* of 'CURS' (black and white cursor) resources. Each color cursor includes a black and white equivalent that is used by Tools Plus if Color QuickDraw is not available or not used at run time.

You can replace the default arrow and wrist watch with color cursors in your application by including 'crsr' resources with IDs that are identical to those cursors you are replacing. Tools Plus is shipped with replacement color cursors which you may install in your application at your own discretion.

Automatic Cursor Changes

The cursor changes shape automatically only if your application has *not* set the watchCursor. If it has, see "The Watch Cursor" below. Each time that your application finishes executing an event handler routine, the cursor's position is checked and changed if necessary. The change is in accordance to the following rules:

1 If no windows are open, or if a desk accessory is the active window, the cursor is displayed as the standard Macintosh arrow. Desk accessories can manipulate the cursor as they see fit.

When your application has an active standard window, and/or open tool bar and/or floating palettes…

2 While the cursor is outside of the active window(s), it is displayed as the standard Macintosh arrow.

3 While the cursor is in a window's title bar (including the close box and zoom box), or in a documents "size box," it is displayed as the standard Macintosh arrow.

When the cursor is within one of your application's active window's content area…

4 When a cursor is within any editing field, it is displayed as an I-beam.

5 When the cursor is outside all editing fields, then it is displayed as the standard Macintosh arrow providing the window is not using a Cursor Table.

6 If the window is using a Cursor Table, the table's default cursor is displayed when the cursor is not in any of the zones but within the window's content region. When the cursor enters one of the zones, it changes to the shape specified for that zone. See "The Cursor Table" for details.

Tools Plus generates a doMoveCursor event when the cursor moves between cursor zones and between active windows. Most applications will ignore this event. It is needed for situations when you want a “status area” to tell the user what they are pointing at. In such situations, the GetCurrentCursorZone routine can be used to quickly determine the window, cursor table and cursor zone that contains the cursor.

Note: The cursor will not change while a drag is in progress (i.e., pressing and holding the mouse button). This also includes moving a scroll bar’s thumb, or clicking and holding a scroll bar’s up arrow, “page up” region, down arrow, or “page down” region.

The Watch Cursor

The watch cursor is used to indicate a long wait, such as when a lengthy process is being conducted or when printing is being done. Your application can display the watch cursor by using the CursorShape routine.

While the watch cursor is displayed, your application may spend a long time in the event handler routine, and it may not care about getting any events since it doesn’t care what the user is doing and will want to ignore mouse clicks and typing anyway. Unfortunately, this does not give the user the opportunity to halt a lengthy process, nor does it give other applications running under MultiFinder or System 7 or higher any processing time. The watch cursor solves this dilemma by making Tools Plus shift into a *busy* mode, where it filters out (discards) unwanted events.

As soon as the watch cursor is displayed, the caret in an active editing field stops flashing. The watch cursor will *not* be automatically altered as described in *Automatic Cursor Changes* until your application changes the cursor to some other shape. Tools Plus will stop reporting mouse clicks and typing, except ⌘-. which tells your application to halt the process. This lets your application call Process1EventWhileBusy regularly while it is busy conducting a lengthy task, knowing that any key or mouse event it receives is meaningful (i.e., halt the process). Your application may choose not to call Process1EventWhileBusy or to ignore the ⌘-. key if it is busy only for a *short* time.

When your application is finished its lengthy process, it can either change the cursor using CursorShape, or call ResetCursor which changes the cursor to its appropriate shape according to its position on the screen.

Advanced programmers can take “being busy” one step further. While your application is busy, it can display a modal window with a descriptive message and a “Cancel” push button. By setting the WatchCursorButtons routine *on*, the watch cursor will have the ability to click any push button on the active window. Furthermore, if a button has been designated as the “default,” it is activated by pressing the Return or Enter key. A good illustration of this is in a word processing program that has “Pause” and “Cancel” buttons available during printing. When “Pause” is clicked, printing is temporarily suspended and the button changes to “Resume.” Clicking “Resume” continues the printing process and switches the button back to “Pause.” The “Cancel” button has the same affect as typing ⌘-. to halt the process. Note that typing ⌘-. will always be reported by Tools Plus. Remember to switch the WatchCursorButtons routine *off* when you no longer need it, or else the watch cursor will be able to indiscriminately select buttons on any active window.

See the Event Management chapter for details pertaining to the watch cursor and events that are reported while the watch cursor is displayed.

Warning: If your application is running under MultiFinder or System 7 or higher, it is possible to switch to another application while a watch cursor is displayed unless your active window is of type dBoxProc.

Starting your application

When starting your application, you may notice a difference between working in your development environment and your finished (double-clickable) application. THINK Pascal, for example, changes the cursor to a cross-hair, whereas any program launched from The Finder will have a watch cursor.

To Tools Plus, both of these cursors are treated the same way: the cursor is considered to be *undefined*. This is not a problem because Tools Plus will change the cursor’s shape as required. In applications launched by The Finder, this undefined cursor does not behave like a watch cursor to prevent clicking and typing. Your application must explicitly use the CursorShape routine to set the watch cursor to a *true* busy state.

When your application calls InitToolsPlus, the cursor is temporarily set to a watch cursor (without Tools Plus entering a busy mode). As soon as your application calls ProcessEvents, ProcessToolboxEvent, or Process1EventWhileBusy, the cursor is reset to its correct shape according to its position on the monitor.

The Cursor Table

A cursor table is comprised of one or more *cursor zones*, each being a region in a window's local co-ordinates. When the cursor enters a cursor zone, it changes to the shape specified for that zone. Cursor tables are created independently of windows, thereby allowing many windows to use the same cursor table.

A cursor table is created by the NewCursorTable routine. Each cursor table is referenced by a unique number called the *cursor table number*. This number is specified when the cursor table is created, and refers to the specific table until it is deleted. A default cursor shape is specified for the entire table, in case the cursor does not fall into any of the table's zones.

Cursor zones are added to the table by using the CursorZone (CursorZoneRect or CursorZoneRgn) routine. Each zone specifies a region in a window's local co-ordinates, and the type of cursor that is displayed within those co-ordinates. Individual cursor zones are deleted by using the DeleteCursorZone routine. Cursor zones can be added or deleted in any order.

The UseCursorTable routine is used to make a window use a specific cursor table. The window will continue to use the same cursor table until it is closed or the cursor table is deleted. If the cursor table is changed in any way, all windows using that table will incorporate those changes.

When a cursor table is no longer required, it is deleted by the DeleteCursorTable routine.

When the cursor is outside all editing fields, the window's cursor table is checked to see if the cursor is in any of the table's zones (see "Automatic Cursor Changes"). Cursor zones are checked sequentially starting at the lowest numbered zone. If the cursor's "hot spot" is within a zone's region, the cursor's shape changes to the cursor defined for that zone. Otherwise, it is drawn using the table's default cursor.

Any changes made to cursor tables or zones (such as adding, changing, or deleting) which affect the active window will be in effect when your application finishes executing an event handler routine.

Advanced Features

Cursor zones can be made to "overlap" one another with higher numbered zones being placed on top of lower numbered zones (cursor zone numbers ascend from back to front). The highest numbered zone that encloses the cursor is the *containing* zone, even if another lower numbered zone encloses the higher numbered zone. This is useful if you want to have several smaller zones on top of a large one. Zones can also be considered "click sensitive." You can determine if a click occurred in any cursor zone on the current window by using the FindCursorZone routine and handing it the click's location. By doing this, your application can treat icons or pictures as click-sensitive objects (like buttons).

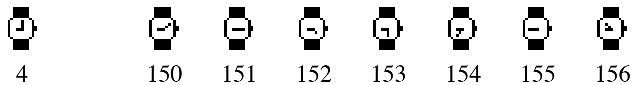
Normally, the cursor changes to the arrow when it is over the window's grow box. You can change this by creating a cursor zone that is exactly the same size as the grow box, that being with the right and bottom side corresponding to the window's right and bottom edge, and the top and left side of the cursor zone being 15 pixels back from the opposite edge.

Note: Cursor tables, and specifically the cursor zones' co-ordinates and shapes are not changed automatically when a window's size is altered. Your application must maintain cursor zones as required. Keep in mind that several windows may be sharing the same cursor table.

Cursor Animation

Tools Plus provides cursor animation, a process where the cursor changes its shape over time. A good example of an animated cursor is System 7's (or higher) Finder. It displays a wrist watch whose minute hand moves clockwise while the Finder copies files. To animate a cursor, you will need to be familiar with a resource editor such as Apple's ResEdit.

First, create the cursors ('crsr' or 'CURS' resources) you need for your animation. The example below uses the system's watch cursor (ID = 4), plus seven custom cursors. When creating new cursors, used IDs 128 or higher.



Second, create an 'acur' resource (shown at right) with an ID of 128 or higher. The 'acur' resource, when included in your application, tells Tools Plus which cursors to use and how quickly they should be animated.

acur ID = 128 from MyResFile

Number of "frames" (cursors): 8
Used a "frame" counter: 45
1) *****
'CURS' Resource Id: 4
2) *****
'CURS' Resource Id: 150
3) *****
'CURS' Resource Id: 151
4) *****
'CURS' Resource Id: 152
5) *****
'CURS' Resource Id: 153
6) *****
'CURS' Resource Id: 154
7) *****
'CURS' Resource Id: 155
8) *****
'CURS' Resource Id: 156
9) *****

Number of "frames" (cursors) specifies the number of cursors you use in a single cycle of the animation sequence.

Used a "frame" counter specifies the time in clock ticks between frames (cursors). A clock tick is 1/60 of a second.

The *'CURS' Resource ID* section is repeated once for each cursor frame in the sequence. It specifies the ID number of the 'crsr' or 'CURS' resource used per frame.

When your application calls CursorShape(watchCursor), the watch cursor is immediately displayed. Any time thereafter when your application finishes executing an event handler routine or calls AnimateCursor, it automatically steps through the animation sequence at the specified frame rate.

If CursorShape(watchCursor) is called again, it manually advances the cursor animation to the next step. If you have an animated cursor that is not a wrist watch (perhaps it is the spinning beach ball used in many applications), your 'acur' resource should exclude the watch cursor.

Tools Plus includes several complete sets of animated cursors (including all cursor resources and the matching 'acur' resource) in the "Optional Resources" folder. You may include any (or all) of these sets in your application. By default, Tools Plus uses the lowest numbered 'acur' resource for cursor animation. You can change the cursor animation sequence at any time with the SetCursorAnimation routine.

Summary:		
	Before you start..........:	Make sure your application contains a valid 'acur' resource
	To start animation ......:	Call CursorShape(watchCursor)
	Animating the cursor .:	Do any of the following -- call AnimateCursor, call CursorShape(watchCursor) to manually step to next cursor frame, call Process1EventWhileBusy, or finish executing an event handler routine.
	New cursor sequence .:	Use SetCursorAnimation
	End cursor animation ..	Use ResetCursor or CursorShape (and don't specify a watch cursor)

☞ **Warning**: Do not modify, delete, or use ReleaseResource on your 'acur' resource or cursor resources while your application is running.

Handling Cursors

The cursor can be set to any shape by calling the CursorShape routine. Tools Plus automatically changes the cursor's shape as described in *Automatic Cursor Changes* using a cursor table if one has been assigned to the active window. Whenever your application sets the watchCursor, Tools Plus filters out unwanted typing and mouse clicks. Filtering continues until your application resets the cursor by using ResetCursor, or changes the cursor by using the CursorShape routine.

CursorShape

Change cursor shape.

The cursor is changed to the shape specified by *CursorType*. The five standard Macintosh cursors defined below are always available to your application. Additional cursors can be used, providing you create these resources and add them to your application. This routine displays the cursor in case it has been hidden by HideCursor or ObscureCursor. CursorShape first tries to find a color cursor ('crsr') resource whose ID matches CursorType. If it can't be found, it looks for a black and white cursor ('CURS') resource. If the specified resource can't be found, the standard arrow cursor is displayed. This operation works properly regardless if the Macintosh running your application has (or uses) Color QuickDraw or not.

When the watch cursor is displayed, Tools Plus shifts into a *busy* mode where it filters out (discards) unwanted events such as the user typing or clicking the mouse. Use ResetCursor to conclude the busy mode, or simply change the cursor to anything other than the watch. There is a section at the beginning of this chapter that details the watch cursor. Also see the WatchCursorButtons routine which disallows or allows a watch cursor to click push buttons.

If your application calls CursorShape(watchCursor) when the watch cursor is already displayed, and your application has an acur resource (for cursor animation), then the cursor changes shape by advancing to the next frame in your cursor animation sequence.

Note: If you specify any cursor other than the watchCursor, it will automatically be reset (as defined in "automatic cursor changes" earlier in this chapter) when your application finishes executing an event handler routine. If you want the window to have a different default cursor, or to display a specific cursor within a certain area, see the section on The Cursor Table earlier in this chapter.

```
CONST                                {Cursor Types                          }
               arrowCursor =0;       {the standard Macintosh arrow          }
               iBeamCursor =1;       {text insertion I-beam                 }
               crossCursor =2;       {cross-hair                            }
               plusCursor  =3;       {select cells in structured documents  }
               watchCursor =4;       {to indicate a long wait               }
```

☞ **Warning**: The InitCursor routine must not be called at any time by your program. To achieve the same affect, use CursorShape(arrowCursor) instead.

Programming Tips:

1 If you want to temporarily display a watch cursor and have it automatically reset to its normal shape, create a watch cursor and give it an ID other than 4 (1004 is a good choice), and create a related constant named TempWatchCursor. When you use CursorShape(TempWatchCursor), a watch is displayed even though Tools Plus does *not* enter its busy mode. As soon as your application finishes executing an event handler routine, the cursor is reset to its normal shape. This is useful at the beginning of a window refreshing routine.

ResetCursor

Reset the cursor to its correct shape according to its orientation to the active window.

C
```
pascal void ResetCursor (void);
```

Pascal
```
procedure ResetCursor;
```

After calling this routine, the cursor's shape automatically changes according to the rules stated in *Automatic Cursor Changes*. Call ResetCursor to negate the effects of CursorShape(watchCursor).

SetCursorAnimation

Set the cursor animation sequence.

C
```
pascal void SetCursorAnimation (short acurResID);
```

Pascal
```
procedure SetCursorAnimation (acurResID: INTEGER);
```

AcurResID specifies the resource ID of the cursor animation ('acur') resource that will be invoked whenever cursor animation is used. If you want to turn off cursor animation (i.e., display only a watch cursor), specify a value of 0 or use Tools Plus's *none* constant.

When Tools Plus is initialized, it finds a default cursor animation sequence (an 'acur' resource). This cursor animation sequence is invoked whenever your application calls CursorShape(watchCursor) then makes subsequent calls to either CursorShape(watchCursor), or AnimateCursor, or when it finishes executing an event handler routine.

The SetCursorAnimation routine lets you start a sequence, change the sequence, or turn off cursor animation at *any* time, even if another sequence is in progress.

AnimateCursor

Force the cursor to animate.

C
```
pascal void AnimateCursor (void);
```

Pascal
```
procedure AnimateCursor;
```

AnimateCursor advances an animated cursor to the next frame providing that the appropriate amount of time has elapsed since the last cursor change. Tools Plus automatically calls AnimateCursor when your application finishes executing an event handler routine, and your application will normally not need to do so.

The only time your application needs to use the AnimateCursor routine is when you want to keep cursor animation running without calling Process1EventWhileBusy, such as during the reading or writing of a file.

NewCursorTable

Create a new cursor table.

C
```
pascal void NewCursorTable (short CursorTable, short CursorType);
```

Pascal
```
procedure NewCursorTable (CursorTable, CursorType: INTEGER);
```

CursorTable specifies the cursor table number (from 1 to 255) that is created. Your application will reference the cursor table by this number. If a cursor table has been previously created using the same number, it is deleted along with its zones. Any windows using the deleted table are reset to not reference a cursor table.

CursorType specifies the default cursor shape for the table. It is the shape the cursor takes when it is within the window's region, but outside editing fields and cursor zones. If watchCursor is specified for this value, Tools Plus changes it to arrowCursor since the watch can only be displayed explicitly by your application.

```
CONST                                {Cursor Types                           }
               arrowCursor =0;       {the standard Macintosh arrow           }
               iBeamCursor =1;       {text insertion I-beam                  }
               crossCursor =2;       {cross-hair                             }
               plusCursor  =3;       {select cells in structured documents   }
```

GetFreeCursorTableNum

Get the first unused cursor table number.

C
```
pascal short GetFreeCursorTableNum (void);
```

Pascal
```
function GetFreeCursorTableNum: INTEGER;
```

Some developers may prefer to write code that more closely resembles a traditional Macintosh application, in that creating an object returns a reference to it such as a handle or pointer. Instead of having to assign your own cursor table number, GetFreeCursorTableNum returns the first unused (free) cursor table number. Using this routine, you can assign an unused cursor table number to a variable, then use that variable throughout your application without concern for the true cursor table number.

If the maximum number of cursor tables has already been created (no new ones can be created), GetFreeCursorTableNum returns a value of zero (0).

DeleteCursorTable

Delete an existing cursor table.

C
```
pascal void DeleteCursorTable (short CursorTable);
```

Pascal
```
procedure DeleteCursorTable (CursorTable: INTEGER);
```

CursorTable specifies the cursor table number (from 1 to 255) that is deleted, along with its associated cursor zones. Any windows using the deleted table are reset such that they don't reference a cursor table. If the cursor table does not exist, DeleteCursorTable does nothing.

CursorZone

Add a new cursor zone to an existing table, or replace an existing zone.

C
```
pascal void CursorZone (short CursorTable, short Zone, short CursorType,
            short left, short top, short right, short bottom);
```

Pascal
```
procedure CursorZone (CursorTable, Zone, CursorType,
            left, top, right, bottom: INTEGER);
```

CursorTable specifies the cursor table number (from 1 to 255) that is to be affected. If the cursor table does not exist, CursorZone does nothing.

Zone specifies the cursor zone number (from 1 to 32767) within the table that is created. If the zone already exists, it is deleted and recreated according to the values of the CursorZone routine. If two or more cursor zones overlap, the highest numbered one is considered to be topmost.

CursorType specifies the cursor shape for the zone. Windows using this table display this type of cursor when the cursor enters this zone. If watchCursor is specified for this value, Tools Plus changes it to arrowCursor since the watch can only be displayed explicitly by your application.

Left, *top*, *right*, and *bottom* define the zone's size and location in a window. If the cursor's location falls within these co-ordinates on the active window, it is said to be within the zone.

Also see: CursorZoneRect.

```
CONST                                 {Cursor Types                          }
            arrowCursor =0;           {the standard Macintosh arrow          }
            iBeamCursor =1;           {text insertion I-beam                 }
            crossCursor =2;           {cross-hair                            }
            plusCursor  =3;           {select cells in structured documents  }
```

CursorZoneRect

Add a new cursor zone to an existing table, or replace an existing zone (co-ordinates specified using a rectangle).

C
```
pascal void CursorZoneRect (short CursorTable, short Zone, short CursorType,
            const Rect *Bounds);
```

Pascal
```
procedure CursorZoneRect (CursorTable, Zone, CursorType: INTEGER;
            Bounds: RECT);
```

CursorZoneRect is identical to the CursorZone routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

CursorZoneRgn

Add a new cursor zone to an existing table, or replace an existing zone (co-ordinates specified using a region).

C
```
pascal void CursorZoneRgn (short CursorTable, short Zone, short CursorType,
            RgnHandle ZoneRgn);
```

Pascal
```
procedure CursorZoneRgn (CursorTable, Zone, CursorType: INTEGER;
            ZoneRgn: RGNHANDLE);
```

CursorZoneRgn is identical to the CursorZone routine, except that it accepts the *ZoneRgn* region handle in place of the individual left, top, right and bottom co-ordinates. CursorZoneRgn makes a copy of the region specified by the *ZoneRgn* handle, so you can dispose or alter the original region without affecting the cursor zone.

GetFreeCursorZoneNum

Get the first unused cursor zone number.

C
```
pascal short GetFreeCursorZoneNum (short CursorTable);
```

Pascal
```
function GetFreeCursorZoneNum (CursorTable: INTEGER): INTEGER;
```

Some developers may prefer to write code that more closely resembles a traditional Macintosh application, in that creating an object returns a reference to it such as a handle or pointer. Instead of having to assign your own cursor zone number, GetFreeCursorZoneNum returns the first unused (free) cursor zone number in a specified cursor table. Using this routine, you can assign an unused cursor zone number to a variable, then use that variable throughout your application without concern for the true cursor zone number.

CursorTable specifies the cursor table number (from 1 to 255) that is to be queried. If the cursor table does not exist, GetFreeCursorZone returns with a value of zero (0).

If the maximum number of cursor zones has already been created in the specified cursor table (no new ones can be created), GetFreeCursorZoneNum returns a value of zero (0).

DeleteCursorZone

Delete an existing cursor zone.

C
```
pascal void DeleteCursorZone (short CursorTable, short Zone);
```

Pascal
```
procedure DeleteCursorZone (CursorTable, Zone: INTEGER);
```

CursorTable specifies the cursor table number (from 1 to 255) that is affected. If the cursor table does not exist, DeleteCursorZone does nothing.

Zone specifies the cursor zone number (from 1 to 32767) that is to be deleted within the table. If the zone does not exist, DeleteCursorZone does nothing.

SetCursorZoneCurs

Change the cursor displayed in a cursor table or cursor zone.

C
```
pascal void SetCursorZoneCurs (short CursorTable, short Zone,
            short CursorType);
```

Pascal
```
procedure SetCursorZoneCurs (CursorTable, Zone, CursorType: INTEGER);
```

CursorTable specifies the cursor table number (from 1 to 255) that is affected. If the cursor table does not exist, SetCursorZoneCurs does nothing.

Zone specifies the affected cursor zone number (from 1 to 32767). If you specify zero (0), the operation applies to the cursor table's default cursor. If the zone does not exist, SetCursorZoneCurs does nothing.

CursorType specifies the cursor shape for the table or zone. It is the shape the cursor takes when it is within the window's region, but outside editing fields and cursor zones. If watchCursor is specified for this value, SetCursorZoneCurs does nothing. The change will becomes visible the next time you call ResetCursor or when your application finishes executing an event handler routine.

GetCursorZone

Get the bounding rectangle for a cursor zone.

C
```
pascal void GetCursorZone (short CursorTable, short Zone, Rect *Bounds);
```

Pascal
```
procedure GetCursorZone (CursorTable, Zone: INTEGER;var Bounds: RECT);
```

CursorTable specifies the cursor table number (from 1 to 255) that is used.

Zone specifies the cursor zone number (from 1 to 32767) within the table being queried.

Bounds returns the cursor zone's bounding rectangle using the window's local co-ordinates. If the specified zone does not exist in the specified table, *Bounds* returns as an empty rectangle with the co-ordinates (0,0,0,0).

GetCursorZoneRgn

Get a handle to a specified cursor zone's region.

C
```
pascal RgnHandle GetCursorZoneRgn (short CursorTable, short Zone);
```

Pascal
```
function GetCursorZoneRgn (CursorTable, Zone: INTEGER): RGNHANDLE;
```

CursorTable specifies the cursor table number (from 1 to 255) that is used.

Zone specifies the cursor zone number (from 1 to 32767) within the table being queried.

The routine's value returns a region handle to the cursor zone's region. If the specified zone does not exist in the specified table, the routine returns nil. Once you have obtained a region handle to a cursor zone, you may perform region operations on it, such as OffsetRgn, InsetRgn, MapRgn, etc. These operations will affect the related cursor zone. When you are finished making all the necessary changes, use ChangedCursorZone to inform Tools Plus that the cursor's shape *may* have to be recalculated.

Note: Use this routine only to get a handle to a cursor zone's region. Do not initialize the receiving region handle variable by using NewRgn (just create a variable that is a RgnHandle type). When you have finished using the handle, do not use DisposeRgn.

ChangedCursorZone

Indicate that one or more cursor zone regions have been manually altered.

C
```
pascal void ChangedCursorZone (void);
```

Pascal
```
procedure ChangedCursorZone;
```

If any cursor zone's regions have been altered (by obtaining the cursor zone's region handle and altering the region), you must call ChangedCursorZone. This routine informs Tools Plus that the cursor's shape *may* have to be recalculated because of cursor zone changes. Next time your application processes and event or when it finishes executing an event handler routine, the cursor's shape is updated (if necessary).

UseCursorTable

Make the current window use a cursor table, or stop it from using any cursor table.

C
```
pascal void UseCursorTable (short CursorTable);
```

Pascal
```
procedure UseCursorTable (CursorTable: INTEGER);
```

CursorTable specifies the cursor table number (from 1 to 255) that is used by the current window. If the current window does not belong to your application, if no windows are open, or if the cursor table does not exist, UseCursorTable does nothing. If CursorTable has a value of 0, the current window will stop using its present cursor table. The table, however, remains unaffected.

FindCursorZone

Determine which cursor zone contains a specified point in the current window.

C
```
pascal short FindCursorZone (Point thePoint);
```

Pascal
```
function FindCursorZone (thePoint: POINT): INTEGER;
```

ThePoint is a location in the current window's local co-ordinates.

The routine's value returns the cursor zone that contains the specified point. If the current window does not belong to your application, or if no windows are open, or if the current window does not use a cursor table, the routine returns a value of zero (0). Zero will also be returned if the point is not contained in any cursor zone in the window's cursor table.

In situations where cursor zones overlap one another and the specified point is in the intersecting region of multiple zones, FindCursorZone returns the highest numbered zone which is equivalent to the topmost of the overlapping zones.

This routine ignores the placement of buttons, scroll bars, edit fields, list boxes, custom controls, etc. It reports strictly on the relationship between the provided point and the current window's cursor table. If you obtained thePoint as a "mouse down" event from Tools Plus's doClick event, you can be sure that the click did not occur in a button, scroll bar, edit field, list box or custom control.

GetCurrentCursorZone

Determine which window, cursor table and cursor zone contains the cursor.

C
```
pascal void GetCurrentCursorZone (short *Window, short *CursorTable,
            short *Zone);
```

Pascal
```
procedure GetCurrentCursorZone (var Window: INTEGER;
            var CursorTable: INTEGER; var Zone: INTEGER);
```

Most applications never need to use this routine. GetCurrentCursorZone is typically used after a doMoveCursor event informs your application that the cursor has moved into a new cursor zone. A good example is a situation when an application has a "status line" that tells the user something about the object they are pointing at. Each object is enclosed by a cursor zone, so when the user moves the cursor over an object, a doMoveCursor event is generated and your application can determine exactly which window and cursor zone contains the cursor. The values returned by this routine are always available to your application, and they are updated each time your application finishes executing an event handler routine.

Window indicates the active window number containing the cursor. The cursor can be over any part of the window including the title bar or grow box. A value of zero indicates the cursor is not over an active Tools Plus window.

CursorTable returns the cursor table number used by the window containing the cursor. If the cursor is anywhere in the window's content region and the window uses a cursor table, the table's value is returned. Otherwise zero (0) is returned.

Zone returns the cursor zone number containing the cursor. If the cursor is not within a cursor zone, zero (0) is returned. If your cursor zone encloses an editing field and the cursor is inside the field, the cursor zone number is still returned.

WatchCursorButtons

Allow or disallow buttons to be clicked when the cursor is a wrist watch.

C
```
pascal void WatchCursorButtons (Boolean Allowed);
```

Pascal
```
procedure WatchCursorButtons (Allowed: BOOLEAN);
```

Allowed specifies if buttons can be clicked on the active window while the wrist-watch cursor is displayed. With a value of *true*, push buttons can be clicked. With a value of *false*, they cannot. WatchCursorButtons is set to *false* when Tools Plus is initialized.

This feature is best used when a modal window is displayed with one or more push buttons while the Macintosh is busy with some lengthy process. Turn WatchCursorButtons *on*, then continue continue normally during the process. This lets the user click a “Cancel” button, or to press ⌘-. to halt the process. Remember to turn WatchCursorButtons back *off* when you no longer need it, or else the watch cursor will be able to click any push buttons indiscriminately.

```
CONST                                     {click push buttons with watchCursor?     }
               on       = true;           {allow push buttons to be clicked when    }
               enabled  = true;           {  the watchCursor is displayed.          }
               off      = false;          {do not allow push buttons to be clicked  }
               disabled = false;          {  when the watchCursor is displayed.     }
```

15 Balloon Help

Tools Plus implements the Macintosh's Help Manager by automating a number of its services, and by making it much easier for you to implement those services. Tools Plus also integrates the Help Manager into Tools Plus's model of specialized objects, such as buttons, list boxes, sliders, editing fields, and so on. This lets you apply Help to specific objects instead of setting Help for areas on a dialog. Even though Tools Plus makes it easy to add Help to your application, the Help Manager has some design constraints that are inherited by Tools Plus. These constraints are detailed in this chapter.

Within this chapter, you will see the word "help" (no capitalization) which is used to indicate assistance of some sort. When you see the word "Help" (capitalized), it is an abbreviation for the proper name Balloon Help, and refers specifically to Apple's Balloon Help services and resources.

The most common approach to adding help to an application or plug-in is to first create the app, then to add Balloon Help later. With Tools Plus's implementation of Balloon Help, this is usually the best approach, although you should still devote appropriate thought to the wording in each Help item.

Before you continue with this chapter, please note:

- You should already be familiar with the Help Manager as documented in Inside Macintosh. This will teach you the basic principles of Balloon Help, and it will give you an understanding of how your Help messages are related to specific objects. It also details various resources that are used for Balloon Help. This chapter refers to those resources, but does not detail their structure or contents. If you are a moderate programmer or better and have not read about Balloon Help in Inside Macintosh, you can probably get away with just reading this chapter.
- You will almost certainly need to use a powerful resource editor to create the required suite of Help resources for your application or plug-in. We recommend that you use Resorcerer, or an equally powerful editor. Lesser resource editors such as ResEdit, may not be able to edit the necessary Help resources, thereby not allowing you to implement most of the Help Manager's functionality.
- The Help Manager was implemented by Apple as being almost exclusively resource driven. Consequently, if you've been avoiding resources until now, then you will either have to start using a powerful resource editor, or forgo having Balloon Help for pull-down menus, hierarchical menus, pop-up menu lists, the application icon, and window parts outside of the content region. Without the use of resources, you will only be able to assign Help to user interface elements in windows.

Tools Plus automatically associates Help data (the information that appears in a Help balloon as well as details about its appearance and behavior) with a specific user interface element. That way, if you ever show, hide, move, resize, delete, or scroll a user interface element, Tools Plus makes sure that the related Balloon Help data and "hot rectangle" follows the object. When you are relating this chapter to Inside Macintosh, Tools Plus incorporates the more advanced "Dynamic Window" model for Help.

As you create your Help resources ('hdlg', 'hrct' and 'hmnu'), realize that Tools Plus ignores the tip location and "hot rectangle" specified in the resource because Tools Plus calculates these two items in real time when the user points to an object. Tools Plus automatically takes care of showing and hiding Help balloons depending on whether the user has selected the showing or hiding of balloons in the Help menu. Tools Plus also takes care of displaying Help balloons as the user moves the cursor over various objects, clicks in a menu, and points at various menu items without selecting them. All you need to do is assign the right Help data to various objects, and Tools Plus takes care of showing the Help balloons.

Help Inheritance

Tools Plus includes the optional ability for user interface elements to inherit Help messages from their parent object, or the object behind them. This works with the Appearance Manager's control embedding, as well as with standard Tools Plus user interface elements. The following examples show you how Help Inheritance works:

- The mouse if moved over a radio button control that has no Help messages. Tools Plus knows the radio button is embedded in a group box control that has Help messages, so it uses those messages for the radio button as well.
- The developer has an identical Help disabled message for all user interface elements embedded in a Placard control. Each of the embedded elements does not include a disabled Help message, but the Placard does. When the mouse is over any disabled element in the Placard control, Help is displayed using the Placard control's disabled Help message.

Help Inheritance is off by default. You turn it on by including the initInheritHelp option in the InitToolsPlus routine's spec parameter when you initialize Tools Plus. By default, Tools Plus determines which Help message to use by checking the object under the mouse in the order listed below. Once it finds an object, it stops searching and uses that object's Help message. When you turn Help Inheritance on, Tools Plus keeps searching down the list (detailed below) until it finds an object under the mouse that has the required Help message.

- Control (including Edit Text, Static Text, List Box, and Pop-Up Menu control)
- Parent control if the Appearance Manager is available (searching stops at the ultimate parent control, excluding the window's root control)
- TextEdit field (editable or static)
- List Manager list box
- Tools Plus pop-up menu (not using a control)
- Tools Plus picture button
- Tools Plus panel
- Tools Plus cursor zone
- Tools Plus cursor table

Balloon Help for the Finder ('hfdr' resource)

The 'hfdr' resource is used exclusively by the Finder to describe your application when the user point to it. This service is provided entirely by the Finder and the Help Manager, so there are no special Tools Plus services associated with this element. If you create an 'hfdr' resource, assign it resource ID -5696.

Balloon Help for Menus ('hmnu' resource)

Balloon Help for menus is provided exclusively through the 'hmnu' resource (this is a Macintosh toolbox requirement, and not specific to Tools Plus). Each 'hmnu' resource relates to a single 'MENU' resource with an identical resource ID. When you call any Tools Plus routine that loads a 'MENU' resource, Tools Plus also loads the related 'hmnu' Help resource and attaches it to the menu. This applies to all menus: pull-down, hierarchical, and pop-up menus. When you delete the menu, Tools Plus automatically detaches the related Balloon Help, and correctly disposes the Help data.

Balloon Help for Objects in Windows

Help can be assigned to user interface elements in a window in three different ways:

- Use an 'hdlg' or 'hrct' Help resource with your dialog ('DLOG') or dialog item list ('DITL') resource to have Help automatically assigned to the window's objects.
- Programatically assign help to a single user interface element using an 'hdlg', 'hrct' and/or 'hmnu' Help resources.
- Programatically assign help to a single user interface element without using a Help resource.

Using ‘hdlg’ and/or ‘hrct’ Resources in Dialogs or Dialog Lists

This approach most closely resembles standard Macintosh Help Manager support. It lets you create standard Macintosh Help resources, and automatically attach their Help data to user interface elements when you open your dialog using Tools Plus’s LoadDialog routine or equivalent. Similarly, you can have a Help resource that parallels a dialog item list (‘DITL’ resource), so when you load or append that dialog item list by using Tools Plus’s AppendDialogList routine, Tools Plus automatically copies that Help resource’s Help data to the user interface elements as they are being created.

The ‘hdlg’ Dialog Help resource contains Help data for four object states:
- Enabled and not checked (value = 0)
- Disabled (checked or unchecked)
- Enabled and checked (value = 1)
- Enabled, other (value is not 0 or 1)

Objects like buttons, scroll bars, and picture buttons have associated values, so they can potentially display all four Help messages at some time. Other objects like Tools Plus Panels and Cursor Zones have only a single state (enabled). When objects are disabled on an inactive window, Balloon Help does not explain individual objects, but rather, it tells the user that the window is inactive, and that it can be activated by clicking on it.

When you create an ‘hdlg’ Dialog Help resource, set its resource ID to be the same as the related ‘DLOG’ resource or ‘DITL’ resource. Each item in the ‘hdlg’ resource corresponds to the same item number in the dialog (i.e., first item in ‘hdlg’ provides Help for the first item in the dialog). Remember that each Help item’s point and “hot rectangle” in the ‘hdlg’ resource are ignored because Tools Plus calculates this information dynamically as required.

Alternatively, you can use an ‘hrct’ Help Rectangle resource. Unlike the ‘hdlg’ resource, each item in the Help Rectangle resource has a message for only a single state. This is ideal for providing help for Cursor Zones and Panels if required. When you load a dialog or append a dialog item list, Tools Plus first looks for Help data in an ‘hdlg’ resource. If it does not find the required help for the object, Tools Plus then tries to find the required Help data in an ‘hrct’ resource. If you are confused as to which resource you should use, use the more robust ‘hdlg’ resource.

The following is a list of all dialog items, and the various Help states they can support. Notice that some dialog items to not translate into Tools Plus “objects,” and therefore, Help is not assigned to them.

	Supported States			
‘DITL’ Item	**Enabled**	**Disabled**	**Checked**	**Other**
Button, or any control that is implemented as a button	✔	✔	✔	✔
Scroll Bar, or any control that is implemented as a scroll bar	✔	✔	✔	✔
Static Text (srcCopy)	✔	✔	✔	✔
Static Text (other than srcCopy)				
Edit Text	✔	✔		
List Box (implemented using a CNTL)	✔	✔		
Pop-Up Menu (implemented using a CNTL)	✔	✔		
Icon				
Picture				
User Item				

You will notice in the table above, that Tools Plus does not automatically assign Help data to icons, pictures and user items in a dialog or dialog item list. This is because these items have no direct equivalent to a Tools Plus object, and are therefore incapable of “owning” Help data. The easiest work around is to create Icon controls and Picture controls using the Appearance Manager (see the Buttons chapter for details). You can also use Tools Plus routines to create a cursor zone for any of these objects, and attach Help data to the cursor zone. The user will experience the same result

in that they can point to an icon and a Help balloon will pop up.

Manually Assigning Help Resources Data to a User Interface Element

You can assign Help to any user interface object that you create in a window using a Tools Plus routine. In each case, when you create a user interface object, you'll be able to refer to it later by an object number. For example, panels and cursor zones each have their own unique number that you use when you want to move, resize, or delete that object later. It is critically important that you distinguish these types of objects from Tools Plus routines that simply draw something, such as the DrawPict routine which draws a picture and the DrawIcon routine that draws an icon. Neither of these routines create an object that can be referenced later. They simply draw something (a picture or an icon) when you call them. You cannot associate Help with the image they create (although you can associate Help with a cursor zone that overlays that image).

A number of Tools Plus routines are available to let you associate the Help data in a standard Help resource with a user interface element. For example, the SetButtonHelpRes routine lets you associate Help data in a Help resource with a button, or any control that is implemented as a button. Similar routines are available for scroll bars, picture buttons, list boxes, and all other user interface elements. These routines can read any or all of the following Help resources: 'hdlg' (dialog or dialog item list), 'hrct' (rectangle), and 'hmnu' (menu). You specify a resource ID, and index (i.e., the first item in the Help resource is indexed as 1, the second item is indexed as 2, and so on), and you specify which of the resources you want to access, and for which object state.

Use the following constants to access any or all of the three Help resources:

```
helpUseHdlgRsrc = $00000001;           {Reference 'hdlg' resource to get Help info    }
helpUseHrctRsrc = $00000002;           {Reference 'hrct' resource to get Help info    }
helpUseHmnuRsrc = $00000004;           {Reference 'hmnu' resource to get Help info    }
helpUseAllRsrc  = $0000000F;           {Reference all resources to get Help info      }
```

The Tools Plus routines access the resource(s) specified above that have the required resource ID, and searches for Help data for any of the following states:

```
helpEnabledState        = $00000100;   {Get Help for 'enabled' state                  }
helpDisabledState       = $00000200;   {Get Help for 'disabled' state                 }
helpEnabledCheckedState = $00000400;   {Get Help for 'enabled and checked' state      }
helpEnabledOtherState   = $00000800;   {Get Help for 'enabled, other' state           }
helpAllStates           = $00000F00;   {Get Help info for all object states           }
```

The inner functioning of the routines are as follows:

1. Reference the first specified resource type (in the order of 'hdlg', 'hrct', 'hmnu') whose resource ID matches the one you specify in the Tools Plus routine.
2. Search for the first required Help data state (in the order of Enabled, Disabled, Enabled and Checked, Enabled Other).
3. If that Help data exists, take either a copy of that Help data, or a reference to that Help data (which ever is more memory efficient) and store it as part of the user interface object such as a button or scroll bar.
4. If the required Help data does not exist for that state, or if you did not ask for Help data for that state, then advance to the next state until all your specified states have been tested or obtained.
5. If Help data has not been obtained for all the Help states you specified, the routine then tries the next Help resource. It skips Help data for the states that it obtained in previously.

The table below lists all the user interface elements that you can create with Tools Plus, and which Help states they support. Note that a user interface element is an object that can later be referenced by unique number, such as a button (you can later delete that button by calling Tools Plus's DeleteButton routine and passing the button number as a parameter). Images that are drawn in a window, such as an icon or a picture, can not directly support Help. You can indirectly associate Help with these items by using a cursor zone and associating Help to the cursor zone. If certain Help states are not supported in the table below, do not bother creating Help messages for those states because those messages will never be displayed.

User Interface Element	Supported States			
	Enabled	**Disabled**	**Checked**	**Other**
Button, or any control that is implemented as a button	✔	✔	✔	✔
Picture button	✔	✔	✔	✔
Scroll Bar, or any control that is implemented as a scroll bar	✔	✔	✔	✔
Field (any kind)	✔	✔		
List Box	✔	✔		
Pop-Up Menu	✔	✔		
Panel	✔			
Cursor Table	✔			
Cursor Zone	✔			

Manually Assigning Help Data Without Using Resources

In some cases, you may want to assign Help data to a user interface element without first having to define that data in a Help resource. Tools Plus provides a routine for each user interface element, such as SetButtonHelp for a button, that lets you assign any kind of Help data to that object. The routine requires a number of parameters that ultimately give you absolute control over an object's Help data and how it is displayed. When you use this kind of routine, you set the Help data for a single object one state at a time (i.e., once for the enabled state, once for the disabled state, and so on).

The following parameters are required to set an object's Help data for a single state:

- Object Number: The unique number that identifies the button, scroll bar, or other object.
- State: Enabled, disabled, enabled and checked, enabled other (see constants defined earlier)
- Tip Proc: A UPP to the routine that is used to determine where the balloon's tip appears (nil = default). Inside Macintosh details how to write your own Balloon Tip Proc. In pure 680x0 code, this is just the address to the routine. In code that compiles to a PowerPC executable and optionally 680x0, use the toolbox's NewTipFunctionProc routine to convert your Balloon Tip Proc to a UPP.
- Window Resource ID: The 'WDEF' resource ID for the balloon window (0 = default)
- Window Variant: The balloon window's variant code (0 = default)
- Method: The method for displaying a Help balloon (0 = default)…

  ```
  kHMRegularWindow    = 0;  {Create a regular window floating above all windows        }
  kHMSaveBitsNoWindow = 1;  {Save the image behind the balloon and draw (like a menu)  }
  kHMSaveBitsWindow   = 2;  {Save the image behind the balloon + generate update event }
  ```

  Do not mix different methods within the same window or you will get improper window refreshing.
- Message: A Help Manager Message Record (detailed below)

The Help Manager Message Record is a variant record that is used to specify the message that is displayed by the Help balloon. This message can contain anything from a simple Pascal string, to reference to an STR# resource or a handle to a picture that can be created or obtained in real time. The Help Manager Message Record is defined in Apple's Balloons.h C/C++ header and Balloons.p Pascal interface file as follows:

C

```
struct HMStringResType {                    // Record for accessing 'STR#' resources…
   short    hmmResID;                       // STR# resource ID
   short    hmmIndex;                       // String number in 'STR#' resource
};
typedef struct HMStringResType HMStringResType;

struct HMMessageRecord {                    // Balloon Help Message Record…
   short              hmmHelpType;          // Type of data (khmmString, khmmPict, etc.)
   union {
      Str255            hmmString;          // Pascal string
      short             hmmPict;            // 'PICT' resource ID
      TEHandle          hmmTEHandle;        // Handle to a TextEdit record
      HMStringResType   hmmStringRes;       // Reference to a 'STR#' resource's string
      PicHandle         hmmPictHandle;      // Handle to a picture
      short             hmmTERes;           // 'styl'/'TEXT' resource ID
      short             hmmSTRRes;          // 'STR ' resource ID
   } u;
};
```

Pascal

```
HMStringResType = record                    {Record for accessing 'STR#' resources…        }
   hmmResID:            integer;            {STR# resource ID                               }
   hmmIndex:            integer;            {String number in 'STR#' resource               }
end;

HMMessageRecord = record                    {Balloon Help Message Record…                   }
   hmmHelpType:         integer;            {Type of data (khmmString, khmmPict, etc.)      }
   case integer of                          {                                               }
      khmmString: (                         {                                               }
         hmmString:     Str255;             {Pascal string                                  }
      );                                    {                                               }
      khmmPict: (                           {                                               }
         hmmPict:       integer;            {'PICT' resource ID                             }
      );                                    {                                               }
      khmmTEHandle: (                       {                                               }
         hmmTEHandle:   TEHandle;           {Handle to a TextEdit record                    }
      );                                    {                                               }
      khmmStringRes: (                      {                                               }
         hmmStringRes:  HMStringResType;    {Reference to a 'STR#' resource's string        }
      );                                    {                                               }
      khmmPictHandle: (                     {                                               }
         hmmPictHandle: PicHandle;          {Handle to a picture                            }
      );                                    {                                               }
      khmmTERes: (                          {                                               }
         hmmTERes:      integer;            {'styl'/'TEXT' resource ID                      }
      );                                    {                                               }
      khmmSTRRes: (                         {                                               }
         hmmSTRRes:     integer;            {'STR ' resource ID                             }
      );                                    {                                               }
end;                                        {                                               }
```

When you populate the HMMessageRecord record, the hmmHelpType field is used to indicate what type of data is being stored in the record. The following constants are defined in Apple's Balloons.h C/C++ header and Balloons.p Pascal interface file for this purpose:

```
khmmString       1;   {Help message contains a Pascal String                            }
khmmPict         2;   {Help message contains a resource ID to a 'PICT' resource         }
khmmStringRes    3;   {Help message contains a resource ID & index to an 'STR#' resource }
khmmTEHandle     4;   {Help message contains a Text Edit handle                         }
khmmPictHandle   5;   {Help message contains a Picture handle                           }
khmmTERes        6;   {Help message contains a resource ID to 'TEXT' & 'styl' resources  }
khmmSTRRes       7;   {Help message contains a resource ID to an 'STR ' resource        }
```

If you do a lot of programming in which you manually set an object's Help data, you may want to write your own specialized routine that has very few parameters (button number, object state, Pascal Help string), and internally, it populates the required fields in the HMMessageRecord record, then calls the required Tools Plus routine such as SetButtonHelp.

'hdlg' and 'hmnu' Resource Settings

The first Help entry in a dialog list help ('hdlg') resource and a menu help ('hmnu') resource is for missing messages. A "missing message" occurs when the Help resource states that it has Help data for a specific item number, but it is missing the Help message for a specific state for that item. For example, if item #8 in the Help resource contains Pascal strings, and no string is supplied for the disabled state, then the "missing message" data is used for the disabled state. This is useful if you have a default message that is applicable to many items, and you do not want to repeat entering that message.

When using an 'hdlg' resource, realize that you can change the starting point from which Tools Plus reads Help messages. The "Item offset for first message" field in the 'hdlg' resource has a default value of 0, meaning that item number 0, or the "missing message" data, is considered the first readable item. This data will be used for the first item in the dialog item list if you have an 'hdlg' Help resource for a dialog.

We recommend that you change the value of the "Item offset for first message" field in the 'hdlg' resource to 1 so that anything accessing the 'hdlg' resource will initially ignore the "missing message" data, and look to the next entry in the resource as the first Help message.

Efficiently Storing Numerous Help Messages

All Help resources ('hmnu', 'hdlg' and 'hmnu') have the ability to refer another resource for Help data rather than storing the data in the resource itself. For example, instead of storing the same Help data for 40 editing fields as 40 separate Pascal strings in an 'hdlg' resource, each with the same text, it is wiser to have each of the 40 items simply reference a single 'STR ' resource that contains that string. In this case, the "Message Type" in the Help resource would be "Use 'STR ' resource=7", and you would provide an 'STR ' resource ID number where the string can be found. The disadvantage of this strategy is that you may end up creating a lot of 'STR ' resources, and you may end up exceeding the Mac OS limit of approximately 2700 individual resources in a single file.

Another strategy that is more effective in reducing the number and volume of resources in your application, is to use an 'STR#' indexed string resource to store Help messages that are repeated. A single 'STR#' resource can store up to 32,737 Pascal strings. You have a single occurrence of the repeating string in the 'STR#' resource, and the 'hdlg' resource simply references that 'STR#' entry. The disadvantage of using an 'STR#' resource is that the entire resource must be loaded into memory even though the caller only wants to extract a single string. You can easily account for this by flagging the 'STR#' string as preloaded and non-purgeable so you won't be surprised by a sudden consumption of memory as the resource is loaded because the resource will always reside in memory.

Balloon Help Performance Issues

When the user turns Balloon Help on (by selecting the Help menu's Show Balloons item), your application will experience a periodic performance decrease while Tools Plus searches for the appropriate Help message and displays a Help balloon. This is normal behavior. The performance decrease takes place each time that Tools Plus needs to check the mouse's location, determine which object is under the mouse, build optimizing data structures, search for the Help message, and finally display the Help balloon. Tools Plus takes every possible precaution to minimize the frequency of these occurrences as well as their duration to make your application run as quickly as possible while still providing Balloon Help services. When the user turns Balloon Help off, Tools Plus resumes its maximum performance.

For those developers who strive for peak performance while Balloon Help is on, the following items detail the factors that influence performance:

- Where ever it is possible, use Appearance Manager controls in place of Tools Plus equivalents (i.e., use a Group Box control instead of a Tools Plus Panel, use an Edit Text control in place of a standard Tools Plus editing field, use a Static Text control in place of a standard Tools Plus read-only field). This lets Tools Plus identify the object

requiring Help a little faster.

- Provide Help for all objects in a window rather than using Help inheritance. This lets Tools Plus find the appropriate Help message a little faster.
- The "worst case scenario" is when Tools Plus spends the most time looking for Help in a window that has many objects and the cursor is not over any item. The more complex the window, the longer it takes. Even so, with Tools Plus's optimization, the impact is minimized.

When Balloon Help is on, the following actions will cause Tools Plus to recalculate the Help message and display a Help balloon if a message is available:

- Suspend or resume your application
- Open, close, move or resize a window
- Activate an inactive window
- Create, delete, show, hide or obscure a user interface element
- Resize, move, or offset a user interface element
- Enable or disable a user interface element
- Change a user interface element's value. Tools Plus has special optimization that quickly determines if a Help state change resulted from the value change, and recalculates the Help message and redisplays the Help balloon only if the state has changed. With an enabled scroll bar, for example, a Help state of "enabled" is used when the scroll bar's value is 0. When it's value is 1, a Help state of "enabled and checked" is used. For all other values, a Help state of "enabled, other" is used. Therefore, as the scroll bar's value progresses from 2 through 32767, Tools Plus uses the "enabled, other" Help message and it does not need to recalculate Help.

Issues with THINK Pascal

If you are working in the THINK Pascal development environment (not when you run a double-clickable application that was created with THINK Pascal), you will experience several bugs that are related to THINK Pascal’s interaction with the Help Manager:

- When you turn Balloon Help off (Help Menu: Hide Balloons) or on (Help Menu: Show Balloons), you will experience a delay of several seconds before your application responds to events. These events are queued, but THINK Pascal becomes “dormant” for a short time. If you want immediate response, the work-around is to simply wait a few seconds before typing or clicking your mouse.
- If you switch between having Balloon Help on and off, and you quit your application without quitting THINK Pascal (i.e., you are running your application within THINK Pascal’s development environment), the next time you run your application, you will notice that your Apple Menu contains multiple copies of its items. This is compounded each time your run your application inside the THINK Pascal development environment. You may eventually crash THINK Pascal while it is running or when you quit THINK Pascal. To resolve this problem, simply quit THINK Pascal and restart it again. A better work-around is to leave Balloon Help on all the time while you are developing and testing Help.
- When running your application in the THINK Pascal development environment, Balloon Help may not be displayed for menu titles in the menu bar (i.e., Apple, File, Edit, etc.) There is no known work-around for this.

These issues do not appear in any other development environment that is upported by Tools Plus, or in double-clickable applications you create with THINK Pascal.

SetButtonHelp

Set Help data for a button without using Help resources.

C
```
pascal void SetButtonHelp (short Button, short State,
             TipFunctionUPP TipProc, short WinResID, short Variant,
             short Method, const HMMessageRecord *HelpMessage);
```

Pascal
```
procedure SetButtonHelp (Button, State: INTEGER; TipProc: TipFunctionUPP;
             WinResID, Variant, Method: INTEGER; HelpMessage: HMMessageRecord);
```

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, SetButtonHelp does nothing.

State is the single object state for which you are setting the Help data. If the button already has Help data for that state, the old data is overwritten with the new data. If you want to set Help data for multiple states, use the SetButtonHelp routine once for each state. Use one of the following constants as defined in Apple's C/C++ Balloons.h header and in the Pascal Balloons.p interface file:

`kHMEnabledItem`	Item is enabled but not checked (control value = 0).
`kHMDisabledItem`	Item is disabled.
`kHMCheckedItem`	Item is enabled and checked (control value = 1).
`kHMOtherItem`	Item is enabled (control value > 1).

TipProc is a UPP to the routine that is used to determine where the balloon's tip appears (nil = default). Inside Macintosh details how to write your own Balloon Tip Proc. In pure 680x0 code, this is just the address to the routine. In code that compiles to a PowerPC executable and optionally 680x0, use the toolbox's NewTipFunctionProc routine to convert your Balloon Tip Proc to a UPP.

WinResID is the window ('WDEF') resource ID that is used for the balloon's window (0 = default).

Variant is the variant code for the balloon's window (0 = default).

Method describes how the Help balloon is drawn, and even more important, how your application refreshes a window when the balloon is removed. Use one of the following constants as defined in Apple's C/C++ Balloons.h header and in the Pascal Balloons.p interface file:

`kHMRegularWindow`	Create a regular window floating above all windows. When the balloon window is removed, your application gets doPreRefresh and doRefresh events for the region that used to be occupied by the balloon (0 = default).
`kHMSaveBitsNoWindow`	Save the image behind the balloon before drawing the balloon. When the balloon is removed, the image behind it is quickly restored, like with a pull-down menu. No refreshing events are generated. This method is very quick, but it can only be used in windows where the elements cannot change while the balloon is displayed.
`kHMSaveBitsWindow`	Save the image behind the balloon before drawing the balloon. When the balloon is removed, the image behind it is quickly restored, like with a pull-down menu, then your application gets doPreRefresh and doRefresh events for the region that used to be occupied by the balloon. This method is very quick at replacing the image behind the balloon, but it can only be used in windows where the elements cannot change while the balloon is displayed.

HelpMessage is the Balloon Help Message Record that contains the data that is attached to the button. You need to populate this record's hmmHelpType to tell it the kind of data it contains, then populate a second field with the specified data. You can find details on how to do this earlier in this chapter.

```
CONST                                   {Object States:                                    }
               kHMEnabledItem  = 0;     {Item is enabled, but not checked or control value = 0    }
               kHMDisabledItem = 1;     {Item is disabled                                  }
               kHMCheckedItem  = 2;     {Item is enabled, and checked or control value = 1        }
               kHMOtherItem    = 3;     {Item is enabled, and control value > 1            }
```

```
                                {Methods for displaying Help balloons:}
kHMRegularWindow    = 0;  {Create a regular window floating above all windows       }
kHMSaveBitsNoWindow = 1;  {Save the image behind the balloon and draw (like a menu) }
kHMSaveBitsWindow   = 2;  {Save the image behind the balloon + generate update event}
```

SetButtonHelpRes

Set Help data for a button using Help resources.

C
```
pascal void SetButtonHelpRes (short Button, short ResID, short ResIdx,
            long Spec);
```

Pascal
```
procedure SetButtonHelpRes (Button, ResID, ResIdx: INTEGER; Spec: LONGINT);
```

The SetButtonHelpRes routine reads one or more Help resources, and copies their data or a reference to their data (which ever is more efficient) to a button, or a control that is implemented as a button. This routine is used to manually connect Help resources to user interface elements. It is especially useful for elements that are exclusive to Tools Plus: panels, cursor tables and cursor zones, because these items cannot have Help assigned to them automatically when a dialog is opened.

Button specifies the button number (from 1 to 511) that is affected in the current window. If the current window doesn't belong to your application, or if no windows are open, or if the button does not exist in the current window, SetButtonHelpRes does nothing.

ResID specifies the resource ID that is accessed in the search for Help data. Note that up to three resources may be checked for the required data: 'hdlg', 'hrct', and 'hmnu'. The *Spec* parameter is used to specify which Help resources are accessed.

ResIdx is the item index number that is searched for Help data in the resource(s) specified by *ResID*. A value of 1 searches the first item in the Help resource, a value of 2 searches the second, and so on.

The *Spec* parameter is used to specify which Help resources are used, and which object states require Help data from the resource(s). The constants defining the available options are as follows:

Optionally choose any of the following resource searching options…

`helpUseHdlgRsrc`	Reference the 'hdlg' (dialog item) Help resource to get Help data.
`helpUseHrctRsrc`	Reference the 'hrct' (rectangles) Help resource to get Help data.
`helpUseHmnuRsrc`	Reference the 'hmnu' (menu) Help resource to get Help data.

Optionally choose the following resource searching option if none of the previous ones were used…

`helpUseAllRsrc`	Reference all Help resource ('hdlg', 'hrct' and 'hmnu') to get Help data.

Optionally choose any of the following object state searching options…

`helpEnabledState`	Get Help for the enabled state (control value = 0).
`helpDisabledState`	Get Help for the disabled state.
`helpEnabledCheckedState`	Get Help for the enabled and checked state (control value = 1).
`helpEnabledOtherState`	Get Help for the enabled, other state (control value > 1).

Optionally choose the following state searching option if none of the previous ones were used…

`helpAllStates`	Get Help for all object states.

Optionally choose the following searching option if none of the previous options were used…

`helpUseAllHelp`	Get Help for all object states, using all resource types.

```
CONST                                               {Object States:                              }
             helpUseHdlgRsrc = $00000001;           {Reference 'hdlg' resource to get Help info  }
             helpUseHrctRsrc = $00000002;           {Reference 'hrct' resource to get Help info  }
             helpUseHmnuRsrc = $00000004;           {Reference 'hmnu' resource to get Help info  }
             helpUseAllRsrc  = $000000FF;           {Reference all resources to get Help info    }
             helpEnabledState        = $00000100;   {Get Help for 'enabled' state                }
             helpDisabledState       = $00000200;   {Get Help for 'disabled' state               }
             helpEnabledCheckedState = $00000400;   {Get Help for 'enabled and checked' state    }
             helpEnabledOtherState   = $00000800;   {Get Help for 'enabled, other' state         }
             helpAllStates           = $0000FF00;   {Get Help info for all object states         }
             helpUseAllHelp = $0000FFFF;            {Get Help for all states, using all rsrc types }
```

SetPictButtonHelp

Set Help data for a picture button without using Help resources.

C
```
pascal void SetPictButtonHelp (short Button, short State,
              TipFunctionUPP TipProc, short WinResID, short Variant,
              short Method, const HMMessageRecord *HelpMessage);
```

Pascal
```
procedure SetPictButtonHelp (Button, State: INTEGER; TipProc: TipFunctionUPP;
              WinResID, Variant, Method: INTEGER; HelpMessage: HMMessageRecord);
```

SetPictButtonHelp is identical to the SetButtonHelp routine, except that this routine works on a picture button rather than a button. See the SetButtonHelp routine for details about the routine's parameters.

SetPictButtonHelpRes

Set Help data for a picture button using Help resources.

C
```
pascal void SetPictButtonHelpRes (short Button, short ResID, short ResIdx,
              long Spec);
```

Pascal
```
procedure SetPictButtonHelpRes (Button, ResID, ResIdx: INTEGER;
              Spec: LONGINT);
```

SetPictButtonHelpRes is identical to the SetButtonHelpRes routine, except that this routine works on a picture button rather than a button. See the SetButtonHelpRes routine for details about the routine's parameters.

SetScrollBarHelp

Set Help data for a scroll bar without using Help resources.

C
```
pascal void SetScrollBarHelp (short ScrollBar, short State,
              TipFunctionUPP TipProc, short WinResID, short Variant,
              short Method, const HMMessageRecord *HelpMessage);
```

Pascal
```
procedure SetScrollBarHelp (ScrollBar, State: INTEGER;
              TipProc: TipFunctionUPP; WinResID, Variant, Method: INTEGER;
              HelpMessage: HMMessageRecord);
```

SetScrollBarHelp is identical to the SetButtonHelp routine, except that this routine works on a scroll bar rather than a button. See the SetButtonHelp routine for details about the routine's parameters.

SetScrollBarHelpRes

Set Help data for a scroll bar using Help resources.

C
```
pascal void SetScrollBarHelpRes (short ScrollBar, short ResID, short ResIdx,
            long Spec);
```

Pascal
```
procedure SetScrollBarHelpRes (ScrollBar, ResID, ResIdx: INTEGER; Spec:
            LONGINT);
```

SetScrollBarHelpRes is identical to the SetButtonHelpRes routine, except that this routine works on a scroll bar rather than a button. See the SetButtonHelpRes routine for details about the routine's parameters.

SetFieldHelp

Set Help data for a field without using Help resources.

C
```
pascal void SetFieldHelp (short Field, short State, TipFunctionUPP TipProc,
            short WinResID, short Variant, short Method,
            const HMMessageRecord *HelpMessage);
```

Pascal
```
procedure SetFieldHelp (Field, State: INTEGER; TipProc: TipFunctionUPP;
            WinResID, Variant, Method: INTEGER; HelpMessage: HMMessageRecord);
```

SetFieldHelp is identical to the SetButtonHelp routine, except that this routine works on a field rather than a button. See the SetButtonHelp routine for details about the routine's parameters.

SetFieldHelpRes

Set Help data for a field using Help resources.

C
```
pascal void SetFieldHelpRes (short Field, short ResID, short ResIdx,
            long Spec);
```

Pascal
```
procedure SetFieldHelpRes (Field, ResID, ResIdx: INTEGER; Spec: LONGINT);
```

SetFieldHelpRes is identical to the SetButtonHelpRes routine, except that this routine works on a field rather than a button. See the SetButtonHelpRes routine for details about the routine's parameters.

SetListBoxHelp

Set Help data for a list box without using Help resources.

C
```
pascal void SetListBoxHelp (short ListBox, short State,
            TipFunctionUPP TipProc, short WinResID, short Variant,
            short Method, const HMMessageRecord *HelpMessage);
```

Pascal
```
procedure SetListBoxHelp (ListBox, State: INTEGER; TipProc: TipFunctionUPP;
            WinResID, Variant, Method: INTEGER; HelpMessage: HMMessageRecord);
```

SetListBoxHelp is identical to the SetButtonHelp routine, except that this routine works on a list box rather than a button. See the SetButtonHelp routine for details about the routine's parameters.

SetListBoxHelpRes

Set Help data for a list box using Help resources.

C
```
pascal void SetListBoxHelpRes (short ListBox, short ResID, short ResIdx,
            long Spec);
```

Pascal
```
procedure SetListBoxHelpRes (ListBox, ResID, ResIdx: INTEGER; Spec: LONGINT);
```

SetListBoxHelpRes is identical to the SetButtonHelpRes routine, except that this routine works on a list box rather than a button. See the SetButtonHelpRes routine for details about the routine's parameters.

SetPopUpHelp

Set Help data for a pop-up menu without using Help resources.

C
```
pascal void SetPopUpHelp (short MenuNumber, short State,
            TipFunctionUPP TipProc, short WinResID, short Variant,
            short Method, const HMMessageRecord *HelpMessage);
```

Pascal
```
procedure SetPopUpHelp (MenuNumber, State: INTEGER; TipProc: TipFunctionUPP;
            WinResID, Variant, Method: INTEGER; HelpMessage: HMMessageRecord);
```

SetPopUpHelp is identical to the SetButtonHelp routine, except that this routine works on a pop-up menu rather than a button. See the SetButtonHelp routine for details about the routine's parameters. Note that this routine only sets Help for the pop-up menu's body, and not the list that is displayed when the user clicks on the pop-up menu. To set Help for the list items, see the 'hmnu' resource earlier in this chapter.

SetPopUpHelpRes

Set Help data for a pop-up menu using Help resources.

C
```
pascal void SetPopUpHelpRes (short MenuNumber, short ResID, short ResIdx,
            long Spec);
```

Pascal
```
procedure SetPopUpHelpRes (MenuNumber, ResID, ResIdx: INTEGER;
            Spec: LONGINT);
```

SetPopUpHelpRes is identical to the SetButtonHelpRes routine, except that this routine works on a pop-up menu rather than a button. See the SetButtonHelpRes routine for details about the routine's parameters. Note that this routine only sets Help for the pop-up menu's body, and not the list that is displayed when the user clicks on the pop-up menu. To set Help for the list items, see the 'hmnu' resource earlier in this chapter.

SetPanelHelp

Set Help data for a panel without using Help resources.

C
```
pascal void SetPanelHelp (short Panel, short State, TipFunctionUPP TipProc,
            short WinResID, short Variant, short Method,
            const HMMessageRecord *HelpMessage);
```

Pascal
```
procedure SetPanelHelp (Panel, State: INTEGER; TipProc: TipFunctionUPP;
            WinResID, Variant, Method: INTEGER; HelpMessage: HMMessageRecord);
```

SetPanelHelp is identical to the SetButtonHelp routine, except that this routine works on a panel rather than a button. See the SetButtonHelp routine for details about the routine's parameters.

SetPanelHelpRes

Set Help data for a panel using Help resources.

C
```
pascal void SetPanelHelpRes (short Panel, short ResID, short ResIdx,
            long Spec);
```

Pascal
```
procedure SetPanelHelpRes (Panel, ResID, ResIdx: INTEGER; Spec: LONGINT);
```

SetPanelHelpRes is identical to the SetButtonHelpRes routine, except that this routine works on a panel rather than a button. See the SetButtonHelpRes routine for details about the routine's parameters.

SetCursorTableHelp

Set Help data for a Cursor Table without using Help resources.

C
```
pascal void SetCursorTableHelp (short CursorTable, short State,
            TipFunctionUPP TipProc, short WinResID, short Variant,
            short Method, const HMMessageRecord *HelpMessage);
```

Pascal
```
procedure SetCursorTableHelp (CursorTable, State: INTEGER;
            TipProc: TipFunctionUPP; WinResID, Variant, Method: INTEGER;
            HelpMessage: HMMessageRecord);
```

SetCursorTableHelp is identical to the SetButtonHelp routine, except that this routine works on a Cursor Table rather than a button. See the SetButtonHelp routine for details about the routine's parameters.

CursorTable specifies the cursor table number that is affected. This cursor table may be shared by any number of windows. If the cursor table does not exist, SetCursorTableHelp does nothing.

SetCursorTableHelpRes

Set Help data for a Cursor Table using Help resources.

C
```
pascal void SetCursorTableHelpRes (short CursorTable, short ResID,
            short ResIdx, long Spec);
```

Pascal
```
procedure SetCursorTableHelpRes (CursorTable, ResID, ResIdx: INTEGER;
            Spec: LONGINT);
```

SetCursorTableHelpRes is identical to the SetButtonHelpRes routine, except that this routine works on a Cursor Table rather than a button. See the SetButtonHelpRes routine for details about the routine's parameters.

CursorTable specifies the cursor table number that is affected. This cursor table may be shared by any number of windows. If the cursor table does not exist, SetCursorTableHelpRes does nothing.

SetCursorZoneHelp

Set Help data for a Cursor Zone without using Help resources.

C
```
pascal void SetCursorZoneHelp (short CursorTable, short CursorZone,
            short State, TipFunctionUPP TipProc, short WinResID,
            short Variant, short Method, const HMMessageRecord *HelpMessage);
```

Pascal
```
procedure SetCursorZoneHelp (CursorTable, CursorZone, State: INTEGER;
            TipProc: TipFunctionUPP; WinResID, Variant, Method: INTEGER;
            HelpMessage: HMMessageRecord);
```

SetCursorZoneHelp is identical to the SetButtonHelp routine, except that this routine works on a Cursor Zone rather than a button. See the SetButtonHelp routine for details about the routine's parameters.

CursorTable specifies the cursor table number that is affected. This cursor table may be shared by any number of windows. If the cursor table does not exist, SetCursorZoneHelp does nothing.

CursorZone specifies the cursor zone number that is affected within the cursor table. If the cursor zone does not exist within the cursor table specified by *CursorTable*, SetCursorZoneHelp does nothing.

SetCursorZoneHelpRes

Set Help data for a Cursor Zone using Help resources.

C
```
pascal void SetCursorZoneHelpRes (short CursorTable, short CursorZone,
            short ResID, short ResIdx, long Spec);
```

Pascal
```
procedure SetCursorZoneHelpRes (CursorTable, CursorZone, ResID,
            ResIdx: INTEGER; Spec: LONGINT);
```

SetCursorZoneHelpRes is identical to the SetButtonHelpRes routine, except that this routine works on a Cursor Zone rather than a button. See the SetButtonHelpRes routine for details about the routine's parameters.

CursorTable specifies the cursor table number that is affected. This cursor table may be shared by any number of windows. If the cursor table does not exist, SetCursorZoneHelpRes does nothing.

CursorZone specifies the cursor zone number that is affected within the cursor table. If the cursor zone does not exist within the cursor table specified by *CursorTable*, SetCursorZoneHelpRes does nothing.

SetControlHelp

Set Help data for a non-Tools Plus control without using Help resources.

C
```
pascal void SetControlHelp (ControlHandle hControl, short State,
            TipFunctionUPP TipProc, short WinResID, short Variant,
            short Method, const HMMessageRecord *HelpMessage);
```

Pascal
```
procedure SetControlHelp (hControl ControlHandle; State: INTEGER;
            TipProc: TipFunctionUPP; WinResID, Variant, Method: INTEGER;
            HelpMessage: HMMessageRecord);
```

Use SetControlHelp to set Help for a control that you have created without using a Tools Plus routine. This can happen when you use a toolbox routine to create your own controls. If you use SetControlHelp, remember to use Tools Plus's DeleteControl routine to manually delete the control, or allow Tools Plus to delete the control when the window is closed.

SetControlHelp is identical to the SetButtonHelp routine, except that this routine works on a non-Tools Plus control rather than a button. See the SetButtonHelp routine for details about the routine's parameters.

hControl is a handle to a non-Tools Plus control.

SetControlHelpRes

Set Help data for a non-Tools Plus control using Help resources.

C
```
pascal void SetControlHelpRes (ControlHandle hControl, short ResID,
            short ResIdx, long Spec);
```

Pascal
```
procedure SetControlHelpRes (hControl ControlHandle; ResID, ResIdx: INTEGER;
            Spec: LONGINT);
```

Use SetControlHelpRes to set Help for a control that you have created without using a Tools Plus routine. This can happen when you use a toolbox routine to create your own controls. If you use SetControlHelpRes, remember to use Tools Plus's DeleteControl routine to manually delete the control, or allow Tools Plus to delete the control when the window is closed.

SetControlHelpRes is identical to the SetButtonHelp routine, except that this routine works on a non-Tools Plus control rather than a button. See the SetButtonHelp routine for details about the routine's parameters.

hControl is a handle to a non-Tools Plus control.

DeleteControl

Delete a non-Tools Plus control.

C
```
pascal void DeleteControl (ControlHandle hControl);
```

Pascal
```
procedure DeleteControl (hControl ControlHandle);
```

Use the DeleteControl routine to manually delete a control that was not created with a Tools Plus routine (i.e., you used a toolbox routine to create the control, or the control was created by a third-party package other than Tools Plus). When you use Tools Plus's SetControlHelp or SetControlHelpRes routine, Tools Plus allocates private data that is associated with the control. Toolbox routines are not aware of this data, and therefore, they cannot deallocate it. When you call the DeleteControl routine, it deallocates the private data, and deletes the control by calling the toolbox's DisposeControl routine. If the control is an Appearance Manager container control, DeleteControl also deallocates

private Tools Plus data for all subcontrols before deleting them.

hControl is a handle to a non-Tools Plus control. If *hControl* is nil, *hControl* is control that is not in a Tools Plus window, or *hControl* is a control that was created by a Tools Plus routine, DeleteControl does nothing.

ChangedHelp

Force recalculation of Balloon Help.

C
```
pascal void ChangedHelp (void);
```

Pascal
```
procedure ChangedHelp;
```

The ChangedHelp routine forces Tools Plus to recalculate the Help balloon and Help message based on the current cursor position and the latest Help data. The new Help balloon is displayed when your event handler routine finishes executing. Normally, your application never needs to call this routine. It is provided as a “back door” for developers who may have this special need.

---

16 Event Management

This chapter deals entirely with managing and processing events. In a traditional Macintosh application, it is referred to as “polling” for events, or an “event loop.” Although a Tools Plus application does not poll for events, it still does have to respond to events. This chapter details how Tools Plus reports events, and how your application should respond to them. It details the following:

- Polling versus Dispatching
- Task switching
- The Macintosh’s event reporting and queuing mechanism
- The watch cursor
- Tools Plus’s event record and its fields
- Event modifiers (Caps Lock, Shift, Option, Command and Control)
- Background processing
- Event handler routine
- Filtering events (the Event Filter routine)
- Serial events
- Tools Plus event codes
- Apple Event support
- Routines for handling and processing events
- Timer events
- Each possible event is detailed, along with the correct response that should be taken by your application
- A *Field To Event Cross Reference* table quickly identifies when each field in the Tools Plus event record is used

Polling versus Dispatching

In the purest sense, polling is a periodic process a tradition Macintosh application performs in which it asks *what has happened?* This gives the application the ability to respond to the user’s actions and to Macintosh’s own internal mechanisms. The Macintosh toolbox’s Event Manager generates *events* that describe what has happened. An example of an event that is reported by the Event Manager is a mouse-down event, which contains only the time of the event and global co-ordinates at which the mouse-down occurred.

Normally, without the benefit of Tools Plus, your application must decode the event structure, interpret its meaning, and account for all possible interpretations of the event. Here is a simple example of what a traditional application, that is one without Tools Plus, would have to do:

1. Get an event using the toolbox’s GetNextEvent or WaitNextEvent
2. Based on the type of event received, the application follows a particular path of logic to decode the event structure’s data and to interpret it. In this example we’ll use a relatively simple mouse-down event. The event record’s Event.what field contains a value equal to the mouseDown constant.
3. Determine the location of the mouse-down event using the toolbox’s FindWindow routine. FindWindow returns two pieces of information: the *type* of mouse down event, and a pointer to a window if the mouse-down event occurred in a window. The possible mouse-down types are: inDesk, inMenuBar, inSysWindow, inContent, inDrag, inGrow, inGoAway, inZoomIn, and inZoomOut. The traditional application needs to account for each one of these and do additional processing based on the type. In this example, let’s choose the simple “inContent” type that indicates the mouse-down event is inside a window’s content region. The application now has a *pointer* to a window that it must somehow relate back to the original window, and it knows the mouse went down somewhere inside the window’s content region. Let’s also assume, for the sake of simplicity, that the target window is already active.
4. Determine where in the window the mouse-down occurred. A traditional application needs to check all objects in the window to determine in which object, if any, the mouse-down occurred. Without Tools Plus, the application needs to account for list boxes, buttons and scroll bars, editing fields, pop-up menus, picture buttons, and

possibly the beginning of a click or drag in the window. We'll explore an easy example in which the user interacts with a simple push button. A traditional application uses the toolbox's FindControl routine to determine which control is affected. FindControl returns a handle to the control in which the mouse-down event occurred. Realize that the handle could point to virtually any kind of control: a push button, radio button, check box, scroll bar, or a less common custom control. Let us assume that the traditional application somehow determines through its own logic that the control is a simple push button.

5. The traditional application now needs to track the mouse in and out of the push button control while the mouse button is down using the toolbox's TrackControl routine. This routine highlights the button while the mouse is in it, and unhighlights it while the mouse is out of the button. The TrackControl routine returns control to the application when the user releases the mouse button, and it informs the application if the mouse was inside the control when the mouse button was released.
6. If TrackControl returns *true*, the application can now process the action resulting from the user clicking the push button. Otherwise it ignores the mouse-down event.
7. When processing the button click, the traditional application knows only the following information:
    - a mouse-down event occurred
    - in a window (it has a pointer to the window)
    - in a control (it has a handle to the control)

   The traditional application must still relate the pointer and handle back to what they represent, such as an "Order Entry" window and a "Cancel" button inside that window.

In sharp contrast to all the drudgery that is found in a traditional application, Tools Plus automatically does the vast majority of the work for you. In a Tools Plus application, you simply write an event handler routine (detailed later) that responds to a specific occurrence in your application, such as the user clicking a button. Tools Plus calls your event handler routine each time it has an event that needs to be processed. This is called "event dispatching" because Tools Plus does all the getting and processing of events, and it hands the resulting occurrence (dispatches it) to your application for an appropriate response. Here is how an application written *with* Tools Plus processes the same mouse-down event that took a page to detail earlier:

1. Tools Plus calls your event handler routine.
2. Your event handler routine has access to a global event record that tells your application what happened:
    - a doButton event occurred (user selected a button)
    - window #2
    - button #10
    - modifier keys such as Caps Lock, Control, Command, Shift, etc.
    - toolbox event record (in case you want it)

   A window number and button number are provided since you create and reference all Tools Plus objects by a number instead of by a pointer or handle.

Your application simply does what should be done in response to the user clicking the button, such as closing the window after a "Cancel" button is clicked. Tools Plus does similar event processing for all other elements of your user interface, thus making your interface work automatically.

Task Switching

With the introduction of System 5, MultiFinder made cooperative multitasking a reality on the Macintosh. Cooperative, or "switched" multitasking as it is often called, lets several applications appear to run simultaneously by having the processor cycle amongst all the applications (or tasks). Even though only one task is being executed at any given moment, the cycling happens so quickly that it gives a user the impression that they are happening simultaneously. The term "cooperative" is used because each application must cooperate with all others by relinquishing control to give the others adequate processing time.

Only one application can be active (the frontmost window) at a time even though, potentially, you may be able to see dozens of windows from multiple applications simultaneously. Therefore, the active application is temporarily "suspended" when another application or desk accessory is activated. Suspended applications can also receive events and processing time to let them perform operations while they are suspended, such as refreshing a window after it has been uncovered.

There are two kinds of “switches” that occur in cooperative multitasking: *minor* and *major*. In a major switch, your application is either *suspended* when another application or desk accessory is activated, or *resumed* when your application is activated. The doSuspend and doResume events report this occurrence to your application. In a minor switch, your application is given some processing time, then it releases control to another application or desk accessory.

Tools Plus takes care of task switching. Every time your application finishes processing the code in its event handler routine, a major or minor switch will always occur. You only need to concern yourself with minor switches by making sure that the code in your event handler routine does not take too long to run. In other words, don’t write your event handler routine so that it takes ten seconds to execute, or other applications will work jerkily, or worse yet, they may appear to mysteriously “hang” when in actuality they just aren’t getting enough cooperation from your application. The doSuspend and doResume events detail how your application should respond to a major switch.

See the SIZE resource to specify how your application will behave while suspended, and during the transition from being active to inactive or vice versa.

Macintosh Events

This section is included for reference purposes only. Your application will likely never need to interact directly with the toolbox’s Event Manager or its data since Tools Plus reads, decodes, interprets and processes these events automatically.

The Event Manager, as described in Inside Macintosh, is the link between your application and its user, and its machine environment. Whenever the user types a key on the keyboard or numeric keypad, presses the mouse button, or inserts a floppy disk in the disk drive, the Event Manager reports an event to the application. In addition to monitoring the user’s actions, the Event Manager also detects other types of events that serve to inform your application as to “what is happening.” Such an event is reported when a partially obscured window is uncovered and its contents need to be redrawn (update event).

The Event Manager’s event record contains an event code that identifies the type of event. Event codes are available as constants. Later, you will learn how the Event Manager’s events are automatically translated into Tools Plus events. The following is a list of events that can be generated by the Event Manager:

```
CONST          nullEvent       = 0;   {no event detected                      }
               mouseDown       = 1;   {mouse button was pressed down          }
               mouseUp         = 2;   {mouse button was released              }
               keyDown         = 3;   {a key was pressed                      }
               keyUp           = 4;   {a key was released                     }
               autoKey         = 5;   {a key was held down and is repeating   }
               updateEvt       = 6;   {a window must be updated (refreshed)   }
               diskEvt         = 7;   {a disk was inserted                    }
               activateEvt     = 8;   {a window was activated or deactivated  }
               networkEvt      = 10;  {network                                }
               driverEvt       = 11;  {device driver                          }
               app1Evt         = 12;  {application-defined event number 1     }
               app2Evt         = 13;  {application-defined event number 2     }
               app3Evt         = 14;  {application-defined event number 3     }
               app4Evt         = 15;  {appl-defined event 4 / MultiFinder's   }
                                      {  and Switcher's Suspend/Resume event  }
               osEvt           = 15;  {Operating-system event (System 7+)     }
               kHighLevelEvent = 23;  {High-level event (System 7+)           }
```

The Event Queue

When events are generated, they are stored in a “First In First Out” (FIFO) event queue until your application can retrieve and process them. When your application is ready to process an event, the oldest event is removed from the queue and processed first. This journaling mechanism lets the Macintosh remember a series of rapidly occurring events and store them until your application is ready to process them.

Events have a certain priority, meaning that some events will be processed before others, regardless of when they were generated. Their priority is as follows:

1. activate/deactivate a window
2. mouse-down/up, key down/up, disk insert, network driver, application-defined events (first in first out)

3. auto-key (key pressed and held, causing it to repeat)
4. update a window (refresh a window in front-to-back order)

The priority of events insures that illogical events are not reported. For instance, the user may click *twice* in a window's close box before an application gets around to processing the event. The first click signals your application to close the window. The *second* click will not be reported as a click in a non-existent (closed) window. The second click's location is analyzed *after* the window is closed, and reported accordingly.

Tools Plus works in an identical manner when interacting with the event queue. In fact, Tools Plus obtains events from the Event Manager and translates them into something useful before reporting them to your application.

Note: The event queue can store a maximum of 20 events. If your application is so busy that it lets more than 20 events accumulate in the queue, the oldest events are discarded to make room for the new ones. Your application may lose events and may misbehave if it takes a long time to execute the code in your event handler routine. If this is the case, call Process1EventWhileBusy during long processes.

Watch Cursor -- a busy system

The watch cursor is used to indicate a long wait, such as when a lengthy process is being conducted or when printing is being done. Your application can display the watch cursor by calling CursorShape(watchCursor).

While the watch cursor is displayed, your application may choose not to process events, since it doesn't care what the user is doing and it will want to ignore the user's mouse clicks and typing anyway. Unfortunately, this does not give the user the opportunity to halt a lengthy process, nor does it give other applications any processing time. Tools Plus solves this dilemma by shifting into a *busy* mode when the watch cursor is displayed.

When your application displays the watch cursor, Tools Plus automatically filters out (discard) the mouse and typing events generated by the user. The exception to this, of course, is when the user types ⌘-. that tells your application to halt the process. This lets your application process events regularly while it is busy conducting a lengthy task, knowing that any key or mouse event is meaningful (it indicates that the user wants to halt the process).

When your application finishes its lengthy process, it can either change the cursor itself or call ResetCursor to change the cursor to its appropriate shape according to its position on the screen.

Note: When the watch cursor is displayed, your application can still receive Tools Plus events that are unrelated to mouse clicks or typing, such as redrawing a window (doRefresh) or a disk insert event.

Warning: If your application is running under MultiFinder or System 7 (or later), it is possible to switch to another application while a watch cursor is displayed unless your active window is modal across all applications, like the dBoxProc type window.

Tools Plus Event Record

When you create your application, you declare a global Tools Plus event record variable (TPEventRecord type) that is used by both Tools Plus and your application. Tools Plus populates the global event record with information about the most recent event, and your application can reference this information at any time. The Designing Your Application chapter details how to define the global event record, and the Initialization chapter details how Tools Plus becomes aware of this record in order to populate it. Although the event record *looks* cumbersome, it is logically organized and it is very easy to use.

The entire Tools Plus event record accounts for every type of event that can possibly occur. During any *one specific event*, your application will need to access only one or two of these fields. Later in this chapter when you are told how to respond to the various Tools Plus events, you will also be told which fields in the event record hold information that is pertinent to the event.

The event record is defined as follows:

C

```
/* Event record's "event modifiers" info*/
  union TPModifiersRec {                           /*This variable record contains an event's            */
                                                   /*  "modifiers" in 2 formats…                         */
                                                   /*  · Macintosh Event:                                */
    short Num;                                     /*      integer (bit operations required)             */
    struct {                                       /*  · Modifier short parsed into components:          */
      unsigned short bit15 :1;                     /*      (reserved bit)                                */
      unsigned short bit14 :1;                     /*      (reserved bit)                                */
      unsigned short bit13 :1;                     /*      (reserved bit)                                */
      unsigned short ControlKey :1;                /*      Control key was down at event (=1)            */
      unsigned short OptionKey :1;                 /*      Option key was down at event (=1)             */
      unsigned short CapsLock :1;                  /*      Caps Lock was down at event (=1)              */
      unsigned short ShiftKey :1;                  /*      Shift key was down at event (=1)              */
      unsigned short CmdKey :1;                    /*      Command key was down at event (=1)            */
      unsigned short MouseUp :1;                   /*      Mouse button was UP at event (=1)             */
      unsigned short bit6 :1;                      /*      (reserved bit)                                */
      unsigned short bit5 :1;                      /*      (reserved bit)                                */
      unsigned short bit4 :1;                      /*      (reserved bit)                                */
      unsigned short bit3 :1;                      /*      (reserved bit)                                */
      unsigned short bit2 :1;                      /*      (reserved bit)                                */
      unsigned short bit1 :1;                      /*      (reserved bit)                                */
      unsigned short bit0 :1;                      /*      (reserved bit)                                */
    } Bits;                                        /*                                                    */
  };
  typedef union TPModifiersRec TPModifiersRec;
/*= Event record's "button" info*/
  struct TPEventButtonRec {
    short Num;                                     /*Button number                                       */
    short Part;                                    /*Button's part code (usually ignored)                */
    Boolean DoubleClick;                           /*Did a double-click occur?                           */
  };
  typedef struct TPEventButtonRec TPEventButtonRec;
/*= Event record's "scroll bar" info*/
  struct TPEventScrollBarRec {
    short Num;                                     /*Scroll bar number                                   */
    short Part;                                    /*Scroll bar's part                                   */
  };
  typedef struct TPEventScrollBarRec TPEventScrollBarRec;
/*= Event record's "list box" info*/
  struct TPEventListBoxRec {
    short Num;                                     /*List box number                                     */
    Boolean DoubleClick;                           /*Did a double-click occur?                           */
  };
  typedef struct TPEventListBoxRec TPEventListBoxRec;
/*= Event record's "menu" info*/
  struct TPEventMenuRec {
    short Num;                                     /*Menu number                                         */
    short Item;                                    /*Menu item number                                    */
    short SubMenu;                                 /*Submenu number (for pop-up menus)                   */
  };
  typedef struct TPEventMenuRec TPEventMenuRec;
/*= Event record's "key-stroke" info*/
  struct TPEventKeyRec {
    short Code;                                    /*Logical key number of the typed key                 */
    char Chr;                                      /*ASCII character generated by typed key              */
    Byte Unused;                                   /* (reserved byte)                                    */
  };
  typedef struct TPEventKeyRec TPEventKeyRec;
/*= Event record's "mouse location and time" info for mouse-down and mouse-up events*/
  struct TPEventMousePointRec {
    Point Where;                                   /*Event location in local co-ordinates                */
    long When;                                     /*Event time in clock ticks from boot (1/60 sec)      */
    TPModifiersRec Modifiers;                      /*Event modifiers                                     */
  };
  typedef struct TPEventMousePointRec TPEventMousePointRec;
/*= Event record's "mouse click/drag" info*/
  struct TPEventMouseRec {
    short What;                                    /*What type of mouse event? (click or drag)           */
    TPEventMousePointRec Down[3];                  /*Where & when did the mouse-down occur               */
    TPEventMousePointRec Up[3];                    /*Where & when did the mouse-up occur                 */
    Point Where;                                   /*Current mouse location in local co-ordinates        */
    short DialogItem;                              /*Dialog item number "hit" with first mouse-down      */
  };
  typedef struct TPEventMouseRec TPEventMouseRec;
/*= Event record's "Timer" info*/
  struct TPTimerRec {
    short Num;                                     /*Timer number                                        */
    long Count;                                    /*Number of events reported                           */
    long NextTime;                                 /*Time of next event                                  */
  };
  typedef struct TPTimerRec TPTimerRec;
/*= EVENT record for Tools Plus*/
  struct TPEventRecord {                           /*Tools Plus Event record…                            */
    short What;                                    /*What type of event has occurred?                    */
    short Window;                                  /*Window number of the event                          */
    TPEventButtonRec Button;                       /*Button number/double-click status                   */
    TPEventScrollBarRec ScrollBar;                 /*Scroll bar number/scroll bar part                   */
    short Field;                                   /*Field number of event                               */
    TPEventListBoxRec ListBox;                     /*List box number/double-click status                 */
    TPEventMenuRec Menu;                           /*Menu number/menu item of an event                   */
    TPEventKeyRec Key;                             /*Logical key number & character of typed key         */
    TPEventMouseRec Mouse;                         /*Click/drag info: [1..3] where & when                */
    TPTimerRec Timer;                              /*Timer info                                          */
    TPModifiersRec Modifiers;                      /*Modifier flags                                      */
    EventRecord Event;                             /*Macintosh Toolbox Event (raw)                       */
  };
  typedef struct TPEventRecord TPEventRecord;
  typedef TPEventRecord *TPEventPointer;           /*Pointer to a Event record                           */
```

Pascal

```
type
{= Event record's "event modifiers" info}
  TPModifiersRec = packed record                {This variable record contains an event's        }
    case integer of                             {  "modifiers" in 2 formats…                     }
      0: (                                      {  · Macintosh Event:                            }
        Num: integer                            {      integer (bit operations required)         }
      );                                        {                                                }
      1: (                                      {  · Modifier integer parsed into components:    }
        bit15, bit14, bit13: boolean;           {      (reserved bits)                           }
        ControlKey: boolean;                    {      Control key was down at event (=1)        }
        OptionKey: boolean;                     {      Option key was down at event (=1)         }
        CapsLock: boolean;                      {      Caps Lock was down at event (=1)          }
        ShiftKey: boolean;                      {      Shift key was down at event (=1)          }
        CmdKey: boolean;                        {      Command key was down at event (=1)        }
        MouseUp: boolean;                       {      Mouse button was UP at event (=1)         }
        bit6, bit5, bit4, bit3, bit2, bit1,     {      (reserved bits)                           }
        bit0: boolean                           {                                                }
      );                                        {                                                }
    end;
{= Event record's "button" info}
  TPEventButtonRec = record
    Num: integer;                               {Button number                                   }
    Part: integer;                              {Button's part code (usually ignored)            }
    DoubleClick: boolean                        {Did a double-click occur in the button?         }
   end;
{= Event record's "scroll bar" info}
  TPEventScrollBarRec = record
    Num: integer;                               {Scroll bar number                               }
    Part: integer                               {Scroll bar's part                               }
  end;
{= Event record's "list box" info}
  TPEventListBoxRec = record
    Num: integer;                               {List box number                                 }
    DoubleClick: boolean                        {Did a double-click occur in a line?             }
  end;
{= Event record's "menu" info}
  TPEventMenuRec = record
    Num: integer;                               {Menu number                                     }
    Item: integer;                              {Menu item number                                }
    SubMenu: integer                            {Submenu number (for pop-up menus)               }
  end;
{= Event record's "key-stroke" info}
  TPEventKeyRec = packed record
    Code: integer;                              {Key number of key-stroke                        }
    Chr: char;                                  {Character generated by key-stroke               }
    Unused: byte                                { (reserved byte)                                }
  end;
{= Event record's "mouse location and time" info for mouse-down and mouse-up events}
  TPEventMousePointRec = record
    Where: point;                               {Event location in local co-ordinates            }
    When: longint;                              {Event time in ticks from boot (1/60 sec)        }
    Modifiers: TPModifiersRec                   {Event modifiers                                 }
  end;
{= Event record's "mouse click/drag" info}
  TPEventMouseRec = record
    What: integer;                              {What type of mouse event (click or drag)?       }
    Down: array[1..3] of TPEventMousePointRec;  {Where & when did the mouse-down occur           }
    Up: array[1..3] of TPEventMousePointRec;    {Where & when did the mouse-up occur             }
    Where: point;                               {Current mouse location in local co-ordinates    }
    DialogItem: integer;                        {Dialog item number "hit" with first mouse-down  }
  end;
{= Event record's "Timer" info}
  TPTimerRec = record
    Num: integer;                               {Timer number                                    }
    Count: longint;                             {Number of events reported                       }
    NextTime: longint;                          {Time of next event                              }
   end;

{= EVENT record for "Tools Plus" }
  TPEventRecord = record
    What: integer;                              {What type of event has occurred?                }
    Window: integer;                            {Window number of the event                      }
    Button: TPEventButtonRec;                   {Button number/double-click status               }
    ScrollBar: TPEventScrollBarRec;             {Scroll bar number/scroll bar part               }
    Field: integer;                             {Field number of event                           }
    ListBox: TPEventListBoxRec;                 {List box number/double-click status             }
    Menu: TPEventMenuRec;                       {Menu number/menu item of an event               }
    Key: TPEventKeyRec;                         {Key number & character of key-stroke            }
    Mouse: TPEventMouseRec;                     {Click/drag info: [1..3] where & when            }
    Timer: TPTimerRec;                          {Timer info                                      }
    Modifiers: TPModifiersRec;                  {Modifier flags                                  }
    Event: EventRecord                          {Macintosh Toolbox Event (raw)                   }
  end;
  TPEventPointer = ^TPEventRecord;              {Pointer to a Event record, in case you          }
                                                {   want to reduce global variable memory.       }
```

Event Record Fields

The event record's structure is best explained by detailing each subrecord and field with a programmer's approach. This explanation looks at the event record in detail, starting from the record level and ending with fields. Your application should define a global variable or global pointer than lets it use the event record. In the following text, the assumption is made that the global variable is called *Event*. If your application uses a pointer, replace *Event* with *Event^* in the text.

Note that all fields are not valid at the same time. For example, the window field does not contain a valid number when a pull-down menu event is reported because menus work independently of windows. A "Field to Event Cross Reference" is included later in this section and it lists each field and the events that make use of it.

Field	Description
`Event`	This variable is the entire tools plus event record, and is of TPEventRecord type (it is a TPEventPointer type if your application uses a pointer to the record)
`Event.What`	**Event Code**: Explains what type of event has occurred. This field is used by your application to decide what action should be taken, and what other fields contain pertinent event information.
`Event.Window`	**Window Number**: Window number on which the event occurred.
`Event.Button`	**Button Record**
`Event.Button.Num`	**Button Number**: Button number that was clicked by the user.
`Event.Button.Part`	**Button Part**: Part of the button that was clicked by the user (usually ignored).
`Event.Button.DoubleClick`	**Button's Double-Click Status**: Was the button double-clicked?
`Event.ScrollBar`	**Scroll Bar Record**
`Event.ScrollBar.Num`	**Scroll Bar Number**: Scroll bar number that was clicked by the user. This does *not* include a scroll bar that is part of a list box or editing field, as they work automatically.
`Event.ScrollBar.Part`	**Scroll Bar Part**: Part of scroll bar that was clicked by user (up arrow, down arrow, "page up" region, "page down" region, thumb)
`Event.Field`	**Editing Field Number**: Editing field that was clicked by the user.
`Event.ListBox`	**List Box Record**
`Event.ListBox.Num`	**List Box Number**: List box number that was clicked by the user.
`Event.ListBox.DoubleClick`	**List Box's Double-Click Status**: Was a line in the list box double-clicked?
`Event.Menu`	**Menu Record**
`Event.Menu.Num`	**Menu Number**: Menu number that was selected by the user. This could be a pull-down menu, a hierarchical menu, or a pop-up menu.
`Event.Menu.Item`	**Menu Item**: Item number that was selected by the user.
`Event.Menu.SubMenu`	**Submenu Number**: Hierarchical menu number selected from a pop-up menu.
`Event.Key`	**Key Record**
`Event.Key.Code`	**Key Number**: Number of the key that was pressed, released, or is auto-repeating. This key code is a key number that is not affected by the Caps Lock, Shift, Option, Command and Control modifiers.
`Event.Key.Chr`	**Key Character**: Character resulting from a key that was pressed, released, or is auto-repeating. This character is altered by the Caps Lock, Shift, Option, Command and Control modifiers.
`Event.Mouse`	**Mouse Activity Record**
`Event.Mouse.What`	**Mouse Event Code**: Type of mouse event has occurred (i.e., single, double, triple-click, dragging, etc.). This field is used by your application to decide what action should be taken, and which other fields in the mouse record contain pertinent information.
`Event.Mouse.Down`	**Mouse-Down Array Record:** The elements of the array contain the first, second and third mouse-down events of a single, double, or triple-click, or drag.
`Event.Mouse.Down[1].Where`	**Mouse Down Location**: Location of 1st mouse-down in a single, double, or triple-click, or drag. Window's local co-ordinates are used.
`Event.Mouse.Down[1].When`	**Mouse Down Time**: Time of 1st mouse-down. Number of "ticks" since startup.
`Event.Mouse.Down[1].Modifiers`	**Mouse Down Modifier Record**: Event modifiers at the time of the 1st mouse-down event. See the note on modifiers below.
`Event.Mouse.Down[2]…`	**Mouse-Down Array Record:** Same fields, but for 2nd mouse-down in a double, or triple-click, or drag.

`Event.Mouse.Down[3]…` **Mouse-Down Array Record:** Same fields, but for 3rd mouse-down in a triple-click, or drag.

`Event.Mouse.Up` **Mouse-Up Array Record:** The elements of the array contain the first, second and third mouse-up events of a single, double, or triple-click, or drag.

`Event.Mouse.Up[1].Where` **Mouse Up Location**: Location of 1st mouse-up in a single, double, or triple-click, or drag. Window's local co-ordinates are used.

`Event.Mouse.Up[1].When` **Mouse Up Time**: Time of 1st mouse-up. Number of "ticks" since startup.

`Event.Mouse.Up[1].Modifiers` **Mouse Up Modifier Record**: Event modifiers at the time of the 1st mouse-up event. See the note on modifiers below.

`Event.Mouse.Up[2]…` **Mouse-Up Array Record:** Same fields, but for 2nd mouse-up in a double, or triple-click, or drag.

`Event.Mouse.Up[3]…` **Mouse-Up Array Record:** Same fields, but for 3rd mouse-up in a triple-click, or drag.

`Event.Mouse.Where` **Mouse Location**: Current mouse location in window's local co-ordinates.

`Event.Mouse.DialogItem` **Dialog Item Number**: Item number that was "hit" by the first mouse-down event in a window that has a dialog item list.

`Event.Timer` **Timer Record**

`Event.Timer.Num` **Timer Number**: Timer number that generated the Timer event.

`Event.Timer.Count` **Timer's Event Count**: Sequential event counter (i.e., the number of times this Timer reported an event).

`Event.Timer.NextTime` **Next time Timer will report an event**: Time in ticks from boot time when this Timer will report its next event.

`Event.Modifiers` **Modifier Record**: Event modifiers at the time when the event occurred. See the note on modifiers below.

`Event.Event` **The Event Manager's Event Record**: A completely unaltered event record as retrieved from the Event Manager. This is used for applications that want to process their own custom events or custom controls.

Note: *For C programmers…* The event record's arrays are documented as using Pascal nomenclature (the elements are numbered 1, 2 and 3). In C, the same array's elements are numbered 0, 1 and 2 (they start at zero). In this manual, when C programmers read:

```
Event.Mouse.Down[1].Where
```

it indicates the first element of the array, which translates to the following C source code:

```
Event.Mouse.Down[0].Where
```

Event Modifiers

Tools Plus's event modifier field provides information identical to that obtained directly from the Macintosh's Event Manager. To reiterate, the Modifiers field of the event record contains information about the position of the Caps Lock, Shift, Option, Command and Control keys at the time of the event, as well as the position of the mouse button. This can be used, for example, to detect if the Command key was down when a key was typed (i.e., a ⌘-key sequence). In effect, your application could respond to command key sequences that are *not* menu equivalents by using this method. Your application could also place special significance on Option-Clicks. The most common use, however, is Shift-Tab that indicates the user wants to tab to the *previous* field.

In the Macintosh's Event Manager, the Modifiers field of the event record is an integer. Several of the bits indicate the state of the various modifiers as detailed in Inside Macintosh. Tools Plus goes a step further and automatically decodes individual modifiers in the field. This is handled through a *union* in C, and a *variant record* in Pascal.

Event Modifiers Using C

When programming in C, Tools Plus's Modifiers structure is a *union* that lets you access both the integer (16-bit short) obtained from the Macintosh's Event Manager, as well as the individual flags (bits) within the integer. Tools Plus's Modifiers structure looks like this:

```
union TPModifiersRec {                /*This variable record contains an event's       */
                                      /*  "modifiers" in 2 formats…                   */
                                      /*  · Macintosh Event:                          */
  short Num;                          /*      integer (bit operations required)       */
  struct {                            /*  · Modifier short parsed into components:    */
    unsigned short bit15 :1;          /*      (reserved bit)                          */
    unsigned short bit14 :1;          /*      (reserved bit)                          */
    unsigned short bit13 :1;          /*      (reserved bit)                          */
    unsigned short ControlKey :1;     /*      Control key was down at event (=1)      */
    unsigned short OptionKey :1;      /*      Option key was down at event (=1)       */
    unsigned short CapsLock :1;       /*      Caps Lock was down at event (=1)        */
    unsigned short ShiftKey :1;       /*      Shift key was down at event (=1)        */
    unsigned short CmdKey :1;         /*      Command key was down at event (=1)      */
    unsigned short MouseUp :1;        /*      Mouse button was UP at event (=1)       */
    unsigned short bit6 :1;           /*      (reserved bit)                          */
    unsigned short bit5 :1;           /*      (reserved bit)                          */
    unsigned short bit4 :1;           /*      (reserved bit)                          */
    unsigned short bit3 :1;           /*      (reserved bit)                          */
    unsigned short bit2 :1;           /*      (reserved bit)                          */
    unsigned short bit1 :1;           /*      (reserved bit)                          */
    unsigned short bit0 :1;           /*      (reserved bit)                          */
  } Bits;                             /*                                              */
};
```

Whenever you access any of the individual modifier flags, you reference the “bits” part of the structure. For example, if you want to check if the Command key was down and the Option key was up when a key was typed, you can use an expression such as this:

```
if ((Event.Modifiers.Bits.CmdKey) && (!Event.Modifiers.Bits.OptionKey))
```

In contrast, the Modifiers field provided by the Macintosh's Event Manager requires bitwise “And” operations to determine if a bit is set or not, thereby resulting in source code that looks more cryptic. The following line duplicates the previous example's functionality using bitwise “And” operations instead of the available bits. Note that we are using the toolbox's event record, theEvent, in place of Tools Plus's event record.

```
if ((theEvent.modifiers & cmdKey) && !(theEvent.modifiers & optionKey))
```

When you are working with the Modifiers structure, you may perform operations exclusively on the integer variant of the structure, on the bits variant, or if you choose, you can mix and match as needed. Several constants representing the “bit-equivalents” for the various flags contained in the Modifiers integer are available as follows:

```
                                 /*Modifier masks                          */
#define   btnState     0x0080    /*set to 1 if mouse button is up          */
#define   cmdKey       0x0100    /*set to 1 if Command key is down         */
#define   shiftKey     0x0200    /*set to 1 if Shift key is down           */
#define   alphaLock    0x0400    /*set to 1 if the Caps Lock key is down   */
#define   optionKey    0x0800    /*set to 1 if the option key is down      */
#define   controlKey   0x1000    /*set to 1 if the control key is down     */
```

Pascal Event Modifiers Using Pascal

When programming in Pascal, Tools Plus's Modifiers structure is a *variant record* that lets you access both the integer (16 bits) obtained from the Macintosh's Event Manager, as well as the individual flags (bits) within the integer. Tools Plus's Modifiers record looks like this:

```
TPModifiersRec = packed record          {This variable record contains an event's      }
  case integer of                        {  "modifiers" in 2 formats…                   }
    0: (                                 {  · Macintosh Event:                          }
      Num: integer                       {      integer (bit operations required)       }
    );                                   {                                              }
    1: (                                 {  · Modifier integer parsed into components:  }
      bit15, bit14, bit13: boolean;      {      (reserved bits)                         }
      ControlKey: boolean;               {      Control key was down at event (=1)      }
      OptionKey: boolean;                {      Option key was down at event (=1)       }
      CapsLock: boolean;                 {      Caps Lock was down at event (=1)        }
      ShiftKey: boolean;                 {      Shift key was down at event (=1)        }
      CmdKey: boolean;                   {      Command key was down at event (=1)      }
      MouseUp: boolean;                  {      Mouse button was UP at event (=1)       }
      bit6, bit5, bit4, bit3, bit2, bit1,{      (reserved bits)                         }
      bit0: boolean                      {                                              }
    );                                   {                                              }
  end;
```

You use each field in the record as an individual modifier variable. For example, if you want to check if the Command key was down and the Option key was up when a key was typed, you can use an expression such as this:

```
if Event.Modifiers.CmdKey and not Event.Modifiers.OptionKey then
```

In contrast, the Modifiers field provided by the Macintosh's Event Manager requires bitwise "And" operations to determine if a bit is set or not, thereby resulting in source code that looks more cryptic. The following line duplicates the previous example's functionality using bitwise "And" operations instead of the available bits. Note that we are using the toolbox's event record, theEvent, in place of Tools Plus's event record.

```
if (BitAnd(theEvent.modifiers,cmdKey) <> 0) and (BitAnd(theEvent.modifiers,optionKey) = 0) then
```

When you are working with the Modifiers record, you may perform operations exclusively on the integer variant of the record, on the bits variant, or if you choose, you can mix and match as needed. Several constants representing the "bit-equivalents" for the various flags contained in the Modifiers integer are available as follows:

```
CONST                              {Modifier masks                        }
          btnState   = $0080;  {set to 1 if mouse button is up        }
          cmdKey     = $0100;  {set to 1 if Command key is down       }
          shiftKey   = $0200;  {set to 1 if Shift key is down         }
          alphaLock  = $0400;  {set to 1 if the Caps Lock key is down }
          optionKey  = $0800;  {set to 1 if the option key is down    }
          controlKey = $1000;  {set to 1 if the control key is down   }
```

Background Processing

Applications written with Tools Plus can easily be made to do "background processing," that is, performing an on-going process while waiting for events. An example of this is repaginating a word-processing document or searching a database for specific records.

When Tools Plus calls your event handler routine and the global event record's Event.What field has a value of zero (doNothing), it indicates that no event has occurred. This is commonly called a "null event." Usually, your application won't do anything when it receives a null event. If your application does background processing, it should do its work *only* when it receives a doNothing event. A single cycle of your application's background process should be rather short, ideally about 1/60 of a second or less. A background process that takes 1/20 of a second (about 3 clock ticks) is on the threshold of human perception. Any longer and other applications may run sluggishly or in spurts and jumps.

When no events are available for your application, Tools Plus calls your event handler routine and reports a doNothing event. This can consume a fair amount of processor capacity just to tell your application that nothing has happened. An average Macintosh is capable of getting several hundred doNothing events per second, and a Power Macintosh is capable of over one thousand per second. To reduce or eliminate this waste of processing power, your application can schedule how often it receives doNothing events by using the SetNullTime routine.

If your application…
- requires significant processing power for a background process
- requires uninterrupted execution for its background process
- has multiple background processes running simultaneously,

consider writing the background process as a thread using the Thread Manager, or as a “server task” under Mac OS 8 or later. This is documented in Inside Macintosh.

There are situations when your application simply cannot avoid having a lengthy process running, such as when your application’s event handler routine calls a printing routine. During the lengthy printing process, it is necessary to give other applications some processing time as well as to process events in your own application, such as the user typing ⌘-. to halt the lengthy process. In such situations, your application should periodically call the Process1EventWhileBusy routine. The Process1EventWhileBusy routine temporarily lets Tools Plus check for an event, process it, give some processing time to other application, then return control to your application.

If your application needs to do background processing, and it needs more precise control over the frequency with which the background process is called, see “Timer Events” later in this chapter. The Tools Plus Timer lets you easily set periodic or timed tasks.

The Event Handler Routine

Traditional Macintosh applications have an “event loop” where the application repeatedly asks what has happened (gets an event), responds to the event (translates and processes it) then gets another event. This logic can become rather difficult to manage in more complex applications that need multiple event loops, or those that need to process events from within a routine that is already responding to an event, such as a printing routine or sorting routine.

In Tools Plus, you write an *event handler* routine for your application. Tools Plus calls your routine to respond to high-level Tools Plus events as well as to some toolbox events that cannot be processed by Tools Plus. Your event handler routine has the following C/C++ prototype or Pascal interface:

C

```
pascal void MyEventHandler (Ptr CustomDataPtr)
   {
   }
```

Pascal

```
procedure MyEventHandler (CustomDataPtr: Ptr);
   begin
   end;
```

CustomDataPtr is a pointer to your own custom data structure, just in case you want to write your application to call its own event handler. Most applications will ignore this parameter. The CustomDataPtr pointer is always set to nil when Tools Plus calls your event handler routine. Inside your event handler routine, all you need is a C/C++ switch statement or a Pascal case statement to respond to your application’s global event record. The way you write your event handler is very much a matter of personal taste, so there isn’t a “standard” way of writing an event handler. The simplest event handler responds to only one window. Here is an example:

C

```
pascal void MainEventHandler (Ptr CustomDataPtr)
  {
  switch (Event.What)                         /*Respond to each type of event…           */
    {
    case doActivate:                          /*User wants to activate the window…       */
      MyActivateRoutine();
      break;
    case doRefresh:                           /*Window needs to be refreshed…            */
      MyRefreshRoutine();
      break;
    case doGoAway:                            /*User clicked window's close box          */
      MyCloseRoutine();
      break;
    case doButton:                            /*User clicked a button…                   */
      switch (Event.Button.Num)               /*Respond to specific type of button…      */
        {
        case kOKbutton:                       /*User clicked OK button…                  */
          myOKroutine();                      /*                                         */
          break;
        case kCancelButton:                   /*User clicked Cancel button               */
```

```
          myCancelRoutine();                 /*                                              */
          break;
        /*cases for other buttons*/
        }
      break;
    case doMenu:                             /*User selected a menu…                         */
      MyMenuRoutine();
      break;
    case doNothing:                          /*No event available. If your app does any      */
      MyBackgroundRoutine();                 /*  background processing, execute one “cycle.” */
      break;

    /*cases for other events*/
    default:                                 /*Ignore events that are not listed in the cases */
      break;
    }
  }
```

Pascal

```
procedure MainEventHandler (CustomDataPtr: Ptr);
  begin
    case Event.What of                        {Respond to each type of event…                }
      doActivate:                             {User wants to activate the window…            }
        MyActivateRoutine;
      doRefresh:                              {Window needs to be refreshed…                 }
        MyRefreshRoutine;
      doGoAway:                               {User clicked window's close box               }
        MyCloseRoutine;
      doButton:                               {User clicked a button…                        }
        case Event.Button.Num of              {Respond to specific type of button…           }
          kOKbutton:                          {User clicked OK button…                       }
            myOKroutine;                      {                                              }
          kCancelButton:                      {User clicked Cancel button                    }
            myCancelRoutine;                  {                                              }
          {cases for other buttons}
        end;
      doMenu:                                 {User selected a menu…                         }
        MyMenuRoutine;
      doNothing:                              {No event available. If your app does any      }
        MyBackgroundRoutine;                  {  background processing, execute one “cycle.” }

      {cases for other events}
      otherwise                               {Ignore events that are not listed in the cases }
    end;
  end;
```

As you can see above, your application does not have to do much more than respond to a specific occurrence. When you are writing your event handler, you can pose the question: what does my application do when the user interacts with this item? See the tutorials for various examples of event handler code where the application responds to clicks in buttons, interaction with list boxes, and so on. When Tools Plus calls your event handler routine and the event pertains to a specific window (Event.Window is not zero), Tools Plus makes the target window the current window for your convenience.

Ideally, your event handler should execute its code rather quickly, especially in response to doNothing events. An ideal application finishes its event handler code is less than 1/60 of a second, or one tick. If your event handler takes a while to finish, your application will be perceived as a “CPU hog” and make other applications run sluggishly or in spurts and jumps. The Process1EventWhileBusy routine alleviates this problem during lengthy operations.

The Window Event Handler Routine

So far, we have discussed how to use a single event handler routine to handle events for your entire application. Your application installs its main event handler when it initializes Tools Plus (InitToolsPlus routine). When your application becomes more complex and has multiple windows, you will likely want to have a separate event handler routine that is customized for each window. This makes for small, tidy, and highly specialized window event handler routines. Tools Plus lets you install window event handler routines using the NewEventHandlerProc and SetWindowEventHandler routines.

Each window can have its own window event handler routine, share a common event handler with another window, or have no window event handler routine in which case its events are passed to the application’s main event handler. The

logic for determining which event handler routine is called by Tools Plus is simple:

- When the event does not pertain to a window (Event.Window is zero), your application's main event handler is called.
- When the event pertains to a specific window (Event.Window is not zero), and that window has an event handler installed, that window's event handler routine is called.
- When the event pertains to a specific window (Event.Window is not zero), and that window does not have an event handler installed, your application's main event handler is called.

See the end of this chapter for a list of all events that pertain to windows. That way, you'll know which events are sent to window event handlers, and which events are sent to the main (default) event handler.

Modal Event Handling

Your application handles events for modal windows in exactly the same manner as for modeless windows, and that by responding to those events in either your application's main event handler routine, or in a window event handler routine. Some developers who have come from traditional Macintosh toolbox development are used to calling a toolbox routine that opens a modal dialog, calls the appropriate event processing routines, and continues only when the user dismisses the modal dialog. A similar effect can be accomplished as follows:

- Open the modal dialog.
- Install a window event handler for the modal dialog.
- Your application resumes with the understanding that the dialog has been closed and all appropriate actions have been taken.

With the strategy outlined above, you write your modal dialog's window event handler to carry out the appropriate actions in response to the user's interaction rather than having the caller carry out those actions. Your modal dialog's window event handler also closes the dialog when it is done. For example, you may have a modal dialog whose actions include a "Cancel" button and a "Save Data" button. Your modal dialog's window event handler would respond to these buttons as follows:

- Cancel button: Close the modal dialog window.
- Save Data button: Save the data, and if no errors occurred, close the modal dialog window.

Some developers much prefer to write their code as a sequence of steps knowing that the following step will be executed only when the current step is completed, and that the "current step" could be the processing of a modal dialog. This coding style is consistent with traditional Macintosh coding in which the code calls a toolbox modal dialog routine that returns control to the caller only when the user dismisses the dialog. To do this, you can write your code as follows:

```
Open the modal dialog (let's say it is window #15 in this example)
Install a window event handler for the modal dialog (this step is optional)
while WindowIsOpen(15) do
    Process1EventWhileBusy({true or false})
Execute actions after the modal dialog is dismissed
```

In the pseudo code above, your event handler routine populates global variables based on the user's interactions, such as "which button was clicked." The event handler routine also closes the dialog when it is done, thereby signaling to WindowIsOpen(15) that it can continue to the next step.

Filtering Events (the Event Filter Routine)

By default, Tools Plus gets an event from the Macintosh's Event Manager, does all its internal processing, then only if it is necessary, reports the event to your application by calling your event handler routine. Advanced developers may want to have their application do some work *before* Tools Plus processes the event. An example is an application that has numerous menus whose items are constantly enabled and disabled as the user clicks on objects in a window. The developer, wanting to get the absolute maximum performance from his application, does not want to enable and disable sixty menu items in response to each mouse click in the window. Instead, the developer chooses to set boolean flags that correspond to each affected menu item. When the user clicks in the menu bar, the application correctly enables and disabled the sixty menu items before Tools Plus processes the mouse-down event and shows the user the correctly enabled or disabled menu items. In another example, the programmer may want to convert all straight

apostrophes (') to curly ones (’) as the user types.

Your application can do this if you write an *event filter*. An event filter lets your application inspect each toolbox event, optionally modify it, and optionally send it to Tools Plus for processing, or discard the event. If you are writing a plug-in where your plug-in is the master, this is where you can send events such as window updating events back to your host application instead of having them processed by Tools Plus. Your application installs its event filter when it initializes Tools Plus (InitToolsPlus routine). An event filter routine is written as follows:

C

```
pascal Boolean MyEventFilter (EventRecord *theEvent)
  {
    /* Inspect and possibly modify the toolbox event record */

    return(1);                                  /*Should Tools Plus process the event?         */
  }
```

Pascal

```
function MyEventFilter (var theEvent: EventRecord): Boolean;
  begin
    {Inspect and possibly modify the toolbox event record}

    MyEventFilter := true;                      {Should Tools Plus process the event?          }
  end;
```

If your event filter routine wants to pass the toolbox event to Tools Plus for processing, your event filter routine should return with a value of *true*, or 1 in C/C++. Returning with a value of *false*, or 0 in C/C++, indicates that Tools Plus should ignore the toolbox event because your event filter routine has either processed the event itself, or it wants to filter out that event.

You should be careful of creating multiply re-entrant code or recursive code when writing your event filter routine. If your event filter opens a Dynamic Alert, then you must realize that the event being analyzed by your event filter routine will not be reported to Tools Plus until *after* the Dynamic Alert is dismissed by the user. In essence, the Dynamic Alert is holding up the event from being reported. Furthermore, your event filter routine will be called while the Dynamic Alert is displayed, meaning that your event filter routine can be entered multiple times before exiting (i.e., it is multiply re-entrant). You need to account for this in some way.

You also need to be aware of doing any other event processing within the event filter routine, such as calling Tools Plus’s Process1EventWhileBusy routine or ProcessToolboxEvent routine, because they too will call your event filter routine to filter a toolbox event before passing it to Tools Plus. Similarly to the condition that was described earlier, Process1EventWhileBusy and ProcessToolboxEvent will delay the reporting of the event until after these routines have finished executing.

The easiest ways to avoid *potential* pitfalls when writing your event filter routine are as follows:

- Avoid calling Tools Plus’s AlertBox, AlertBox3, Process1EventWhileBusy and ProcessToolboxEvent routine.
- Avoid opening “Save As…” or “Open…” dialogs

In most cases, you can circumvent the above actions by setting a global flag within the event filter routine, then responding to that flag in your event handler routine.

Serial Events

Tools Plus does much more than just translating a toolbox event to a Tools Plus event. It can generate a number of Tools Plus events from a single toolbox event. This series of events is called “serial events,” and it represents some sort of sustained activity. A simple example of this is the toolbox’s update event that informs an ordinary Macintosh application that it needs to update a window. When Tools Plus detects a toolbox update event, it reports a doPreRefresh event to your event handler to tell it to draw any background elements. Tools Plus then refreshes its own elements such as buttons, scroll bars and list boxes, then it reports a doRefresh event to let your application refresh any elements after it has refreshed its own.

Another example of serial events can be seen when the user holds the mouse down in the line down region of a scroll bar. Event though the Macintosh’s toolbox reports only one mouse-down event, Tools Plus translates this into a stream of doScrollBar events for as long as the user holds the mouse button down and keeps the mouse in the line down region of the scroll bar. Picture buttons can also behave similarly.

Your application can determine if Tools Plus is generating serial events by calling GetTPSerialEvent which reports the type of serial event being generated, such as doScrollBar. Your application can also terminate serial events by calling KillTPSerialEvent. In the case of scroll bars and picture buttons generating serial events, calling KillTPSerialEvent has the equivalent effect of the user releasing the mouse button. Most applications will never need to use these two routines.

Tools Plus Event Codes

Each event reported by Tools Plus is identified by the first field of the global event record, Event.What. The Event.What field contains an event code that tells your application what to do with the event record's information. Constants are used to identify event codes as follows:

```
CONST                                      {Tools Plus event codes                   }
               doNothing      =   0;       {No event                                 }
               doChgWindow    =   1;       {User clicked in an inactive window       }
               doRefresh      =   2;       {A window has to be refreshed             }
               doGoAway       =   3;       {The close box was clicked                }
               doButton       =   4;       {Button was clicked                       }
               doMenu         =   5;       {Menu was selected                        }
               doKeyDown      =   6;       {A keyboard key was pressed               }
               doAutoKey      =   7;       {A keyboard key is auto-repeating         }
               doKeyUp        =   8;       {A keyboard key was released              }
               doClickToFocus =   9;       {Mouse clicked in inactive editing field  }
               doScrollBar    =  10;       {Mouse clicked in a scroll bar            }
               doListBox      =  11;       {Some sort of List Box activity           }
               doClick        =  12;       {Mouse click/drag [1..3]                  }
               doPopUpMenu    =  13;       {Pop-up menu was selected                 }
               doPictButton   =  14;       {Picture button activity                  }

               doClickControl = 101;       {Mouse clicked in a custom control        }
               doManualEvent  = 102;       {Manually processed events                }
               doMoveWindow   = 103;       {A window was moved by user               }
               doGrowWindow   = 104;       {A window was "grown" by user             }
               doClickDesk    = 105;       {Mouse clicked in the desk top            }
               doZoomWindow   = 106;       {Zoom box was clicked by user             }
               doSuspend      = 107;       {Application suspended (in background)    }
               doResume       = 108;       {Application resumed (now active appl.)   }
               doChgInField   = 109;       {Contents of active edit field was changed }
               doPreRefresh   = 110;       {A window may be refreshed before Tools   }
                                           {   Plus objects are drawn)               }
               doActivate     = 111;       {A window was activated                   }
               doDeactivate   = 112;       {A window was deactivated                 }
               doMoveCursor   = 113;       {Cursor has entered a new Cursor Zone     }
               doKeyInControl = 114;       {A keystroke was applied to an Appearance }
                                           {   Manager control                       }
               doChgMonitorSettings = 115; {Monitor settings were changed            }
               doTimer              = 120; {Timer event                              }
                                           {Apple Event from Finder or other apps:   }
               doOpenApplication    = 200; { · Your app was launched with no open docs }
               doOpenDocuments      = 201; { · Your app should open 1 or more docs   }
               doPrintDocuments     = 202; { · Your app should print 1 or more docs  }
               doQuitApplication    = 203; { · Your app should quit                  }
```

Event codes in the 100's will likely be ignored by most applications. Event codes in the 200's result from Tools Plus reporting an Apple Event to your application. If you install your own event handler for a specific Apple Event, the corresponding Tools Plus event will not be reported to your main event handler. All events are detailed later in this chapter, telling you how to respond to the event and which fields in the event record contain valid information.

Translating Toolbox events to Tools Plus events

Internally, Tools Plus gets low level events from the toolbox's Event Manager and translates them into high-level Tools Plus events that your application can use right away. This may seem like a simple translation on the surface, however, the recognition and processing of internal events is quite an extensive duty for Tools Plus. The table below describes this formidable task by listing the Event Manager's event, the internal processes that follow, and the Tools Plus event that finally reaches your application. Keep in mind that that some toolbox events are processed internally by Tools Plus and are never reported to your application, and your application may choose to ignore some events that are reported.

Toolbox Event	Conditions / Tools Plus's Internal Processing	Tools Plus Event
`nullEvent`	*none*	`doNothing` `doTimer`
`mouseDown`	If the watch cursor is displayed then the event is ignored (clicks in push buttons are optionally exempted)	
	Outside a modal window (beep)	
	In an inactive window that belongs to your application (does not apply to the tool bar or floating palettes which are always active)	`doChgWindow`
	In a floating palette that is not the frontmost palette, and is partially obscured by another palette. After refreshing the window, the mouse-down event is processed.	`doPreRefresh` `doRefresh`
	In a floating palette's title bar	
	In an inactive window that belongs to another application or desk accessory	`doSuspend`
	In Apple menu, except for "About…" item (selected item is opened or activated)	
	In Apple menu's "About…" item	`doMenu`
	Edit menu's Undo/Redo, Cut, Paste, or Clear item is selected for an active editing field in your application. Operation is done automatically. Your application is informed of the change.	`doChgInField`
	Edit menu's Copy item is selected for an active editing field in your application (operation is done automatically)	
	Edit menu's Undo, Cut, Copy, Paste, or Clear item is selected for an active editing field in your application (Operation is done automatically)	
	In other (pull-down or hierarchical) menu selection	`doMenu`
	In a pop-up menu	`doPopUpMenu`
	In an active desk accessory or other application (clicking, dragging, or closing, etc.)	
	Selecting or deselecting lines in a list box	`doListBox`
	In a button in the active window (only if mouse was released inside the button's area)	`doButton`
	In an Appearance Manager-savvy control in the active window, providing that the control only wants a mouse click to indicate that it assumed the keyboard focus, and the control is not tracked (such as the Clock) control.	
	Picture button in the active window: buttons with the "repeating events" option turned on produce events while the picture button is held down, as long as the button does not reach the end of its range. Those without the "repeating events" option produce an event only if mouse was released inside the button's area.	`doPictButton`
	Scroll bar in the active window (while in up arrow, down arrow, Page up region, or Page Down region, or if thumb was moved)	`doScrollBar`
	Active editing field, when setting a new insertion point or selection range (Edit menu's items are enabled/ disabled according to the insertion point or selection)	
	Inactive editing field or an item that wants the keyboard focus	`doClickToFocus`
	Single-click, double-click, triple-click, and/or dragging	`doClick`
	Active window is dragged by user	`doMoveWindow`
	Active window's size is changed by using the size box	`doGrowWindow`
	Active window's close box was clicked	`doGoAway`
	Active window's "zoom box" was clicked	`doZoomWindow`
	Custom control	`doClickControl`
	On the desk top	`doClickDesk`
`mouseUp`	End of a single-click, double-click, triple-click, and/or dragging	`doClick`
	All other mouse-up events	`doNothing`

Toolbox Event	Conditions / Tools Plus's Internal Processing	Tools Plus Event
`keyDown` `autoKey`	If the watch cursor is displayed then the event is ignored (except for ⌘-. which halts lengthy processes)	
	Command key invoking Edit menu's Undo/Redo, Cut, or Paste item in an active field in your application. Operation is done automatically. Your application is informed of the change.	`doChgInField`
	Command key invoking Edit menu's Copy item is selected for an active editing field in your application (operation is done automatically)	
	Command key invoking Edit menu's Undo, Cut, Copy, or Paste item in an active desk accessory under Finder (the editing operation is done automatically)	
	Command key invoking a menu	`doMenu`
	Command key not invoking a menu	`doKeyDown` or `doAutoKey`
	Enter or Return key invoking a default push-button	`doButton`
	Keystroke applied to an Appearance Manager list box resulting a in different line being selected	`doListBox`
	Keystroke applied to an Appearance Manager-savvy control such as the clock (the affected control has the keyboard focus)	`doKeyInControl`
	Active editing field processing keys (Edit menu's items are enabled/disabled according to the selection in the editing field. Undo item is changed according to field's contents, such as "Undo Typing")	`doChgInField`
	Other key strokes	`doKeyDown` or `doAutoKey`
`keyUp`	If the watch cursor is displayed then the event is ignored	
	Active editing field ignores key up events	
	Other key-ups, providing that SetEventMask has not masked out key up events	`doKeyUp`
`updateEvt`	An updateEvt for a Tools Plus window is reported as a doPreRefresh followed by a doRefresh event.	`doPreRefresh` `doRefresh`
	If a tool bar is open (though possibly hidden), and the user changes the main monitor's resolution or size, Tools Plus automatically resizes the tool bar to the width of the main monitor, and generates a doGrowWindow event for the tool bar.	`doGrowWindow`
	If a tool bar is open (though possibly hidden), and the user moves the menu bar to another monitor thereby changing its co-ordinates, Tools Plus generates a doMoveWindow event for the tool bar.	`doMoveWindow`
	An updateEvt for a window or dialog created by toolbox routines (not by Tools Plus) is reported as a doManual event. You will only need to take this into account if you create true dialogs such as custom "Open…" or "Save As…" dialogs, or if you create a plug-in. You can think of this as telling your application or plug-in that a foreign window need to be updated	`doManualEvent`
`activateEvt`	When a window is *deactivated*, the following happens automatically: [1] text in the active editing field is deselected and the insertion point is removed [2] list box lines are deselected [3] scroll bars are displayed using a "frame" for the control (a hollow control) [4] buttons and picture buttons are disabled [5] pop-up menus are disabled [6] if the window has a "grow box," it is hidden.	`doDeactivate`

	When a window is *activated*, the following happens automatically (objects are restored to their original state that existed prior to deactivating the window):
	[1] if the window needs to be refreshed, a doPreRefresh event is generated
	[2] text in the active editing field is selected or the insertion point is restored
	[3] list box lines are restored to their normal state
	[4] scroll bars are restored to their normal state
	[5] buttons and picture buttons are restored to their normal state
	[6] pop-up menus are restored to their normal state
	[7] if the window has a "grow box," it is displayed
	[8[Tools Plus's objects are refreshed
	[9] a doRefresh event is generated
		`doActivate`
		`(doPreRefresh)`
		`(doRefresh)`

Toolbox Event	Conditions / Tools Plus's Internal Processing	Tools Plus Event

`diskEvt`
`networkEvt`
`driverEvt`
`app1Evt`
`app2Evt`
`app3Evt`
	no internal processing	`doManualEvent`

`kHighLevel-Event`	No internal processing occurs if any of the following are true: • Your application's SIZE resource is set to *not* respond to High Level events, and it posted a High Level event. • Your application is running under System 6 or older, and it posted a High Level event. • Your application's SIZE resource is set to respond to High Level events, it is running under System 7 or later, but you have not defined an Apple Event handler for a specific Apple Event, and the event is not one of the four core events for which Tools Plus has a default event handler (i.e., "open application", "open documents", "print documents", and "quit application")	`doManualEvent`
	An Apple Event of class 'aevt' and ID 'oapp' (the core "open application" event) is detected, and you have not overridden this Apple Event with your own Event Handler routine.	`doOpen-Application`
	An Apple Event of class 'aevt' and ID 'odoc' (the core "open documents" event) is detected, and you have not overridden this Apple Event with your own Event Handler routine.	`doOpen-Documents`
	An Apple Event of class 'aevt' and ID 'pdoc' (the core "print documents" event) is detected, and you have not overridden this Apple Event with your own Event Handler routine.	`doPrint-Documents`
	An Apple Event of class 'aevt' and ID 'quit' (the core "quit application" event) is detected, and you have not overridden this Apple Event with your own Event Handler routine.	`doQuit-Application`
	If your application's SIZE resource is set to *not* respond to Suspend/Resume events, then no internal processing occurs.	`doManualEvent`
	If your application's SIZE resource is set to respond to Suspend/Resume events, then these event are reported to your application. In System 7 (or later), the osEvt representing the "mouse moved" event.	`doSuspend` or `doResume`
	When your application resumes: if a tool bar is open (though possibly hidden), and the user changes the main monitor's resolution or size, Tools Plus automatically resizes the tool bar to the width of the main monitor, and generates a doGrowWindow event for the tool bar.	`doGrowWindow`
	When your application resumes: if a tool bar is open (though possibly hidden), and the user moves the menu bar to another monitor thereby changing its co-ordinates, Tools Plus generates a doMoveWindow event for the tool bar.	`doMoveWindow`
	In System 7 (or later) the osEvt also carries the "mouse moved" status, thereby becoming a "mouse moved event." Tools Plus uses these events to automatically change the cursor's shape.	

Automatic Apple Event Support

By default, Tools Plus provides automatic support for all four required Apple Event as listed below:

Class	ID	Action
aevt	oapp	Open Application: Tools Plus reports a doOpenApplication event to your main event handler
aevt	odoc	Open Documents: Tools Plus reports a doOpenDocuments event to your main event handler
aevt	pdoc	Print Documents: Tools Plus reports a doPrintDocuments event to your main event handler
aevt	quit	Quit Application: Tools Plus reports a doQuitApplication event to your main event handler

An application that is written with Tools Plus does not have to be Apple Event aware, but we strongly recommend that you make it so, especially if it will run on Mac OS 8.5 or later. Users of Mac OS 8.5 and later can dynamically change the system font, small system font, and views font, and if you enable Apple Event processing, Tools Plus can automatically correct user interface discrepancies due to these changes.

To make your application Apple Event aware, use your development environment (except THINK Pascal users who must use a resource editor) to access your project's 'SIZE' resource settings, and set the "High level event aware" flag on. This enables your application to respond to Apple Events when it is running under System 7 or later. Apple Events are not available under System 6 or older (they weren't invented yet). Even so, Tools Plus still generates equivalent Tools Plus events for virtually all the required events.

As soon as your application starts processing events, it will always receive one of the following Apple Events: "open application", "open documents", or "print documents".

Tools Plus automatically processes the following Apple Events which indicate that the user has changed a font in the user interface:

Class	ID	Action
appr	sysf	System font was changed
appr	ssfn	Small system font was changed
appr	vfnt	Views font was changed
appr	thme	Theme was changed

The Apple Events listed above are generated exclusively as a result of the user making changes to one or more settings in the Appearance Manager. Tools Plus takes the appropriate corrective action when it detects these Apple Events, but it does not report them to your application unless you install an Apple Event Handler routine for the required event(s).

Tools Plus automatically processes Apple Events by (i) reporting an equivalent Tools Plus event as stated for all four required Apple Events this section, (ii) quietly applying corrective measures without informing your application as stated above, and (iii) dispatching the Apple Event to your own Apple Event Handler routine if you have written and installed one. Inside Macintosh details how to write an Apple Event Handler routine, and how to install it in an application. The same rules apply in Tools Plus. You can provide extended support for as many Apple Events as you want. You can also override Tools Plus's default support for any or all of the four required Apple Events by installing your own Apple Event Handler routine(s).

Simulated Apple Event Support

Tools Plus accounts for applications (not plug-ins) that are not Apple Event aware, and for those running under System 6 or older in which Apple Event are not available. Under such conditions, Tools Plus's ProcessEvents routine generates the following synthesized events before it starts processing real events:

doOpenApplication

Who gets it .... :
- 680x0 and PowerPC applications that are not Apple Event aware
- 680x0 applications running under System 6 or older

Under what conditions .. :
- As soon as your application starts processing events by calling the ProcessEvents routine. This happens only if the Finder does not want your application to open or print any documents.

doOpenDocuments

Who gets it .... :
- 680x0 applications that are not Apple Event aware
- 680x0 applications running under System 6 or older

Under what conditions .. : Before your application is launched, any of the following may occur…
- The user selects one or more of your application's documents, then selects the File menu's Open item from the Finder
- The user double-clicks one of your application's documents
- The user selects one or more of your application's documents then double-clicks one
- The user drags one or more documents that your application can open, and drops them on your application

doPrintDocuments	Who gets it .... :	• 680x0 applications that are not Apple Event aware • 680x0 applications running under System 6 or older
	Under what conditions .. :	• The user selects one or more of your application's documents, then selects the File menu's Print item from the Finder
doQuitApplication	Who gets it .... :	• 680x0 applications that are not Apple Event aware • 680x0 applications running under System 6 or older
	Under what conditions .. :	• Following the completion of your application's response to a doPrintDocuments event.

680x0 applications that are not Apple Event aware, or those that run on System 6 or older, can receive their "Open Document" instructions from the Finder after the application is launched, but from the your perspective of coding your application, it's as though the user opened a document by selecting your application's File menu's "Open…" item. See the Completing Your Application chapter for details. It is strongly recommended that applications running on System 7 or later be Apple Event aware and take advantage of Apple Event services.

Routines for Handling and Processing Events

The following routines are used to process events, or to assist in their processing. This includes everything from creating event handler UPPs, to starting Tools Plus's event-processing services, to parsing an Apple Event list into file specifications.

SetWindowEventHandler

Set an event handler routine for a window.

C
```
pascal void SetWindowEventHandler (short Window,
            EventHandlerUPP EventHandler);
```

Pascal
```
procedure SetWindowEventHandler (Window: INTEGER;
            EventHandler: EventHandlerUPP);
```

SetWindowEventHandler sets a routine that is called by Tools Plus to handle an event for a specific window. Each window can have its own event handler routine, or several windows can share the same routine. If a window does not have an event handler routine, then it uses the application's event handler.

Window specifies the window number to which the event handler is attached. If the specified window is not open, SetWindowEventHandler does nothing.

EventHandler is a UPP to the event handler routine that is called by Tools Plus to respond to an event in the window. See the NewEventHandlerProc routine for details about UPPs and how to create them. If you are writing a 680x0 application and the source code will never be compiled to generated PowerPC native code, you can specify the address of the event handler routine as follows in C/C++:

```
SetWindowEventHandler(5, myEventHandler);
```

In Pascal, a similar statement is used except the "@" symbol indicates the address of a routine which is the same thing as a pointer to a routine:

```
SetWindowEventHandler(5, @myEventHandler);
```

Also see: NewEventHandlerProc

NewEventHandlerProc

Create a UPP for an event handler routine.

C
```
pascal EventHandlerUPP NewEventHandlerProc (ProcPtr userRoutine);
```

Pascal
```
function NewEventHandlerProc (userRoutine: ProcPtr): EventHandlerUPP;
```

The EventHandlerUPP type is a Universal Procedure Pointer used for consistency across all interfaces (C/C++ and Pascal using the original Apple interfaces or the newer universal interfaces required for PowerMacs). In 680x0 applications, the EventHandlerUPP is nothing more than a ProcPtr, or a pointer to a Pascal routine. In PowerMac applications, the EventHandlerUPP is a pointer to a structure that is allocated using the NewEventHandlerProc routine. If you are writing a PowerMac application, or if your source code will compile to both 680x0 and PowerMac-native code, you need to use the new universal headers (or universal interfaces for Pascal) and do the following to ensure that your source code compiles and executes correctly when running under a 680x0 or PowerPC processort:

1. Create a global variable for each event handler routine you will use throughout your application. If you are using the same event handler routine for several windows, all the windows can share a single global variable. Declare the variable as an EventHandlerUPP type. In 680x0 applications, this variable is used as a pointer to an event handler routine. In PowerMac applications, it is used as a pointer to a universal procedure structure. In this example, define a global variable named myEventHandlerUPP of type EventHandlerUPP.
2. Populate myEventHandlerUPP so that it points to your event handler routine. In this example, the event handler routine is named myEventHandler. In C/C++, the code looks like this:
```
myEventHandlerUPP = NewEventHandlerProc(myEventHandler);
```
   In Pascal, the code is identical except the "@" symbol indicated the address of a routine:
```
myEventHandlerUPP := NewEventHandlerProc(@myEventHandler);
```
   Do this very early in your application, likely immediately after calling InitToolsPlus, because you are creating a non-relocatable structure, and allocating it early prevents memory fragmentation.
3. After you create your window, you can associate the event handler routine with the window using the following code. This example assumes the event handler routine is being installed into window number 5:
```
SetWindowEventHandler(5, myEventHandlerUPP);
```

The definition and writing of the event handler routine is detailed earlier in this chapter under "The Event Handler Routine." If you want to deallocate the UPP for an event handler routine in a PowerMac application or plug-in, use the toolbox's DisposeRoutineDescriptor routine. PowerMac plug-ins will certainly want to do this as part of their quitting logic along with calling DeinitToolsPlus.

ProcessEvents

Process events continuously.

C
```
pascal void ProcessEvents (void);
```

Pascal
```
procedure ProcessEvents;
```

Your application calls the ProcessEvents routine once in its main routine. Once called, ProcessEvents continuously gets events from the Macintosh toolbox's Event Manager, processes them, and calls your event handler routine(s) as required. When your application signals that it wants to quit by calling QuitToolsPlus, likely in response to the user selecting the File menu's Quit item or in response to a doQuitApplication event, ProcessEvents stops processing events and returns control to your application.

Under normal circumstances, those being when your Apple Event aware application is running under System 7 or later, Tools Plus starts off by reporting one of the following Apple Events: "open application", "open documents", or "print documents". If your application (not a plug-in) is not Apple Event aware, or if it is running under System 6 or older in which Apple Events are not available, the ProcessEvents routine can synthesize these core Apple Events. These synthesized events are reported just after your application's startup to let you know if your application was launched without any open files, with files that need to be opened, or with files that need to be printed. Tools Plus reports these as doOpenApplication, doOpenDocuments, and doPrintDocuments events.

Process1EventWhileBusy

Process a single event while the application is busy.

C
```
pascal void Process1EventWhileBusy (Boolean skipNullEvents);
```

Pascal
```
procedure Process1EventWhileBusy (skipNullEvents: BOOLEAN);
```

Due to the nature of Mac OS, Tools Plus does all its multitasking when your application's event handler routine finishes executing. Unfortunately, your event handler may call a routine that runs a long time, such as a printing routine or a sorting routine. When your application is busy with a lengthy process, it should periodically call the

Process1EventWhileBusy routine to briefly allow some processing time to other applications, and to process a single event within your own application, such as the user typing ⌘-. to halt the lengthy process.

Internally, Process1EventWhileBusy is exactly like the ProcessEvents routine except that is only gets one event from the Event Manager, processes it, and returns to your application whereas ProcessEvents continuously gets and processes events. Ideally, your application should call Process1EventWhileBusy about 60 times or more per second. At 20 times per second, the user will begin to notice very minor delays.

The *skipNullEvents* flag is used to indicate is your event handler should be called when a doNothing event is generated. This may be an issue if your application is calling Process1EventWhileBusy from within your main event handler's doNothing logic, in which case it is possible to call Process1EventWhileBusy recursively and exhaust your application's stack. When the *skipNullEvents* flag is set to *true* (the default), Tools Plus will not call your event handler if a doNothing event is generated as a result of calling Process1EventWhileBusy. If the *skipNullEvents* flag is set to *false*, your main event handler may be called with a doNothing event.

☞ **Warning**: Be careful to prevent situations in which Process1EventWhileBusy is called recursively. For example, your main event handler's doNothing case executes a length process that periodically calls Process1EventWhileBusy(false), and the Process1EventWhileBusy routine calls your main event handler with a doNothing event which causes it to call Process1EventWhileBusy(false) again. This condition will rapidly exhaust your application's stack and crash your application.

ProcessToolboxEvent

Process a single toolbox event.

C
```
pascal void ProcessToolboxEvent (EventRecord *theEvent);
```

Pascal
```
procedure ProcessToolboxEvent (theEvent: EventRecord);
```

This routine processes a single toolbox event, *theEvent*, which was obtained by some other means. Most applications never need to use this routine. Plug-ins, on the other hand, may get their events from their host application and use ProcessToolboxEvent to apply an event to the Tools Plus-built plug-in.

ProcessToolboxEvent returns to your application when it has finished processing the toolbox event. Realize that this may not be immediate because ProcessToolboxEvent may be busy tracking a control or dragging a window for a while.

SetEventError

Set an event error code in response to an Apple Event. For applications only (not plug-ins).

C
```
pascal void SetEventError (short ErrNum, const Str255 ErrStr);
```

Pascal
```
procedure SetEventError (ErrNum: INTEGER; ErrStr: STRING);
```

You can use the SetEventError routine only in response to the following events: doOpenApplication, doOpenDocuments, doPrintDocuments, and doQuitApplication. SetEventError provides a reply to the sender of an Apple Event to inform the sender that your application could not respond appropriately. An example is when your application receives a doOpenDocuments event and it determines that an error occurred while reading the file. Using SetEventError has no effect if your application is responding to a custom Apple Event handler routine that replaces Tools Plus's default Apple Event handling.

ErrNum is an Apple Event error number that is sent back to the caller. Typically, it is the errAEEventNotHandled error (-1708).

ErrStr is an optional string describing the error. If you specify a non-empty string and an error code of zero (0), SetEventError overrides *ErrNum* with a value of errAEEventNotHandled error (-1708).

CountNumberOfFiles

Determine the number of documents that need to be opened or printed. For applications only (not plug-ins).

C
```
pascal long CountNumberOfFiles (void);
```

Pascal
```
function CountNumberOfFiles: LONGINT;
```

CountNumberOfFiles reports the number of files that your application is asked to open or print. Use this routine only in response to a doOpenDocuments event or a doPrintDocuments event. It returns a value of zero (0) at all other times. When your main event handler responds to a doOpenDocuments event or a doPrintDocuments event, it should run a loop from 1 to the value of CountNumberOfFiles, and in that loop, use GetIndexFileFSS or GetIndexFile to get information about each file in the list, thereby allowing you to open each file.

Also see: GetIndexFileFSS and GetIndexFile.

GetIndexFile

Retrieve file information for a file that needs to be opened or printed. For 680x0 applications only (not plug-ins).

C
```
pascal Boolean GetIndexFile (long Index, AppFile *FileInfo);
```

Pascal
```
function GetIndexFile (Index: LONGINT; var FileInfo: AppFile): BOOLEAN;
```

When your 680x0 Apple Event unaware application gets a doOpenDocuments event or a doPrintDocuments event, it first calls CountNumberOfFiles to determine the number of documents it is being asked to open or print. It then uses GetIndexFile to obtain information about a specific file before opening it. For Apple Event aware applications running on System 7 or later, and for all PowerPC applications, use the more advanced GetIndexFileFSS routine instead of GetIndexFile.

Index specifies the relative file number for which you want to obtain information. The value of *Index* must be in the range of 1 through the limit returned by the CountNumberOfFiles routine.

FileInfo returns file information that your application needs to open the file. You can use this record's values in a toolbox routine such as FSOpen to open the file's data fork, or OpenRF to open the file's resource fork. See Inside Macintosh's File Manager chapter for details on opening, reading, writing, and sharing files.

GetIndexFile returns with a value of *true* if it successfully retrieves information for the specified file. It returns with a value of *false* if your *Index* has an invalid value, or if you use the GetIndexFile routine at any time other than in response to a doOpenDocuments event or a doPrintDocuments event. In this case, *FileInfo* is zeroed out and should not be used.

Also see: GetIndexFileFSS.

GetIndexFileFSS

Retrieve file information for a file that needs to be opened or printed. For applications running on System 7 or later only (not plug-ins).

C
```
pascal Boolean GetIndexFileFSS (long Index, FSSpec *FSS);
```

Pascal
```
function GetIndexFileFSS (Index: LONGINT; var FSS: FSSpec): BOOLEAN;
```

When your application gets a doOpenDocuments event or a doPrintDocuments event, it first calls CountNumberOfFiles to determine the number of documents it is being asked to open or print. It then uses GetIndexFileFSS to obtain information about a specific file before opening it. This routine works on System 7 or later, on all 680x0 applications (Apple Event aware or not), and on Apple Event aware PowerPC applications. If your 680x0 application is Apple Event unaware or it is running on System 6 or older, use the older GetIndexFile routine instead of GetIndexFileFSS.

Index specifies the relative file number for which you want to obtain information. The value of *Index* must be in the range of 1 through the limit returned by the CountNumberOfFiles routine.

FSS returns file information that your application needs to open the file. The FSSpec is available only in System 7 or later. You can use this record's values in a toolbox routine such as FSpOpenDF to open the file's data fork, or FSpOpenRF to open the file's resource fork. You can also get information about the file type and other details from the toolbox's FSpGetFInfo routine. See Inside Macintosh's File Manager chapter for details on opening, reading, writing, and sharing files.

GetIndexFileFSS returns with a value of *true* if it successfully retrieves information for the specified file. It returns with a value of *false* if your *Index* has an invalid value, if your application is running under System 6 or older, or if you use the GetIndexFileFSS routine at any time other than in response to a doOpenDocuments event or a doPrintDocuments event. In this case, *FSS* is zeroed out and should not be used.

Also see: GetIndexFile.

QuitToolsPlus

Stop processing events and return control to the main application.

C
```
pascal void QuitToolsPlus (void);
```

Pascal
```
procedure QuitToolsPlus;
```

Your application calls QuitToolsPlus to signal that it wants to quit, likely in response to the user selecting the File menu's Quit item, or in response to a doQuitApplication Apple Event. When your application calls QuitToolsPlus, the ProcessEvents routine (in your application's main routine) stops processing events and allows your application to continue to its own "exit application" code. All other event processing routines are not affected by QuitToolsPlus. They are:

- ProcessToolboxEvent
- Process1EventWhileBusy
- AlertBox
- AlertBox3

ToolsPlusIsQuitting

Determine if Tools Plus has been instructed to stop processing events.

C
```
pascal Boolean ToolsPlusIsQuitting (void);
```

Pascal
```
function ToolsPlusIsQuitting: BOOLEAN;
```

ToolsPlusIsQuitting is used to determine if the QuitToolsPlus routine was called. It can be used throughout your application as a global flag to terminate loops and bypass certain logic if the application is in the process of quitting.

SetNullTime

Set the number of ticks (1/60 sec) your application can wait between receiving a doNothing event and can perform one cycle of its background process. Optionally set event interleave to maximize performance.

C
```
pascal void SetNullTime (long ActiveTime; long SuspendedTime);
```

Pascal
```
procedure SetNullTime (ActiveTime, SuspendedTime: LONGINT);
```

ActiveTime is the amount of time (in 1/60 second "ticks") your application can wait between receiving doNothing events while it is the active application. You can specify a value in the range of 0 to 15. A value of 0 provides maximum background processing speed for your application at the expense of other applications' background processing capability. A value of 15 provides maximum benefit to other applications by allotting minimum time for background processes in your application. If your application does not do anything in response to a doNothing event, as is the case with most application, you can safely be a good citizen by specifying a value of 2 or 3. At this setting, your application gets a doNothing event every 2 or 3 ticks (20 or 30 times per second) which is more than enough for Tools Plus to do its own administration such as keeping a field's insertion point flashing at a regular rate, or animating controls.

When ActiveTime is set to zero, your application gets plenty of background processing time at the expense of suspended applications and system extensions. If your application demands even more processing speed, specify a negative value for the ActiveTime parameter. A value of -1 establishes an event interleave such that your application gets an event (and surrenders processor time) only once per tick (60 times per second). This can result in a dramatic performance improvement. Values of -2 and -3 can also be specified to get an event every two or three ticks. This will give your application ultimate processing power short of turning off system extensions, but you may notice other processes slow down.

SuspendedTime is the amount of time your application can wait between receiving doNothing events while it is suspended (i.e., when another application is active under MultiFinder or System 7 or later).

Tools Plus is initialized with ActiveTime and SuspendedTime set to 0, meaning your application will have high performance at the expense of other applications and processes. Normally, applications that do background processing will set ActiveTime to 0 or 1 and set SuspendedTime to a higher value, thus slowing down the background processing when your application is suspended. If your application doesn't do any background processing, use the maxNullTime constant for the SuspendedTime parameter to allow other applications and process the maximum utilization of the processor.

Your application can change the values of ActiveTime and SuspendedTime at any time.

Also see: WaitAvail.

```
CONST                                {Background process scheduling          }
               maxNullTime = $7FFFFFFF; {Infinite time between doNothing events  }
```

WaitAvail

Determine if the Macintosh running your application supports scheduled processing. This determines if SetNullTime's settings will have any effect on your application.

C
```
pascal Boolean WaitAvail (void);
```

Pascal
```
function WaitAvail: BOOLEAN;
```

This routine returns a value of *true* if scheduled background processing is supported, or *false* if it is not. If the Macintosh running your application doesn't support scheduled background processing, then SetNullTime's settings will have no effect, and your application will receive a doNothing events as frequently as the processor can handle it. Internally, this routine reports if the WaitNextEvent trap is available to your application. WaitNextEvent has been available since later versions of System 6, and all versions of System 7 and later.

WaitForMultiClicks

Wait or don't wait for subsequent mouse events to detect a single, double or triple-click.

C
```
pascal void WaitForMultiClicks (Boolean WaitFlag);
```

Pascal
```
procedure WaitForMultiClicks (WaitFlag: BOOLEAN);
```

WaitFlag specifies if waiting for subsequent mouse-down and mouse-up events is turned on or off. You can use the constants *on* and *off* for this purpose.

Tools Plus reports single-clicks, double-clicks, and triple-clicks, as well as dragging that occurs between a mouse-down and mouse-up event. Normally, it reports mouse-down and mouse-up events as they occur, so if the user double-clicked the window's content region, your application would receive the following mouse event codes:

inClick1Drag	single-click started (mouse still down)
inClick1	single-click completed
inClick2Drag	double-click started (mouse still down)
inClick2	double-click completed

When you use WaitForMultiClicks(true), Tools Plus reports an event only after the user completes a single, double, or triple-click. At the end of a double-click, for example, your application would receive an inClick2 event (double-click completed). Although this is a convenient feature, you may not like the brief delay that is experienced while Tools Plus determines when a click is completed.

ResetMouseClicks

Discontinue a mouse's "drag" or multiple clicks in progress.

C
```
pascal void ResetMouseClicks (void);
```

Pascal
```
procedure ResetMouseClicks;
```

Tools Plus reports single-clicks, double-clicks, and triple-clicks, as well as dragging that occurs between a mouse-down and mouse-up event. In some applications, it may suffice to know that the user pressed the mouse button with the cursor being within a specific region or cursor zone, without concern for the mouse button's release. Alternatively, an application may allow a single-click only, thereby disallowing double or triple clicks.

In such cases, the ResetMouseClicks routine can be used to tell Tools Plus that sufficient mouse information has been obtained, and to reset the click and drag mechanism. This reset will clear the current click or drag from Tools Plus's event queue. The next time the mouse button in pressed down, it will be considered to be a first mouse-down of either a single-click, double-click, triple-click, or drag.

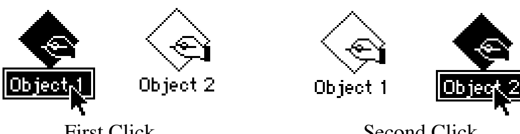
IgnoreFirstMouseClick

Ignore the first click of a multiple click sequence.

C
```
pascal void IgnoreFirstMouseClick (void);
```

Pascal
```
procedure IgnoreFirstMouseClick;
```

Tools Plus reports single-clicks, double-clicks, and triple-clicks, as well as dragging which occurs between a mouse-down and mouse-up event. Applications typically select an object when it is first clicked, then open or activate that object when it is double-clicked. If your application detects a double-click (in a doClick event), and it determines that both clicks did not occur on the same objects (thereby not constituting a double-click), you may elect to discard the first click after it has been processed, then wait for the next double-click to determine if the object should be opened.



In the example above, the user rapidly clicks Object 1 then Object 2. The first click is used to select Object 1. The second click is used to select Object 2. Even though the second mouse-down event occurred quickly enough to be a double-click, it occurred on a different object. When your application gets the second click, you can use IgnoreFirstMouseClick to discard the first click's information and to re-establish the clicking of Object 2 as the first click. If the user clicks Object 2 again quickly enough, your application will receive a double-click in Object 2.

IgnoreFirstMouseClick can be used in response to any doClick variation, regardless of whether the first, second, or third mouse-down event has been detected, or whether the mouse button is up or down.

ApplicationSuspended

Determine if your application has been suspended (i.e., not the active application).

C
```
pascal Boolean ApplicationSuspended (void);
```

Pascal
```
function ApplicationSuspended: BOOLEAN;
```

This routine returns a value of *true* if a desk accessory or another application is active under MultiFinder or System 7 (or later). A value of *false* is returned if your application is active, even though your application may not have any windows open.

When running under MultiFinder or System 7 (or later), your application needs to have a SIZE resource with the "AcceptSuspendEvents" bit set to "1." See the SIZE resource for details.

GetTPSerialEvent

Determine if Tools Plus is processing a series of events. (For advanced event processing only.)

C
```
pascal short GetTPSerialEvent (void);
```

Pascal
```
function GetTPSerialEvent: INTEGER;
```

GetTPSerialEvent is typically used only by applications that need to examine, filter or alter events before allowing Tools Plus to process them. Tools Plus can generate a series of Tools Plus events in response to a single toolbox event obtained from the Event Manager.

GetTPSerialEvent returns with a value that represents a task that is responsible for generating multiple Tools Plus events. The following constants can be used:

```
CONST                                   {Tasks that can generate multiple events:            }
               none         =   0;   {No "serial event" task is in progress             }
               doChgWindow  =   1;   {User clicked on an inactive floating window       }
               doClick      =   9;   {Single/double/triple click/drag in progress       }
               doScrollBar  =  10;   {User is interacting with a scroll bar             }
               doPictButton =  14;   {User is interacting with a picture button         }
               doPreRefresh = 110;   {Window needs refreshing (doPreRefresh / doRefresh) }
               doActivate   = 111;   {Window is being activated and refreshed           }
```

KillTPSerialEvent

Terminate Tools Plus's generation of a series of events. (For advanced event processing only)

C
```
pascal void KillTPSerialEvent (void);
```

Pascal
```
procedure KillTPSerialEvent;
```

Tools Plus can generate a series of Tools Plus events in response to a single event obtained from the Event Manager, as shown in the GetTPSerialEvent routine. An example of this is a single mouse-down event from the Event Manager being used to generate a series of timed doScrollBar events as the user holds the mouse down over a scroll bar's up button.

KillTPSerialEvent terminates the operation that is generating a series of events. In the case of a doScrollBar event, it behaves as though the user released the mouse button and stopped interacting with the scroll bar.

Also see: GetTPSerialEvent.

Timers and Timer Events

Tools Plus incorporates a sophisticated Timer that is very simple to use even though it provides considerable versatility. Even though a Timer is easy to use, we recommend that you read this entire section on Timers if you are in the least bit interested in the accuracy of Timer events, or in factors that may influence the regularity with which your Timer events are reported. The following are some reasons you may want to use a Timer:

- A much more controlled way of doing background processing than waiting for null events, which may not be available frequently enough or regularly enough.
- Allows you to have thousands of timed processes running while being efficient with memory and the CPU.
- Lets you easily manage many timed processes. Doing so manually would be complex.

The following are some examples of when you may want to use a Timer:

- A display object that updates regularly, like a clock.
- A cancel button on a log-on screen that is automatically selected after thirty seconds of inactivity to prevent the user's name from being displayed too long or left up accidentally while the computer is unattended.
- Automatically terminating a server session after ten minutes of inactivity.
- Triggering animation frames.
- An appointment application that reminds you of bookings with a dialog, scrolling banner and or sound.
- A "remind me later" feature that triggers once after a preset time.
- Flash any object, such as a static text item like "Initializing database. Process cannot be canceled."
- Virtually any background process, like searching or sorting a database.
- Virtually any timed or periodic process.

To start a Timer, call the NewTimer routine and supply it with the parameters that detail the nature of the Timer's behavior. Each Timer is referenced using a unique Timer number. This number is specified when the Timer is created, and refers to the specific Timer until it is deleted. After a Timer is created, it will report one or more events (doTimer events) to your application.

Normally, all Timer events are reported to your application's main event handler routine. You can optionally specify a window number for a Timer, and that Timer will generate events that report the specified window number, thereby routing the events to your window's event handler routine. The most convenient and elegant solution is to write an event handler routine specifically for a Timer. This way, the Timer just calls your specialized event handler routine when it needs to.

One of the parameters that are used to create a Timer is its *value*, which can be expressed as either "events per unit of time" (frequency), or "time between events" (period). A Timer's frequency can be expressed as events per tick, events per second, events per minute, events per hour, or events per day. Alternatively, a Timer's period can be expressed as ticks between events, seconds between events, minutes between events, hours between events, or days between events. An *initial delay* can optionally be specified to tell the Timer that it should remain dormant for a specified time, then start reporting events regularly after the initial delay has lapsed.

Timers can optionally be synchronized to other Timers such that two or more Timers will always report their events the same time apart. This is useful, for example, when the first event hides an object and the following one turns it back on to produce a flashing object. You can synchronize many Timers to a single parent Timer.

Timers are automatically deleted when your application quits, or when you call DeinitToolsPlus. If you assigned a Timer to a window, that Timer is deleted when the window is closed, or optionally, when the window is hidden. You can manually delete a Timer with the DeleteTimer routine, which also deletes all "child" Timers that are synchronized to that master. Timers can also be deleted automatically after generating a single event. These are called "one shot" Timers.

If you are running a faceless process such as a sorting routine, you may want to consider writing a multi-threaded application. This takes more effort than using Timers, but you will likely get better performance. Please consult the relevant document (not this User Manual) for information on how to write a multi-threaded application.

How Tools Plus Generates Timer Events

In order to maintain the highest accuracy, Timers have the highest priority in the event queue. When a Timer is due to generate a doTimer event, Tools Plus does the following:

- Memorizes the current time
- Determines the oldest (created first) Timer that needs to report an event at this time
- Calls your event handler routine to report a doTimer event
- Waits for your event handler routine to finish processing
- Calculates the Timer's next event time
- Looks for other Timers that need to report an event at the memorized time, and reports those events. The sequence of Timers is always from oldest to newest.
- Deletes all one shot Timers that have reported an event
- Proceeds with normal Tools Plus event processing by getting an event from the toolbox's Event Manager and processing it.

To the user, it will appear that your application is doing many things at the same time when in fact, it is only cycling between those things, doing only one thing at a time. As a programmer, it is important to remember that a Timer event, just like any other event, can only be reported when your event handler routine has finished processing and has returned control to Tools Plus. The exception to this is if your event handler routine is busy for a while and it calls the Process1EventWhileBusy routine, which may generate additional events.

More sophisticated timing mechanisms can be implemented in Macintosh by other means, all of which require you to write your application in a fundamentally different and more complex manner. Tools Plus has opted for the much simpler approach that lets you get very respectable functionality and performance with very little effort or programming considerations. We reason that developers who need the ultra high performance, precision, or true multitasking, will have the skills necessary to implement those features, and to revised their application to handle the complexities that are introduced when implementing those features.

Timer Accuracy

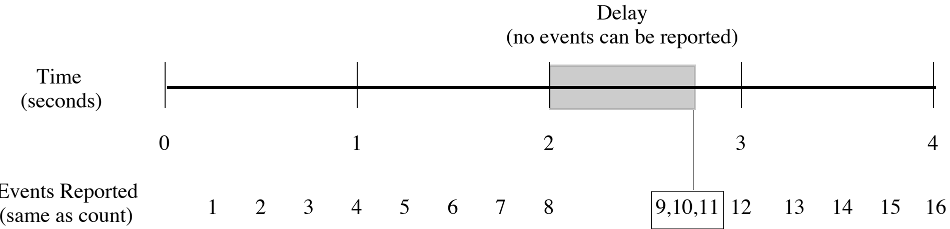
When asking how accurate Tools Plus's Timers are, the answer ranges from "very accurate" to "it depends." The things that influence a Timer's ability to report an event on time are:

- Calling a routine that takes a long time to complete. Timers can't report their events until your routines finish executing and control is returned to Tools Plus. The solution is to call Process1EventWhileBusy if your application is busy for any length of time.
- Event handlers that take a long time to complete. Ideally, an event handler should finish its work in 1/60 of a second or less.
- Natural processes, like Tools Plus refreshing controls and user interface objects in a window. This may take a few seconds in a complex window.
- Reading and writing large blocks of data to and from disk.
- Reading and writing to slow media, like a floppy disk, CD-ROM, or through a LocalTalk network.
- Tracking a control, such as the time between a mouse-down and mouse-up in a control.
- Selecting a pull-down, hierarchical, or pop-up menu.
- Appearance Manager controls that need idling, like the clock control and the busy progress indicator (the "barber pole" thermometer)
- Many Timers trying to report events at the same time (the next Timer can only report its event after the previously invoked event handler routine completes).
- Other applications or processes that are poor citizens and do not share the processor enough.

As you can see, the things that conspire to prevent a Timer from reporting an event on time are the same things that affect all other aspects of you application's performance. It's just that Timers are more time critical, and therefore, you notice it more when they are affected. This will be the nature of Macintosh until Mac OS becomes a fully preemptive multitasking system some time in the future, likely with Mac OS X.

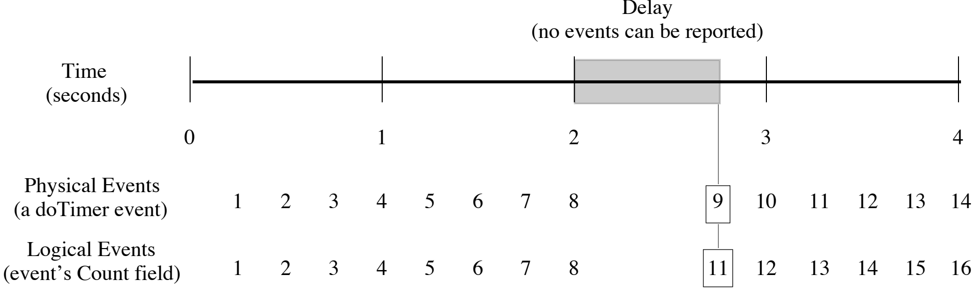
Tools Plus addresses the issue of timing accuracy by letting you choose the relationship between a physical Timer event and logical Timer event. Physical Timer events are the actual events that are reported to your application. Each time a doTimer event is reported, that's a physical event. Logical events are the number of times a Timer event *should*

have been reported. This is reported with each physical Timer event as the *count* field in the event record. In the example below, a Timer is supposed to report 4 events per second. A delay, perhaps caused by some poorly behaved application that holds onto the processor for too long, holds up event processing in your application for almost a second, during which time no Timer events can be reported. As soon as your application is able, it receives a rapid series of Timer events to catch up to where it should be.



In the example above, the delay introduced at around the two second mark prevents any events from being reported. The delay lasts about three quarters of a second, during which time events 9 and 10 *should* have been reported. When your application gets its first chance, events 9 and 10 are reported (the ones that should have been reported during the delay), followed by event 11 that is now due. At this point, the Timer has caught up with your application, and it resumes the regular reporting of events, the next of which is event 12 at the 3 second mark.

Certain kinds of timed processes, such as movies or animation that need to be played at a precise frame rate, need to know where they *should* be along the time line rather than how many events have passed. A movie is like this because if the Macintosh experiences a brief inability to draw all the frames, it wants to jump to the correct frame as soon as it can rather than "fast forwarding" to get to where it should be. The example below is similar to the previous one, except that we have allowed the Timer to report where it should be if it cannot deliver events on time, rather than sending a series of events in an attempt to catch up.



In the example above, the "Physical Events" line indicates when events are actually reported to your application. The "Logical Events" line indicates the event number that *should* have been reported (the Event.Timer.Count field of the event record). As you notice, right up to the two second mark, the physical events and logical events are matching. Each time your application should get an event, it did. At the two second mark, a delay is introduced during which time events 9 and 10 should have been reported but could not. As soon as your application is able, the Timer reports and event. In this example, a Timer event just happens to be due, so the Timer reports the 9th physical Timer event. Logically, the Timer should be on the 11th event, as indicated by the event record's Event.Timer.Count field.

This relationship between physical and logical events is the default behavior for a Timer. It is also the safer behavior, especially if there is a chance that the amount of work that needs to be done in response to each Timer event may take longer to carry out than when the next Timer event is due.

Timer Resolution

The Tools Plus Timer's highest resolution is 1/60 of a second (a "Tick"). This means that the Timer uses the Macintosh's internal mechanisms (the TickCount routine) to determine if it is time to report a Timer event or not. As you would expect, Timers honor Apple's recommendations regarding the use of the TickCount routine. Even though a Timer's highest resolution is 1/60 of a second, it does not mean that a Timer cannot generate more than 60 events per second. Rather, a Timer checks itself once per Tick, or 60 times per second, and reports the correct number of events that are needed to keep in on track. The example below illustrates a Timer that is running at 90 events per second, or

one and a half events per tick.

As you can see in the example above, the Timer's highest resolution of 1/60 second means that the Timer can only check to see if it is time to report an event once per tick, or 60 times per second. When the Timer does check, it periodically determines that it needs to report two events since the last time it reported an event, about one tick ago. The timer then reports two events, one after another, to catch up to where it should be. You can see this behavior at the 1 and 3 tick marks. After the 4th tick has elapsed, a total of 6 events have been reported for an average of one and a half events every tick, or 90 events per second, our originally specified frequency.

Timers and doNothing (null) Events

Due to architectural constraints of Mac OS, a doTimer event will often be followed by a doNothing or "null" event. You only need to be aware of this insofar as your application will likely receive doNothing events more frequently than it expects when a Timer is used. You may decide to have your event handler routine not respond to doNothing events (i.e., not have a case for the doNothing event in your event handler) and instead, use a Timer to invoke background processes that would normally be run in response to a doNothing event.

If a Timer is used, also be aware that any event time stamp (the myEvent.Event.when field) indicates when the event was generated by the Event Manager, and not when it was reported to your application. Timers actually "jump into" the event queue and are reported before any toolbox event is reported.

The Possibility of a Timer Overflow

A Timer uses long integer precision (a 32-bit signed integer) to keep track of its next event time, and the number of events reported (the Event.Timer.Count field of the event record). In virtually all cases, using long integer precision is a benefit because the Macintosh processor's performs calculations on long integers very quickly. There are three very rare situations in which a timer overflow will result, those being a condition that cause a mathematical overflow to occur. When these conditions arise, the Timer's calculated "next time" may not be accurate, meaning that the Timer may report its next event too early, too late, too frequently or too infrequently. Also, the Timer's event count may overflow to report a negative number of events.

Fortunately, the conditions that cause a Timer overflow are very rare. The first condition is one in which the Timer's "next time" overflows. The Timer's "next time," or the Event.Timer.NextTime field of the event record, is a counterpart for the value returned by the toolbox's TickCount routine (TickCount returns the number of ticks, or 1/60 second intervals that have transpired since your Macintosh was booted). A little bit of math tells us that the largest number that can be represented by a signed long integer is a little over two trillion, therefore, the time calculated for the Timer's next event cannot exceed:

2,147,483,647	ticks	or
35,791,394	seconds	or
596,523	minutes	or
9942	hours	or
414	days	

from the instant that your Macintosh was booted. You can cause the Timer to overflow either by specifying that the Timer's period exceeds the limits defined above (i.e., 415 days), or, while the Timer is running, it calculated its next time beyond the legal range. An example of this is a Timer whose period is 210 days: it will report its first event correctly 210 days later, but the subsequent event will be calculated to occur on day 420 (2 x 210 day period), and this will cause the Timer to overflow.

The second condition that can trigger a Timer overflow is when the total number of events reported, either logically or physically, exceed the limit of 2,147,483,647. Just use a calculator to calculate the total number of events a Timer is ever expected to report, and make sure it does not exceed this limit. If your application will certainly exceed this limit after running for a number of days, consider creating a one shot Timer that recreates the other Timer, and sets a global

variable to indicate that the Timer is now on its second cycle.

The last condition is one in which the calculations internal to the Timer overflow. This can only happen when your Timer is set to a frequency (eg: 30 events per hour) as opposed to a period (eg. 20 seconds between events). Make sure that the following formula does not result in a value that exceeds 2,147,483,647:

MaxEvents x TicksPerUnitOfMeasure = *X*

MaxEvents is the maximum number of events you application will ever receive before it quits, and TicksPerUnitOfMeasure is the number of ticks (1/60 second intervals) within the unit of measure specified in your Spec parameter when calling the NewTimer routine. The following is an example of this formula:

	EventsPerMeasure	x	TicksPerUnitOfMeasure	x	RunningTimeInMeasures	= *X*
or	1,000 events	x	per day	x	7 days	= *X*
or	1,000	x	24 hours x 60 minutes	x	7	= *X*
or	1,000	x	1,440 minutes x 60 secs	x	7	= *X*
or	1,000	x	86,400 seconds x 60 ticks	x	7	= *X*
or	1,000	x	5,184,000 tick (in a day)	x	7	= 36,288,000,000

In the example above if you have a Timer running at 1,000 events per day, it will overflow before it reaches the seventh day (36,288,000,000 exceeds the limit of 2,147,483,647).

Fortunately, the “reasonable” way to create a Timer avoids these potential issues. For example, instead of creating a Timer that generates 2400 events per day, create one that generates 100 events per hour. Another alternative is to revise a high frequency/high unit of measure specification to a period. For example, 5000 events per day is equivalent to 1036.8 ticks between events, so a period of 1037 ticks between events would likely be accurate enough. A little reasoning and planning will let you create a Timer that suits your needs.

The table below will assist you in converting a high frequency/high unit of measure specification to a period.

1 tick	=	1 tick
1 second	=	60 ticks
1 minute	=	3,600 ticks
1 hour	=	216,000 ticks
1 day	=	5,184,000 ticks

To convert a frequency into a period, take the number of ticks and divide by the number of events in the unit of measure. For example:

8,400 events per day	*is the same as…*
8,400 events per 5,184,000 ticks,	*therefore…*
5,184,000 ticks in a day ÷ 8,400 events	*is equivalent to…*
617 ticks between events.	

NewTimer

Create a new Timer.

C
```
pascal void NewTimer (short Timer, long DelayTicks, long Value, long Spec,
            short Window; EventHandlerUPP EventHandler);
```

Pascal
```
procedure NewTimer (Timer: INTEGER; DelayTicks, Value, Spec: LONGINT;
            Window: INTEGER; EventHandler: EventHandlerUPP);
```

The NewTimer routine creates a Timer, an automated process that reports a doTimer event at a specified frequency or period.

Timer specifies the Timer number that is created (from 1 to 32767). Once a Timer is created, it is referenced by this *Timer* number. If a Timer using the same *Timer* number already exists, it is deleted, then a new Timer is created as specified by the parameters in the NewTimer routine, thereby re-using the *Timer* number. Tools Plus allows up to 32767 Timers to be created simultaneously, however, your Macintosh’s processor and the amount of work that is done

by your application in response to Timer events will determine the number of Timers that can run concurrently, and how frequently they can generate events.

DelayTicks specifies the time in ticks (1/60 of a second) that the Timer will remain dormant before it reports its first event. The default value of -1 (the timerStandardInitDelay constant) makes the Timer behave as you would expect, reporting its first event at the same period or frequency as every subsequent event. A value of 0 (the timerInstantEvent constant) tells the Timer that it should report an event as soon as possible, then report subsequent events in a normal manner. Any other value specified for *DelayTicks* results in the Timer "sleeping" for that many ticks before it reports its first event. If you synchronize this Timer with another Timer by using the timerSyncToTimer option in the *Spec* parameter, this Timer becomes the "child" and it reports each event *DelayTicks* ticks after the parent Timer reports its event.

The *Value* parameter specifies the frequency or period of the Timer, such as *30* events per second, or *20* seconds between events. If you synchronize this Timer with another Timer by using the timerSyncToTimer option in the *Spec* parameter, this Timer becomes the "child" and it synchronizes to a parent Timer whose number is specified in *Value*. The parent Timer must already exist when synchronizing Timers.

Spec specifies a Timer's behavior. The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result. For example, a Timer that runs at 20 events per second and is deleted automatically when the window is hidden would have a spec of timerEventsPerSecond + timerDeleteForHiddenWindow. The constants defining the available options are as follows:

Optionally choose any of the following behavior options…

`timerOneShot`
: The Timer reports a single event, then it is automatically deleted. If this Timer is synchronized to a parent Timer that is a one shot Timer, then this timer automatically becomes a one shot Timer.

`timerDeleteForHiddenWindow`
: The Timer is deleted when its window is hidden. This option is ignored if the *Window* parameter is set to zero.

`timerSyncToTimer`
: Synchronize to a parent Timer. This Timer always reports an event after the parent Timer, separated in time by the number of ticks (1/60 second) specified in the *DelayTicks* parameter. The *Value* parameter specifies the parent Timer's number.

`timerLockTimerToCount`
: When this option is used, a physical Timer event is always reported for each event that is required (i.e., your event handler routine is called each time the event record's Event.Timer.Count field is incremented). When the system is busy and the required number of events cannot be reported, all backlogged events are reported as soon as the system is able to do so. The risk is that the system may be too busy to ever catch up to where it should be if your event handler routine does a lot of work, or if the events are reported very frequently.

  If this option is not used and the system gets too busy to report the required number of events, the Timer simply resumes reporting events at the normal rate when the system is able to do so, and for each reported event, the event record's Event.Timer.Count field indicates that number of events that *should* have been reported. This default behavior is attained by not using this option.

  This option is ignored if you use any of the following options (detailed below): timerTicksBetweenEvents, timerSecondsBetweenEvents, timerMinutesBetweenEvents, timerHoursBetweenEvents, or timerDaysBetweenEvents.

Choose only one of the following timing options if the timerSynchToTimer option is not used…

`timerEventsPerTick`
: Report the number of Timer events specified in the *Value* parameter for each passing tick (1/60 second).

`timerEventsPerSecond`
: Report the number of Timer events specified in the *Value* parameter for each passing second.

`timerEventsPerMinute`
: Report the number of Timer events specified in the *Value* parameter for each passing minute.

`timerEventsPerHour`
: Report the number of Timer events specified in the *Value* parameter for each passing hour.

`timerEventsPerDay`
: Report the number of Timer events specified in the *Value* parameter for each passing day.

`timerTicksBetweenEvents`
: Report Timer events that are separated by the number of ticks (1/60 second) specified in the *Value* parameter.

`timerSecondsBetweenEvents`
: Report Timer events that are separated by the number of seconds specified in the *Value* parameter.

`timerMinutesBetweenEvents`
: Report Timer events that are separated by the number of seconds specified in the *Value* parameter.

`timerHoursBetweenEvents`
: Report Timer events that are separated by the number of seconds specified in the *Value* parameter.

`timerDaysBetweenEvents`
: Report Timer events that are separated by the number of seconds specified in the *Value* parameter.

Window specifies the window number for which the Timer event is generated. If a value of zero (0) is specified, the event is reported to your application's main event handler routine. If a *window* number is specified, then the Timer event is reported to that window's event handler routine. The specified window must be open when the Timer is created, although it may be hidden. If the specified window is not open, NewTimer does nothing.

EventHandler is a UPP to an event handler routine. If you specify nil, the Timer reports its event to the window number specified by the *Window* parameter. If you specify a UPP, Tools Plus calls that routine to handle the Timer event. See the NewEventHandlerProc for details about UPPs, and the section named "The Event Handler Routine" earlier in this chapter for details on how to write and event handler routine.

```
CONST                                                 {Delay For First Event:                 }
            timerInstantEvent          = 0;           {First event is reported immediately    }
            timerStandardInitDelay     = -1;          {No special timing for first event      }

                                                      {Optional Settings:                     }
            timerOneShot               = $00010000;   {Delete timer after reporting an event  }
            timerDeleteForHiddenWindow = $00020000;   {Delete timer when its window is hidden }
            timerSyncToTimer           = $00040000;   {Synchronize with another timer         }
            timerLockTimerToCount      = $00080000;   {Report physical event for each logical }
                                                      {  event (count)                        }

                                                      {Timer's Unit Of Measure:               }
            timerEventsPerTick         = $00000001;   {Events per tick (1/60 second)          }
            timerEventsPerSecond       = $00000002;   {Events per second                      }
            timerEventsPerMinute       = $00000003;   {Events per minute                      }
            timerEventsPerHour         = $00000004;   {Events per hour                        }
            timerEventsPerDay          = $00000005;   {Events per day                         }
            timerTicksBetweenEvents    = $00000006;   {Number of ticks between events         }
            timerSecondsBetweenEvents  = $00000007;   {Number of seconds between events       }
            timerMinutesBetweenEvents  = $00000008;   {Number of minutes between events       }
            timerHoursBetweenEvents    = $00000009;   {Number of hours between events         }
            timerDaysBetweenEvents     = $0000000A;   {Number of days between events          }
```

DeleteTimer

Delete a timer.

C
```
pascal void DeleteTimer (short Timer);
```

Pascal
```
procedure DeleteTimer (Timer: INTEGER);
```

Timer specifies the timer number (from 1 to 32767) that is deleted. If the timer does not exist, DeleteTimer does nothing. If this Timer is a “parent” Timer that is synchronized to one or more “child” Timers, the children are deleted as well.

Note that Timers that are associated with a window are deleted automatically when the window is closed, or optionally, when the window is hidden. A Timer can also be deleted automatically after reporting a single event. See the NewTimer routine’s Spec parameter for details.

Responding to Events

This section details how your application should respond after receiving an event from Tools Plus. Typically, your application will respond to these events by having a C/C++ switch statement (or Pascal case statement) in your application's main event handler routine with a case for each event that it chooses to respond to. You may also elect to have a window event handler routine that processes events for one or more windows. Abundant details are provided herein, but most applications will only need to observe a small number of the listed programming considerations. Much of the volume that you will see is intended to inform you of what is happening, as opposed to being a list of rules that you must follow.

All Tools Plus events are listed alphabetically by event name for quicker reference.

---

doActivate event

Indicator that a window has been activated.

The doActivate event is reported whenever an inactive window is activated. Unlike the Event Manager, Tools Plus queues doActivate events and reports them correctly regardless of the number of windows that are opened, closed, activated, deactivated, hidden or displayed (or the sequence in which any of these actions occur). The only applications that need to respond to doActivate events are those that manually activate objects when a window becomes active. Most applications will ignore this event.

Programming Considerations

- When your application receives a doActivate event, a doPreRefresh and doRefresh will follow if any portion of the window needs to be refreshed.
- Unlike ordinary Macintosh applications, a doActivate event is not generated when a window is first opened. You can override this behavior when opening a window.
- Your application receives a doActivate event when a standard window (not a tool bar or floating palette) becomes active. This can occur under any of the following conditions:
    - your application calls ActivateWindow to activate a standard window
    - a window is closed to activate the window behind it
    - your application calls WindowDisplay to unhide a standard window
    - your application is resumed after being suspended
    - when running under Finder (pre-System 7), and a desk accessory is closed
- Your application receives a doActivate event when a tool bar or floating palette becomes active. This can occur under any of the following conditions:
    - a modal window was closed and you have a tool bar and/or floating palettes open
    - when running under Finder (pre-System 7), and a desk accessory is closed
- Your application may decide to open (or unhide) a related floating palette when a window is activated, or to close (or hide) a floating palette that is no longer relevant to the window that is being activated.
- A doActivate event is only generated for windows in your application. It is not generated for desk accessories, the Dialog Manager's dialogs, alerts, or Dynamic Alerts. These events are handled automatically.

Valid Event Record Fields

`Event.Window`	**Window Number**: Window number that was activated

---

doAutoKey event

Indicator that the user held down a key, and it is repeating.

The user has pressed down a key on the keyboard or numeric pad, and the key is repeating. The key cannot be processed internally by Tools Plus. In most cases, this event should be treated identically to a doKeyDown event. Each doAutoKey event will follow a doKeyDown of the same sort, so your application may want to make some repeating Command key sequences illegal.

Programming Considerations

- This event is generated only if the key can not be processed internally by Tools Plus.
- Command key sequences that are equivalents to menu items generate doMenu events.
- If a Command key equivalent for a hierarchical menu item is typed, and the hierarchical menu is not ultimately attached to a pull-down menu, a doKeyDown or doAutoKey event is reported instead (as though the hierarchical menu did not exist).
- The Modifiers field tells your application if the repeating key down event was a Command key sequence. All Command key sequences that are not menu equivalents are returned to your application as doKeyDown or doAutoKey events. If your application gets such a Command key sequence, it should carry out the appropriate action, or beep the user if the Command key sequence is illegal.
- If Return or Enter are typed and held down when the active window has a default button, a doButton event is reported for the default button.
- If an active editing field exists on the active window, repeating keys will affect the field and will not generate events. This applies even if the field’s length is limited, or if Return has been disallowed in a field.
- The HaveTabInFocus routine can be used to detect if the user hit the Tab or Shift-Tab key in an active field or in an object that has the keyboard focus, and therefore want to move to the next/previous item that wants the keyboard focus. The TabToFocus routine moves the focus to the next/previous item. If you initialized Tools Plus with the initAutoFocusChanges option, tabbing to the next/previous field or keyboard focus item is automatic.
- If Tab is typed and the window contains an active editing field, it indicates that the user wants to tab to another field or keyboard focus item. You may want to validate the active field’s edited text before allowing the user to tab to the new item. If this is the case, use GetEditString (or GetEditHandle) to obtain a copy of the edited text, then your application can check the string for errors. If an error is detected, display an appropriate alert box and ignore the doKeyDown event. If no error is detected, call the SaveFieldString routine to save the edited text as the field’s string, then call the TabToFocus routine. If you initialized Tools Plus with the initAutoFocusChanges or initAutoSaveFieldString options, an active field’s edited text is automatically saved without having to call SaveFieldString.
- A Return key is reported only if the window does not have a default button or active field.
- If Enter is typed, it usually indicates that the user wants to enter the screen’s data. This is the same process that is carried out by clicking an OK button. See “Tab” above, regarding validating and saving a field’s edited text. An Enter key is reported only if the window does not have a default button. The repeating Enter key may be considered illegal because a doKeyDown event will report the first Enter, after which your application should have acted accordingly. An Enter key is reported only if the window does not have a default button.
- When the watch cursor is displayed, the only doAutoKey event that is reported is a ⌘-., which is the user’s request to halt a lengthy process. Other doAutoKey events are discarded.
- If the active window has an active editing field, Tools Plus automatically processes the Command key equivalents for the Edit menu’s Undo, Cut, Copy, and Paste commands. These selections do not generate events.
- There are two situations where several keys produce the same key character: the Clear and Escape keys, and the F1 through F15 keys. In these cases, use the key code to differentiate the keys. Constants are provided for these key characters and key codes.
- A doAutoKey event is not generated when the user types in a desk accessory. These key strokes are handled automatically.

Valid Event Record Fields

`Event.Window` **Window Number**: Active window number (frontmost standard window, tool bar, or a floating palette)

`Event.Key.Code` **Key Number**: Number of the key that is repeating. This key code is a key number that is not affected by the Caps Lock, Shift, Option, Command or Control modifiers.

`Event.Key.Chr` **Key Character**: Character resulting from a key that is repeating. This character is affected by the Caps Lock, Shift, Option, and/or Command modifiers.

`Event.Modifiers` **Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down immediately before the key was pressed down.

`Event.Field` **Editing Field Number**: Active editing field number when key was pressed.

```
CONST                                        {Key characters and key codes…              }
               EnterKey      = char($03);    {KEYS:  Key characters (ASCII) for common   }
               BackSpaceKey  = char($08);    {       non alpha-numeric keys.             }
               DeleteKey     = BackSpaceKey; {       In some cases, several keys         }
               TabKey        = char($09);    {       produce the same ASCII character.   }
               ReturnKey     = char($0D);    {       They can be differentiated by       }
               EscClearKey   = char($1B);    { ($1B) using the key code as indicated     }
               LeftArrowKey  = char($1C);    {       below.  Some keys are available     }
               RightArrowKey = char($1D);    {       only on an extended keyboard.       }
               UpArrowKey    = char($1E);    {                                           }
               DownArrowKey  = char($1F);    {                                           }
               HelpKey       = char($05);    {                                           }
               HomeKey       = char($01);    {                                           }
               DeleteFwdKey  = char($7F);    {                                           }
               EndKey        = char($04);    {                                           }
               PageUpKey     = char($0B);    {                                           }
               PageDownKey   = char($0C);    {                                           }
               FKey          = char($10);    { ($10) Function keys F1 to F15             }
               EscKeyCode    = $35;          { ($1B) KEY CODES used to differentiate     }
               ClearKeyCode  = $47;          { ($1B) between keys which produce          }
               F1KeyCode     = $7A;          {       the same key characters.            }
               F2KeyCode     = $78;          {                                           }
               F3KeyCode     = $63;          {                                           }
               F4KeyCode     = $76;          {                                           }
               F5KeyCode     = $60;          {                                           }
               F6KeyCode     = $61;          {                                           }
               F7KeyCode     = $62;          {                                           }
               F8KeyCode     = $64;          {                                           }
               F9KeyCode     = $65;          {                                           }
               F10KeyCode    = $6D;          {                                           }
               F11KeyCode    = $67;          {                                           }
               F12KeyCode    = $6F;          {                                           }
               F13KeyCode    = $69;          {                                           }
               F14KeyCode    = $6B;          {                                           }
               F15KeyCode    = $71;          {                                           }
```

doButton event

Indicator that user has clicked a button.

The doButton event reports that the user has clicked a button in the active window. This includes push buttons, check boxes and radio buttons as well as custom CDEFs that are made to behave like buttons by Tools Plus. The doButton event is also reported if the active window has a default push-button, and the user pressed the Return or Enter key to invoke the default. If this is the case, the window's default button will already have been "flashed" as if the user had clicked it.

Programming Considerations

- If the user clicked a check box, use SelectButton to reverse the check box's selection (i.e., select or deselect it).
- If the user clicked a radio button, use SelectButton to select the radio button, and to deselect the other buttons in the radio button's group. Note that a panel can be used to automatically deselect other radio buttons in the group.

- This event will never occur when the watch cursor is displayed, since buttons cannot be clicked. The exception is if the WatchCursorButtons routine has been used to allow the watch cursor to click push-buttons.
- If your application responds to double-clicks in radio buttons and Event.Button.DoubleClick is true, consider the event to mean “click button and OK,” in which case the default button should be flashed and the appropriate action taken.
- A doButton event is not generated when the user clicks buttons in a dialog box, alert box, or desk accessory. These events are handled automatically.

Valid Event Record Fields

`Event.Window` — **Window Number**: Window number containing the selected button (frontmost standard window, tool bar, or a floating palette).

`Event.Button.Num` — **Button Number**: Button number that was clicked by the user.

`Event.Button.Part` — **Button Part**: Part of button that was clicked by user. This field is ignored in most cases. Buttons with multiple parts began to appear with the introduction of the Appearance Manager, such as the “Little Arrows” control which can be clicked inUpButton or inDownButton.

`Event.Button.DoubleClick` — **Button’s Double-click Status**: Was the button double-clicked? (radio buttons only).

`Event.Modifiers` — **Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down when the button was clicked (during the mouse-down).

```
CONST                                {Typical button parts…                   }
               inButton     = 10;   {A standard push button                  }
               inCheckBox   = 11;   {A check box or radio button             }
               inUpButton   = 20;   {Up arrow                                }
               inDownButton = 21;   {Down arrow                              }
```

doChgInField event

Indicator that the contents of your application’s active editing field have been changed.

The doChgInField event is reported whenever the active field in your application is altered, either through typing, cutting, pasting (including the use of the PasteIntoField routine), clearing, or undoing/redoing (using the Edit menu). Most applications will ignore this event.

Valid Event Record Fields

`Event.Window` — **Window Number**: Window number containing the changed field

`Event.Field` — **Editing Field Number**: Editing field that was changed.

Programming Tips:

1 Your application can use the FieldIsEmpty routine to quickly determine if an editing field contains any characters (including spaces). If the field is empty, you may want to disable a “save” button or equivalent.

doChgMonitorSettings event

Indicator that the user has changed monitor settings, which includes alterations to any of the following settings on any monitor:
- number of colors or grays
- monitor size
- monitor resolution
- orientation of multiple monitors (side by side, 1=left/2=right or 2=left/1=right, height offsets, etc.)
- moving menu bar to another monitor
- changing the menu bar's height (available in Mac OS 8.5 and later when a different theme is selected)

Mac OS does not have any direct way of informing your application that the user has changed monitor settings. Instead, Tools Plus checks for these changes whenever a window needs to be refreshed, or when your suspended application resumes. If Tools Plus detects any changes then, it reports a doChgMonitorSettings event. See the CheckForMonitorChanges routine (in the Color Drawing & Multiple Monitors chapter) if you need to check for monitor changes at any other time. In most cases, your application will want to ignore this event.

If the menu bar's height is changed by a different theme, Tools Plus automatically moves all your open (and possibly hidden) windows down or up to accounts for the slight change in the menu bar height.

---

doChgWindow event

Request to activate an inactive window that belongs to your application.

The request to activate an inactive window is made in response to the user clicking anywhere on an inactive window that belongs to your application. However, there is one exception. If the user clicks in an inactive window's title bar while holding the Command key, the window is dragged without being activated.

Programming Considerations

- A window can be activated by using the ActivateWindow routine.
- If the active window has an active editing field, you may want to validate the edited text before allowing the user to activate another window. If this is the case, use GetEditString (or GetEditHandle) to obtain a copy of the edited text, then your application can check the string for errors. If an error is detected, display an appropriate alert box and ignore the doChgWindow event. If no error is detected, call the SaveFieldString routine to save the edited text as the field's string, then activate the required window.
- If your application decides not to allow the user to activate a window, it should display an appropriate alert box to explain why.
- The doChgWindow event will never occur when a modal window is active, or when the watch cursor is displayed since other windows belonging to your application cannot be clicked.
- A doChgWindow event does not occur if the user clicks on a tool bar or floating palettes.
- If your application is running under Finder (System 6 or prior), a doChgWindow event will not occur when the user [1] clicks a desk accessory from an active window, [2] clicks a desk accessory from another active desk accessory, or [3] clicks the previously active window from an active desk accessory. These are all handled automatically.

Valid Event Record Fields

`Event.Window`	**Window Number**: Window number that the user is trying to activate.
`Event.Event`	**The Event Manager's Event Record**: Event record as retrieved from the Event Manager.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down when the window was clicked (during the mouse-down).

---

doClick event

Indicator that the user has completed or is in the midst of a single-click, double-click, triple-click, or drag.

A single-click, double-click, triple-click, or drag has been reported in the active window. The event may still be in progress when Tools Plus calls your event handler routine, such as a double-click with the mouse button still down on the second click. The Mouse part of the event record contains a field called *What*, which is the mouse event code that tells your application what type of mouse click has occurred. The rest of the mouse record contains details on each mouse-down and mouse-up event.

When WaitForMultiClicks(off) is used (this is the default), mouse-related events are reported as they occur. When WaitForMultiClicks is turned on, Tools Plus makes a series of decisions to calculate the single-click and multiple-click information, as well as dragging information. It reports an event only after it has determined that a single-click, double-click, or triple-click is in progress or has been completed. The decisions made by Tools Plus when WaitForMultiClicks is turned on are as follows:

1. If the first mouse-down is in an active window's content region, and it is not in a button, picture button, scroll bar, editing field, list box, pop-up menu or custom control, a click has begun but no event is reported.
2. If the user holds the mouse button down too long during the first click to begin the second click of a double-click, a doClick event is generated with Event.Mouse.What set to inClick1Drag (single-click started, mouse still down). When the user releases the mouse a doClick event is generated with Event.Mouse.What set to inClick1 (single-click completed). Subsequent clicks start counting at one again.
3. If the user releases the mouse button after the first click and leaves it up too long for the next mouse-down to be considered a double-click, a doClick event is generated with Event.Mouse.What set to inClick1 (single-click completed).
4. If the second mouse-down is in an active window's content region, and it is not in a button, picture button, scroll bar, editing field, list box, pop-up menu or custom control, and the mouse-down occurred within the time limit for a multiple click, then Tools Plus continues.
5. If the user holds the mouse button down too long during the second click to begin the third click of a triple-click, a doClick event is generated with Event.Mouse.What set to inClick2Drag (double-click started, mouse still down). When the user releases the mouse a doClick event is generated with Event.Mouse.What set to inClick2 (double-click completed). Subsequent clicks start counting at one again.
6. If the user releases the mouse button after the second click and leaves it up too long for the next mouse-down to be considered a triple-click, a doClick event is generated with Event.Mouse.What set to inClick2 (double-click completed).
7. If the third mouse-down of a click is in an active window's content region, and it is not in a button, picture button, scroll bar, editing field, list box, pop-up menu or custom control, and the mouse-down occurred within the time limit for a multiple click, then Tools Plus continues.
8. If the user holds the mouse button down too long during the third click, a doClick event is generated with Event.Mouse.What set to inClick3Drag (triple-click started, mouse still down). When the user releases the mouse a doClick event is generated with Event.Mouse.What set to inClick3 (triple-click completed). Subsequent clicks start counting at one again.
9. If the user releases the mouse button after the third click, a doClick event is generated with Event.Mouse.What set to inClick3 (triple-click completed).

The easiest way to remember how mouse clicks are reported to your application is as follows:

WaitForMultiClicks(on)	Tools Plus waits until it knows that a single-click, double-click, or triple-click is completed before reporting it to your application.
WaitForMultiClicks(off)	Each mouse-down and mouse-up is immediately reported to your application even if there are mouse-down or mouse-up events in the event queue. For example, by the end of a double-click your application will have received the following mouse events:

inClick1Drag	single-click, mouse still down
inClick1	single-click completed
inClick2Drag	double-click, mouse still down
inClick2	double-click completed

Programming Considerations

- If at some point your application is only concerned with a mouse down in a certain area, or a single-click only, or a double-click only, you can disrupt the click as soon as you have obtained enough information by using ResetMouseClicks. For example, if your application only cares about a single-click and it gets an inClick1 mouse event code (single-click completed), ResetMouseClicks will tell Tools Plus “forget the rest of this click.”
- If your application wants to beep the user once when he clicks in an illegal area, call WaitForMultiClicks(false) early in your application. When your application gets a doClick event and it determines the click is illegal, call ResetMouseClicks and wait until the mouse is released. You can wait with an empty loop that keeps running while the toolbox’s StillDown routine returns *true* (i.e., while StillDown do;)
- If you detect a double click and you determine the first two clicks did not occur within the *same* object, you can use the IgnoreFirstMouseClick routine to discard the first click’s information and continue processing for multiple clicks.
- A double-click and triple-click should be scrutinized by your application because the time between the last mouse-down and mouse-up event may exceed the allowable time limit for consecutive clicks. Use the toolbox’s GetDblTime routine to determine the maximum number of clock ticks that can transpire between mouse-down and mouse-up events to constitute a consecutive click, then compare it to the time of your clicks.
- You can choose to either wait for subsequent mouse-down and mouse-up events or have the beginning and end of a click/drag reported right away by using the WaitForMultiClicks routine.
- Your application can place cursor zones on a window, then use FindCursorZone to determine if a click occurred in one of those zones. This can make parts of your window, such as pictures or icons, click-sensitive. See “Cursors” for information about Cursor Tables and Cursor Zones.
- This event will never occur when the watch cursor is displayed, since only buttons can be (optionally) clicked.
- Event.Event is populated with the mouse-down event (as obtained from the Event Manager) that is responsible for the doClick event. Some applications need this event to drive custom controls.
- A doClick event is not generated when the user clicks in a desk accessory. The process is handled automatically.

Valid Event Record Fields

`Event.Window`	**Window Number**: Window number containing the click(s) (frontmost standard window, tool bar, or a floating palette)
`Event.Mouse.What`	**Mouse Event Code**: Type of mouse event that occurred (inClick1, inClick2, etc., listed below)
`Event.Mouse.Down[1].Where`	**1st Mouse Down Location**: Location of 1st mouse-down event (in the active window’s local co-ordinates)
`Event.Mouse.Down[1].When`	**1st Mouse Down Time**: Time of 1st mouse-down (number of “ticks” since startup)
`Event.Mouse.Down[1].Modifiers`	**1st Mouse Down Modifier Record**: Event modifiers for the mouse-down event.
`Event.Event`	**The Event Manager’s Event Record**: Event record for first mouse-down event only.

The next three fields are valid only if the mouse event code is inClick1, inClick2Drag, inClick2, inClick3Drag, or inClick3.

`Event.Mouse.Up[1].Where`	**1st Mouse Up Location**: Location of 1st mouse-up event (in the active window’s local co-ordinates)
`Event.Mouse.Up[1].When`	**1st Mouse Up Time**: Time of 1st mouse-up (number of “ticks” since startup)
`Event.Mouse.Up[1].Modifiers`	**1st Mouse Up Modifier Record**: Event modifiers for the mouse-up event.

The next three fields are valid only if the mouse event code is inClick2Drag, inClick2, inClick3Drag, or inClick3.

`Event.Mouse.Down[2].Where` **2nd Mouse Down Location**: Location of 2nd mouse-down event (in the active window's local co-ordinates)

`Event.Mouse.Down[2].When` **2nd Mouse Down Time**: Time of 2nd mouse-down (number of "ticks" since startup)

`Event.Mouse.Down[2].Modifiers` **2nd Mouse Down Modifier Record**: Event modifiers for the mouse-down event.

The next three fields are valid only if the mouse event code is inClick2, inClick3Drag, or inClick3.

`Event.Mouse.Up[2].Where` **2nd Mouse Up Location**: Location of 2nd mouse-up event (in the active window's local co-ordinates)

`Event.Mouse.Up[2].When` **2nd Mouse Up Time**: Time of 2nd mouse-up (number of "ticks" since startup)

`Event.Mouse.Up[2].Modifiers` **2nd Mouse Up Modifier Record**: Event modifiers for the mouse-up event

The next three fields are valid only if the mouse event code is inClick3Drag, or inClick3.

`Event.Mouse.Down[3].Where` **3rd Mouse Down Location**: Location of 3rd mouse-down event (in the active window's local co-ordinates)

`Event.Mouse.Down[3].When` **3rd Mouse Down Time**: Time of 3rd mouse-down (number of "ticks" since startup)

`Event.Mouse.Down[3].Modifiers` **3rd Mouse Down Modifier Record**: Event modifiers for the mouse-down event.

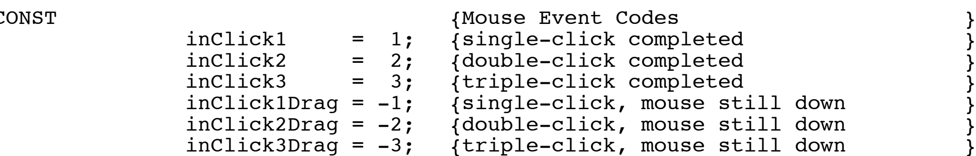
The next three fields are valid only if the mouse event code is inClick3.

`Event.Mouse.Up[3].Where` **3rd Mouse Up Location**: Location of 3rd mouse-up event (in the active window's local co-ordinates)

`Event.Mouse.Up[3].When` **3rd Mouse Up Time**: Time of 3rd mouse-up (number of "ticks" since startup)

`Event.Mouse.Up[3].Modifiers` **3rd Mouse Up Modifier Record**: Event modifiers for the mouse-up event.

`Event.Mouse.Where` **Mouse Location**: Mouse location when your event handler routine was last called by Tools Plus (in the active window's local co-ordinates)



```
CONST                          {Mouse Event Codes                 }
          inClick1     =  1;   {single-click completed            }
          inClick2     =  2;   {double-click completed            }
          inClick3     =  3;   {triple-click completed            }
          inClick1Drag = -1;   {single-click, mouse still down    }
          inClick2Drag = -2;   {double-click, mouse still down    }
          inClick3Drag = -3;   {triple-click, mouse still down    }
```

Note: *For C programmers…* The event record's arrays are documented using Pascal nomenclature (the elements are numbered 1, 2 and 3). In C, the same array's elements are numbered 0, 1 and 2 (they start at zero). When C programmers read:

```
Event.Mouse.Down[1].Where
```

it indicates the first element of the array, which translates to the following C source code:

```
Event.Mouse.Down[0].Where
```

doClickControl event

Indicator that user clicked a custom control.

The doClickControl event reports that the user has clicked in a custom control. This event must be handled entirely by your application, since only you know how your custom control should behave.

Programming Considerations

- This event will never occur when the watch cursor is displayed, since custom controls cannot be clicked.
- This event is not generated for a custom CDEF that is made to behave like a button or scroll bar (i.e., one that is created with Tools Plus's NewButton or NewScrollBar routines). Tools Plus reports activity in those controls through doButton and doScrollBar events.

Valid Event Record Fields

`Event.Window` — **Window Number**: Window number containing the affected control (frontmost standard window, tool bar, or a floating palette)

`Event.Event` — **The Event Manager's Event Record**: Event record as retrieved from the Event Manager.

`Event.Modifiers` — **Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down when the close box was clicked (during the mouse-down).

---

doClickDesk event

Indicator that user clicked in the desk top.

The doClickDesk event reports that the user has clicked in the desk top. Most applications will choose to ignore this event, however, it can be used to deselect objects on the active window.

Programming Considerations

- This event will never occur when a modal window is active, since clicks cannot occur outside of the window.
- This event will never occur when the watch cursor is displayed, since the desk top cannot be clicked.

Valid Event Record Fields

All fields in the event record, except What, are undefined.

---

doClickToFocus event

Indicator that the user clicked in an inactive editing field or in an inactive item that wants the keyboard focus. Normally, your application just calls ClickToFocus in response to this event.

Programming Considerations

- The ClickToFocus routine is used to process a user's click in an inactive editing field. It could be in the same window that the user is working in, or it could be in a tool bar or floating palette.
- If you initialized Tools Plus with the initAutoFocusChanges option, clicking to another field or keyboard focus item is automatic.
- If an editing field is active when you receive a doClickToFocus event, you may want to validate its edited text before allowing the user to click to the new field. If this is the case, use GetEditString (or GetEditHandle) to obtain a copy of the edited text, then your application can check the string for errors. If an error is detected, display an appropriate

alert box and ignore the doClickToFocus event. If no error is detected, call the SaveFieldString routine to save the edited text as the field's string, then call the ClickToFocus routine to process the user's click in the new field. If you initialized Tools Plus with the initAutoFocusChanges or initAutoSaveFieldString options, an active field's edited text is automatically saved without having to call SaveFieldString.

- Between the time when the doClickToFocus event is reported and when your application calls ClickToFocus is called, observe the following rules:
  - do not call Process1EventWhileBusy or ProcessToolboxEvent
  - do not open or close any windows, including alerts and dialogs
  - do not hide or show any windows
  - do not activate any windows
  - do not activate, deactivate, enable, disable or delete any user interface elements

  If these rules are not observed, ClickToFocus will do nothing and the user's click is ignored.
- An event will not occur if the user clicks in an editing field that is already active.
- This event will never occur when the watch cursor is displayed, since editing fields cannot be clicked.
- A doClickToFocus event is not generated when the user clicks in a desk accessory's editing fields. The process is handled automatically.

Valid Event Record Fields

`Event.Window`	**Window Number**: Window number containing the clicked keyboard focus item (the frontmost standard window, tool bar, or a floating palette)
`Event.Button.Num`	**Button Number**: Button number that was clicked by the user. Zero (0) indicates that the item clicked is not implemented as a button.
`Event.Button.Part`	**Button Part**: Part of button that was clicked by user. This field is ignored in most cases. Buttons with multiple parts began to appear with the introduction of the Appearance Manager, such as the "Little Arrows" control which can be clicked inUpButton or inDownButton.
`Event.ScrollBar.Num`	**Scroll Bar Number**: Scroll bar number that was clicked by the user. Zero (0) indicates that the item clicked is not implemented as a scroll bar.
`Event.ScrollBar.Part`	**Scroll Bar Part**: Part of scroll bar that was clicked by user.
`Event.Field`	**Editing Field Number**: Editing field that was clicked by the user. Zero (0) indicated that the item clicked is not an editing field.
`Event.ListBox.Num`	**List Box Number**: List box number that was clicked by the user. Zero (0) indicates that the item clicked is not a list box.
`Event.Event`	**The Event Manager's Event Record**: Event record as retrieved from the Event Manager.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down when the close box was clicked (during the mouse-down).

---

doDeactivate event

Indicator that a window has been deactivated.

The doDeactivate event is reported whenever an active window is deactivated. Unlike the Event Manager, Tools Plus queues doDeactivate events and reports them correctly regardless of the number of windows that are opened, closed, activated, deactivated, hidden or displayed (or the sequence in which any of these actions occur). The only applications that need to respond to doDeactivate events are those that manually deactivate objects when a window becomes active. Most applications will ignore this event.

Programming Considerations

- Your application receives a doDeactivate event when a standard window (not a tool bar or floating palette) becomes inactive. This can occur under any of the following conditions:
  - a standard window is opened or unhidden in front of a standard window
  - your application is suspended
  - when running under Finder (pre-System 7), and a desk accessory is opened
- Your application receives a doDeactivate event when a tool bar or floating palette becomes inactive. This can occur under any of the following conditions:
  - a modal window is opened and you have a tool bar and/or floating palettes open
  - when running under Finder (pre-System 7), and a desk accessory is opened
- A doDeactivate event is only generated for windows in your application. It is not generated for desk accessories, the Dialog Manager's dialogs, alerts, or Dynamic Alerts. These events are handled automatically.

Valid Event Record Fields

`Event.Window` — **Window Number**: Window number that was deactivated

---

doGoAway event

Request to close the active window.

The request to close the active window is made in response to the user clicking inside the window's "close" box. This has the same effect as choosing the File menu's Close item.

Programming Considerations

- A window is closed by using the WindowClose routine.
- If the window has an active editing field, you may want to validate its edited text before allowing the user to close the window. If this is the case, use GetEditString (or GetEditHandle) to obtain a copy of the edited text, then your application can check the string for errors. If an error is detected, display an appropriate alert box and ignore the doGoAway event. If no error is detected, call the SaveFieldString routine to save the edited text as the field's string, then close the required window.
- If any changes have been made in the window's data, your application should display an appropriate alert box and ask the user if changes should be saved. The options should be Yes, No, and Cancel, the last of which would ignore the doGoAway event.
- The doGoAway event will never occur when the watch cursor is displayed, since the close box cannot be clicked.
- A doGoAway event is not generated when the user clicks a desk accessory's close box. The desk accessory is closed automatically.
- You may want to consider hiding a window (with the WindowDisplay routine) instead of closing it in situations where a window is opened and closed frequently, such as a floating palette or tool bar. Hiding and showing a window preserves the window's setting and location, and is much faster than opening a window and recreating the user interface elements. However, hiding a window does not release the memory consumed by the objects in the window (such as picture buttons, list boxes, etc.)

Valid Event Record Fields

`Event.Window` — **Window Number**: Window number the user is trying to close (frontmost standard window, or a floating palette).

`Event.Modifiers` — **Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down when the close box was clicked (during the mouse-down).

---

doGrowWindow event

Indicator that user has changed a window's size.

The doGrowWindow event reports that the user has changed a window by dragging a window's "size box." This will always occur on the active window.

Programming Considerations

- Your application can call the WindowStatus routine to obtain the window's new width and height in pixels.
- If the window is enlarged either vertically or horizontally, Tools Plus will report a doRefresh event for this window.
- This event will never occur when the watch cursor is displayed, since the size box cannot be clicked.
- Tools Plus may resize a tool bar automatically in response to the user changing the main monitor's size or resolution. When this happens, Tools Plus reports a doGrowWindow event for the tool bar.
- A doGrowWindow event is not generated when the user changes a desk accessory's size. The process is handled automatically.

Valid Event Record Fields

`Event.Window`	**Window Number**: Window number that was re-sized.

---

doKeyDown event

Indicator that the user pressed down a key.

The user has pressed down a key on the keyboard or numeric pad, and the key cannot be processed internally by Tools Plus.

Programming Considerations

- This event is generated only if the key can not be processed internally by Tools Plus.
- Command key sequences that are equivalents to menu items generate doMenu events.
- If a Command key equivalent for a hierarchical menu item is typed, and the hierarchical menu is not ultimately attached to a pull-down menu, a doKeyDown or doAutoKey event is reported instead (as though the hierarchical menu did not exist).
- The Modifiers field tells your application if the key down event was a Command key sequence. All Command key sequences that are not menu equivalents are returned to your application as doKeyDown or doAutoKey events. If your application gets such a Command key sequence, it should carry out the appropriate action, or beep the user if the Command key sequence is illegal.
- If Return or Enter are typed when the active window has a default button, a doButton event is reported for the default button.
- If an active editing field exists on the active window, key strokes will affect the field and will not generate events. This applies even if the field's length is limited, or if Return has been disallowed in a field.
- The HaveTabInFocus routine can be used to detect if the user hit the Tab or Shift-Tab key in an active field or in an object that has the keyboard focus, and therefore want to move to the next/previous item that wants the keyboard focus. The TabToFocus routine moves the focus to the next/previous item. If you initialized Tools Plus with the initAutoFocusChanges option, tabbing to the next/previous field or keyboard focus item is automatic.
- If Tab is typed and the window contains an active editing field, it indicates that the user wants to tab to another field or keyboard focus item. You may want to validate the active field's edited text before allowing the user to tab to the new item. If this is the case, use GetEditString (or GetEditHandle) to obtain a copy of the edited text, then your application can check the string for errors. If an error is detected, display an appropriate alert box and ignore the doKeyDown event. If no error is detected, call the SaveFieldString routine to save the edited text as the field's string, then call the TabToFocus routine. If you initialized Tools Plus with the initAutoFocusChanges or initAutoSaveFieldString options, an active field's edited text is automatically saved without calling SaveFieldString.

- A Return key is reported only if the window does not have a default button or active field.
- If Enter is typed, it usually indicates that the user wants to enter the screen's data. This is the same process that is carried out by clicking an OK button. See "Tab" above, regarding validating and saving a field's edited text. An Enter key is reported only if the window does not have a default button.
- When the watch cursor is displayed, the only doKeyDown event that is reported is a ⌘-., which indicates that the user is halting a lengthy process. Other doKeyDown events are discarded.
- If the active window has an active editing field, Tools Plus automatically processes the Command key equivalents for the Edit menu's Undo, Cut, Copy, and Paste commands. These selections do not generate events.
- There are two situations where several keys produce the same key character: the Clear and Escape keys, and the F1 through F15 keys. In these cases, use the key code to differentiate the keys. Constants are provided for these key characters and key codes.
- A doKeyDown event is not generated when the user types in a desk accessory. These key strokes are handled automatically.

Valid Event Record Fields

`Event.Window` — **Window Number**: Active window number (frontmost standard window, tool bar, or a floating palette)

`Event.Key.Code` — **Key Number**: Number of the key that was pressed down. This key code is a key number that is not affected by the Caps Lock, Shift, Option, Command or Control modifiers.

`Event.Key.Chr` — **Key Character**: Character resulting from a key that was pressed down. This character is affected by the Caps Lock, Shift, Option, and/or Command modifiers.

`Event.Modifiers` — **Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down immediately before the key was pressed down.

`Event.Field` — **Editing Field Number**: Active editing field number when key was pressed.

```
CONST                                        {Key characters and key codes…            }
               EnterKey      = char($03);    {KEYS:  Key characters (ASCII) for common  }
               BackSpaceKey  = char($08);    {       non alpha-numeric keys.            }
               DeleteKey     = BackSpaceKey; {       In some cases, several keys        }
               TabKey        = char($09);    {       produce the same ASCII character.  }
               ReturnKey     = char($0D);    {       They can be differentiated by      }
               EscClearKey   = char($1B);    { ($1B) using the key code as indicated    }
               LeftArrowKey  = char($1C);    {       below.  Some keys are available    }
               RightArrowKey = char($1D);    {       only on an extended keyboard.      }
               UpArrowKey    = char($1E);    {                                          }
               DownArrowKey  = char($1F);    {                                          }
               HelpKey       = char($05);    {                                          }
               HomeKey       = char($01);    {                                          }
               DeleteFwdKey  = char($7F);    {                                          }
               EndKey        = char($04);    {                                          }
               PageUpKey     = char($0B);    {                                          }
               PageDownKey   = char($0C);    {                                          }
               FKey          = char($10);    { ($10) Function keys F1 to F15            }
               EscKeyCode    = $35;          { ($1B) KEY CODES used to differentiate    }
               ClearKeyCode  = $47;          { ($1B) between keys which produce         }
               F1KeyCode     = $7A;          {       the same key characters.           }
               F2KeyCode     = $78;          {                                          }
               F3KeyCode     = $63;          {                                          }
               F4KeyCode     = $76;          {                                          }
               F5KeyCode     = $60;          {                                          }
               F6KeyCode     = $61;          {                                          }
               F7KeyCode     = $62;          {                                          }
               F8KeyCode     = $64;          {                                          }
               F9KeyCode     = $65;          {                                          }
               F10KeyCode    = $6D;          {                                          }
               F11KeyCode    = $67;          {                                          }
               F12KeyCode    = $6F;          {                                          }
               F13KeyCode    = $69;          {                                          }
               F14KeyCode    = $6B;          {                                          }
               F15KeyCode    = $71;          {                                          }
```

doKeyInControl event

Indicator that the user pressed down a key, or a key is repeating and it was applied to a control.

The user has pressed down a key on the keyboard or numeric pad, or is holding a key, and that key was applied to an Appearance Manager-savvy control.

Programming Considerations

- Only Appearance Manager-savvy controls can process keystrokes. An example of such a control is the "Clock" control which allows the user to enter the time by typing in numeric values for hours, minutes and seconds. The Clock control also lets the user increase or decrease the selected hour, minute or second field by typing the up cursor or down cursor key.
- The keystroke is always applied to the control with the keyboard focus in the active window.
- Typing the Tab key automatically advances the keyboard focus. Similarly, typing Shift-Tab automatically reverses the keyboard focus. Note that the control indicated by Event.Button.Num or Event.ScrollBar.Num is the control that had the keyboard focus *before* the tab key was typed.

Valid Event Record Fields

`Event.Window`	**Window Number**: Active window number (frontmost standard window, tool bar, or a floating palette)
`Event.Key.Code`	**Key Number**: Number of the key that was pressed down. This key code is a key number that is not affected by the Caps Lock, Shift, Option, Command or Control modifiers.
`Event.Key.Chr`	**Key Character**: Character resulting from a key that was pressed down. This character is affected by the Caps Lock, Shift, Option, and/or Command modifiers.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down immediately before the key was pressed down.
`Event.Button.Num`	**Button Number**: Button number that processed the keystroke. A value of zero indicates that the affected control was implemented as a scroll bar.
`Event.ScrollBar.Num`	**Scroll Bar Number**: Scroll bar number that processed the keystroke. A value of zero indicates that the affected control was implemented as a button.
`Event.ListBox.Num`	**List Box Number**: List box number that processed the keystroke.

doKeyUp event

Indicator that the user released a key.

The user has released a key on the keyboard or numeric pad, and the key cannot be processed internally by Tools Plus. Normally the Macintosh does not report doKeyUp events nor do applications typically respond to them. See "Key-Up Events" in the Designing Your Application chapter for details on the SetEventMask routine.

Programming Considerations

- This event is generated only if your application has used the SetEventMask routine to make the Macintosh respond to key-up events, and if the active window does not have an active editing field.
- Tools Plus and desk accessories do not require key-up events, and therefore ignore them.
- There are two situations where several keys produce the same key character: the Clear and Escape keys, and the F1 through F15 keys. In these cases, use the key code to differentiate the keys. Constants are provided for these key characters and key codes.
- When the watch cursor is displayed, all doKeyUp events are discarded and are not generated.

Valid Event Record Fields

`Event.Window` — **Window Number**: Active window number (frontmost standard window, tool bar, or a floating palette)

`Event.Key.Code` — **Key Number**: Number of the key that was released. This key code is a key number that is not affected by the Caps Lock, Shift, Option, Command or Control modifiers.

`Event.Key.Chr` — **Key Character**: Character resulting from a key that was released. This character is affected by the Caps Lock, Shift, Option, and/or Command modifiers.

`Event.Modifiers` — **Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down immediately before the key was released.

```
CONST                                        {Key characters and key codes…             }
               EnterKey      = char($03);    {KEYS:  Key characters (ASCII) for common   }
               BackSpaceKey  = char($08);    {       non apha-numeric keys.              }
               TabKey        = char($09);    {       In some cases, several keys         }
               ReturnKey     = char($0D);    {       produce the same ASCII              }
               EscClearKey   = char($1B);    { ($1B) character.  They can be             }
               LeftArrowKey  = char($1C);    {       differentiated by using the key     }
               RightArrowKey = char($1D);    {       code, as indicated below.           }
               UpArrowKey    = char($1E);    {       Some keys are available only        }
               DownArrowKey  = char($1F);    {       on an extended keyboard.            }
               HelpKey       = char($05);    {                                           }
               HomeKey       = char($01);    {                                           }
               DeleteFwdKey  = char($7F);    {                                           }
               EndKey        = char($04);    {                                           }
               PageUpKey     = char($0B);    {                                           }
               PageDownKey   = char($0C);    {                                           }
               FKey          = char($10);    { ($10) Function keys F1 to F15             }
               EscKeyCode    = $35;          { ($1B) KEY CODES used to differentiate     }
               ClearKeyCode  = $47;          { ($1B) between keys which produce          }
               F1KeyCode     = $7A;          {       the same key characters.            }
               F2KeyCode     = $78;          {                                           }
               F3KeyCode     = $63;          {                                           }
               F4KeyCode     = $76;          {                                           }
               F5KeyCode     = $60;          {                                           }
               F6KeyCode     = $61;          {                                           }
               F7KeyCode     = $62;          {                                           }
               F8KeyCode     = $64;          {                                           }
               F9KeyCode     = $65;          {                                           }
               F10KeyCode    = $6D;          {                                           }
               F11KeyCode    = $67;          {                                           }
               F12KeyCode    = $6F;          {                                           }
               F13KeyCode    = $69;          {                                           }
               F14KeyCode    = $6B;          {                                           }
               F15KeyCode    = $71;          {                                           }
```

doListBox event

Indicator that user clicked or double-clicked in a list box.

The doListBox event reports that the user has clicked somewhere in a list box (on a line that contains text or a blank line). This event is not generated when the user just scrolls the list box by using its scroll bar.

Programming Considerations

- If your application treats double-clicks in list boxes like clicking the OK button, use the GetListBoxLine routine to determine which line was selected (or GetListBoxLines if multiple lines can be selected in the list box), then flash the OK button with the FlashButton routine.
- This event will never occur when the watch cursor is displayed, since list boxes cannot be clicked.
- A doListBox event is not generated when the user clicks list boxes in a dialog box or desk accessory. These events are handled automatically.

Valid Event Record Fields

`Event.Window`	**Window Number**: Window number containing the affected list box (frontmost standard window, tool bar, or a floating palette)
`Event.ListBox.Num`	**List Box Number**: List box number that was clicked by the user.
`Event.ListBox.DoubleClick`	**List Box's Double-click Status**: Was a line in the list box was double-clicked?
`Event.Event`	**The Event Manager's Event Record**: Event record as retrieved from the Event Manager. This will always be one of the following toolbox events: keyDown, autoKey, or mouseDown.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down when the user interacted with the list box.

doManualEvent event

Indicator that an event has been reported that can't be processed by Tools Plus.

The doManualEvent event reports that some type of event has occurred which may need to be processed by your application. This includes ten possible types of events as defined by the Event Manager, which are listed below. If your application does not make use of any of these events, it can ignore the doManualEvent event code.

`updateEvt`	Your application receives an update event as a doManual event for any window that is opened using standard toolbox routines when that window needs to be refreshed. This happens only in rare circumstances, such as running a custom "Open…" dialog, or in plug-ins when the host application's window needs to be updated. The Event.Event.message field contains a window pointer to a foreign (non-Tools Plus) window or dialog. In this case, you must call the toolbox's BeginUpdate and EndUpdate routines for the target window in order to clear the update event. If you don't, your application or plug-in will not receive doNothing events.
`diskEvt`	The Standard File Package takes care of all disk-inserted events, so applications using this package can ignore diskEvt events. If your application is unusual and it circumvents the Standard File package, your application will have to handle disk-inserted events itself.
`networkEvt`	Your application needs to respond to network events only if it is going to be communicating with the AppleTalk manager. If it is, you should read the appropriate documentation in Inside Macintosh regarding networks.
`driverEvt`	If your application does not intend to work with device drivers, it can ignore these events. Otherwise, you should read the appropriate documentation in Inside Macintosh regarding device drivers.

`app1Evt`
`app2Evt`
`app3Evt`
`app4Evt`
`osEvt`
	Application-defined events 1 through 4. These are events whose meaning is defined by your application. If your application does not define custom events, it can ignore these event codes. Note that app4Evt (or osEvt) will only be reported if your application does *not* respond to Suspend/Resume events. See the SIZE resource for details. If your application does respond to Suspend/Resume events, it will receive a doSuspend or doResume Tools Plus event.

`kHighLevelEvt`

High level events (known as Apple Events) will only be reported to your application as doManualEvent events under any of the following conditions:

- Your application's SIZE resource is set to not respond to High Level events, and your application posted a High Level event.
- Your application is running under System 6 or older, and it posted a high level event.
- Your application's SIZE resource is set to respond to High Level events, it is running under System 7 or later, but you have not written or installed an Apple Event handler routine for a specific Apple Event.

Valid Event Record Fields

`Event.Event`	**The Event Manager's Event Record**: Event record as retrieved from the Event Manager.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down immediately before the key was pressed down.

doMenu event

Indicator that the user selected a pull-down or hierarchical menu item.

The user selected a pull-down or hierarchical menu by either clicking in the menu bar and selecting an item, or by typing a Command key equivalent for a menu item.

Programming Considerations

- The menu's name in the menu bar is automatically highlighted (white letters on black background) when the user makes a menu selection. Your application must call MenuHilite(0) to remove the highlighting when it is finished with its process.
- If a hierarchical menu item is selected, the name of its ultimate parent menu is highlighted in the menu bar (white letters on black background). Your application must call MenuHilite(0) to remove the highlighting when it is finished with its process. If a Command key equivalent for a hierarchical menu item is typed, and the hierarchical menu is not ultimately attached to a pull-down menu, a doKeyDown or doAutoKey event is reported instead (as though the hierarchical menu did not exist).
- If your application determines that the selected menu item affects the active editing field, you may want to validate the field's edited text before allowing the user to complete the action. If this is the case, use GetEditString (or GetEditHandle) to obtain a copy of the edited text, then your application can check the string for errors. If an error is detected, display an appropriate alert box then use MenuHilite to remove the highlighting from the selected menu. If no error is detected, call the SaveFieldString routine to save the edited text as the field's string, then complete the menu's action.
- This event will never occur when a modal window is active, or when the watch cursor is displayed since the menu cannot be clicked and Command key equivalents are ignored.
- If the active window has an active editing field, Tools Plus will process the Edit menu's Undo, Cut, Copy, Paste, and Clear commands automatically. These selections will not generate doMenu events.
- A doMenu event is not generated when the user selects desk accessory menus. These selections are handled automatically.

Valid Event Record Fields

`Event.Menu.Num`	**Menu Number**: Menu number that was selected by the user. Menus 1 through 15 are pull-down menus, and 16 through 200 are hierarchical menus. The following constants are helpful for other menus: mAppleMenu = -1; Apple menu, "About…" item selected mHelpMenu = -2; Help menu (System 7.0 or higher)
`Event.Menu.Item`	**Menu Item**: Item number that was selected by the user
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down immediately before selecting the menu.

doMoveCursor event

Indicator that the cursor has moved into another window or cursor zone.

When the cursor moves between active windows, or between cursor zones, a doMoveCursor event is generated. You can think of this event as telling your application "the cursor is pointing at something different."

Programming Considerations

- If your want to determine what the cursor is pointing at, call GetCurrentCursorZone. It reports the window number, the cursor table used by that window, and the cursor zone beneath the cursor.

---

doMoveWindow event

Indicator that user moved a window.

The doMoveWindow event reports that the user has repositioned a window by dragging it by its title bar. Most applications will choose to ignore this event.

Programming Considerations

- The dragged window might not be the active window. The user can drag an inactive window by holding down the Command key before beginning the drag.
- If part or all of a window is exposed by the drag and needs to be refreshed, Tools Plus will report a doRefresh event for this window.
- Your application may call WindowStatus to obtain the window's new location in the screen's global co-ordinates.
- The only time this event can occur while the watch cursor is displayed is when a movable dialog (procID = movableBoxProc) is moved.
- Tools Plus may move a tool bar automatically in response to the user moving the application's menu bar to another monitor and changing the tool bar's co-ordinates. When this happens, Tools Plus reports a doMoveWindow event for the tool bar.
- A doMoveWindow event is not generated when the user drags a desk accessory. The process is handled automatically.

Valid Event Record Fields

`Event.Window`	**Window Number**: Window number that was repositioned.

---

doNothing event

No event has occurred. Also called a "null event."

The doNothing event indicates that the event queue is empty and that no event has occurred.

Programming Considerations

- If your application does "background processing," that is, it performs an on-going task while it waits for events, then your application should perform one "cycle" of its background process each time it receives a doNothing event. A single cycle of your application's background process should take no longer than 1/20 of a second. Any longer than that and the user will perceive reduced responsiveness. See the SetNullTime routine to set the interval at which your application receives doNothing events.
- Picture buttons with the "repeating events" option turned on may report doNothing events between button value changes. This causes no side-effects, but you should be aware of it.
- Scroll bars that are throttled may produce doNothing events between scroll bar value changes. This causes no side-effects, but you should be aware of it.

- To get the best performance from Tools Plus, do not alter any of the values in the event record.
- Consider using a Timer in place of responding to a doNothing events. See the section on Timers in this chapter for details.

Valid Event Record Fields

`Event.What`	Set to zero (0)
`Event.Event`	**The Event Manager's Event Record**: Event record as retrieved from the Event Manager.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down.

---

doOpenApplication event

This event is reported only if your application is launched without any files to open or print. When your application gets this event, it should open a blank document or perform some similarly suitable operation.

Programming Considerations

- If your application fails to correctly execute its "open application" code for any reason, such as requiring the user to launch the application only by double clicking a document or dragging a document onto the application, display an appropriate message to alert the user. You may also choose to call the QuitToolsPlus to quit your application. You can also optionally call the SetEventError routine with a relevant error code if want to provide feedback to the calling application as to why your application could not open.
- If your application is Apple Event aware (i.e., its 'SIZE' resource is set with the "High level event aware" flag on), and it is running under System 7 or later, Tools Plus reports the "open application" Apple Event as a doOpenApplication event.
- If your application is not Apple Event aware, or if it is running under System 6 or older, the ProcessEvents routine synthesizes a doOpenApplication event if it is appropriate.
- While your application is responding to the doOpenApplication event, Tools Plus suspends the reporting of Apple Events. That way, you can be sure your application will have the opportunity to perform all necessary work before it gets additional requests to open documents, print documents, or quit.
- In an Apple Event aware applications running on System 7 or later, you can override Tools Plus's default reporting of a doOpenApplication event by installing your own Apple Event Handler routine for the "open application" event.

Valid Event Record Fields

`Event.Event`	**The Event Manager's Event Record**: Event record as retrieved from the Event Manager. This record is empty if the ProcessEvents routine synthesized a doOpenApplication event.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down. This record is empty if the ProcessEvents routine synthesized a doOpenApplication event.

---

doOpenDocuments event

This event indicates that the Finder (or another application) has requested that your application open one or more documents. This can occur as a result of the user dragging a document onto your application, double-clicking a document that was created by your application, or a client application making the request of your application via an Apple Event. From a high level, the code you use to step through the list of documents that need to be opened is as follows:

C
```
for (theIndex = 1; theIndex <= CountNumberOfFiles(); theIndex++)
  if (GetIndexFileFSS(theIndex, &myFSS))
    MyOpenDoc(&myFSS);
```

Pascal
```
for theIndex := 1 to CountNumberOfFiles do
  if GetIndexFileFSS(theIndex, myFSS) then
    MyOpenDoc(myFSS);
```

You will likely want to modify the above code to add logic that stops going through the file list if your application cannot open any more files, or if it encounters a error. The GetIndexFileFSS routine retrieves an FSSpec record that can be used to open a document in System 7 or later. For 680x0 applications running on System 6 or older, or those 680x0 applications that are not Apple Event aware, use the GetIndexFile routine in place of GetIndexFileFSS.

Programming Considerations

- This event may be reported if your application is Apple Event aware (i.e., its ‘SIZE’ resource is set with the “High level event aware” flag on) and your application is runing under System 7 or later.
- If your 680x0 application is running under System 6 or older, or it is not Apple Event aware, the ProcessEvents routine may report a doOpenDocuments event if one or more documents need to be opened when your application is first launched. This simulates the effect of an “open documents” Apple Event when Apple Events are not available.
- Call the SetEventError routine if your application fails to open all the requested documents, or chooses not to open any of them for any reason.
- While your application is responding to the doOpenDocuments event, Tools Plus suspends the reporting of Apple Events. That way, you can be sure your application will have the opportunity to open all the requested documents before it gets additional requests to open documents, print documents, or quit.
- If the opening and displaying of documents is a lengthy process, consider displaying a modal window with an appropriate message (such as “Please wait while documents are opening”) and periodically calling Process1EventWhileBusy to give other processes some processing time. The modal window will prevent the user from activating other applications or using pull-down menus while your application opens the required documents.
- In an Apple Event aware applications running on System 7 or later, you can override Tools Plus’s default reporting of doOpenDocuments events by installing your own Apple Event Handler routine for the “open documents” event.

Valid Event Record Fields

`Event.Event`	**The Event Manager’s Event Record**: Event record as retrieved from the Event Manager. This record is set to zeros if the ProcessEvents routine synthesized a doOpenDocuments event.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down. This record is set to zeros if the ProcessEvents routine synthesized a doOpenDocuments event.

doPictButton event

Indicator that user has clicked a picture button, or is holding down a repeating picture button.

The doPictButton event reports that the user has clicked a picture button in the active window. The doPictButton event is also reported if the user holds down a repeating picture button. Repeating picture button's will produce events as long as: [1] the mouse button is held down, [2] the button does not reach the end of its range, and [3] the mouse pointer stays inside the picture button's region.

Programming Considerations

- If the "automatic value change" option is not on, your application has to set the picture button's value by using SetPictButtonVal or SetPictButtonValSelect.
- If you have created a picture button that is "lock selected" with the "automatic value change" option off, and you've done this to validate the picture button before proceeding to the next stage, do the following: validate the button. If it can proceed to the next stage, use SetPictButtonValSelect to deselect the button and increment its value, otherwise use SetPictButtonValSelect to deselect the button and leave its value unchanged.
- If the user clicked a picture button that is functioning like a radio button, use SelectPictButton to deselect the other picture buttons in the group. Note that a panel can be used to automatically deselect other picture buttons in the group.
- This event will never occur when the watch cursor is displayed, since picture buttons cannot be clicked. The exception is if the WatchCursorButtons routine has been used to allow the watch cursor to click picture buttons.
- Picture buttons with the "repeating events" option turned on may produce doNothing events between button value changes. This will cause no ill side-effects, but you should be aware of this.
- A doPictButton event is not generated when the user clicks picture buttons in another application, control panel or desk accessory. These events are handled automatically.

Valid Event Record Fields

`Event.Window` — **Window Number**: Window number containing the selected picture button (frontmost standard window, tool bar, or a floating palette).

`Event.Button.Num` — **Button Number**: Picture button number that was clicked by the user.

`Event.Button.Part` — **Button Part**: Part of button that was clicked by user. A value of inButton is returned for all buttons except those that are split. Buttons with a left/right split return inDownButton when the left side is clicked and inUpButton when the right side is clicked. Buttons with a top/bottom split return inDownButton when the bottom is clicked and inUpButton when the top is clicked.

`Event.Modifiers` — **Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down when the picture button was clicked (during the mouse-down).

```
CONST                                {Picture button parts…                         }
               inButton     = 10;    {A picture button that is not split            }
               inUpButton   = 20;    {Top or right side of a split button was clicked   }
                                     {  (button value is being incremented).        }
               inDownButton = 21;    {Bottom or left side of a split button was clicked }
                                     {  (button value is being decremented).        }
```

doPopUpMenu event

Indicator that the user selected a pop-up menu item.

Programming Considerations

- If your application determines that the item selected by the user should be checked, use CheckPopUp or PopUpMark to mark the item with a check mark or other special character. By default, the previously selected item is automatically unchecked (although you can override this behavior when creating the pop-up menu).
- If your application determines that the selected pop-up menu item affects the active editing field, you may want to validate the field's edited text before allowing the user to complete the action. If this is the case, use GetEditString (or GetEditHandle) to obtain a copy of the edited text, then your application can check the string for errors. If an error is detected, display an appropriate alert box. If no error is detected, call the SaveFieldString routine to save the edited text as the field's string, then complete the menu's action.
- A doPopUpMenu event is not generated when the user selects desk accessory menus. These selections are handled automatically.

Valid Event Record Fields

`Event.Window`	**Window Number**: Win(frontmost standard window, tool bar, or a floating palette)
`Event.Menu.Num`	**Menu Number**: Pop-up menu number that was selected by the user
`Event.Menu.Item`	**Menu Item**: Item number that was selected by the user
`Event.Menu.SubMenu`	**Submenu Number**: Hierarchical menu number in which the item was selected (0 if a submenu is not used)
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down immediately before selecting the pop-up menu.

doPreRefresh event

A window's contents *may* be refreshed (redrawn) before Tools Plus refreshes its own objects.

Most applications will ignore this event. A doPreRefresh event is reported when a window is completely or partially obscured, then becomes uncovered and needs to be redrawn. The doPreRefresh event lets your application draw objects *before* Tools Plus draws its own objects. Tools Plus takes care of redrawing buttons, panels, scroll bars, editing fields, list boxes, pop-up menus, picture buttons and custom controls. Your application must take care of refreshing anything else, such as icons, pictures, lines, or text that it may have drawn to the window. After Tools Plus draws it own objects, your application gets a doRefresh event for the same window to let it do any drawing *after* Tools Plus's objects.

An example of the doPreRefresh event's usefulness is if your application displays a picture as a window's background. When your application receives a doPreRefresh event, it can draw the background picture only. When your event handler routine finishes execution, Tools Plus refreshes its own objects, then it reports a doRefresh event to let you draw anything after Tools Plus's user interface elements.

When a doPreRefresh event is reported, the window's "update region" defines the exact area that needs to be refreshed. Although you can redraw everything in the window, the visible drawing is limited to the parts of the window that are visible and that need refreshing.

Programming Considerations

- If your application did not put anything in the window except buttons, panels, picture buttons, scroll bars, editing fields, list boxes, pop-up menus, and custom controls, it should ignore this event.

- If your application responds to this event, it should do so in the following manner:

```
for theScreen := 1 to NumberOfScreens do        {Repeat drawing for each monitor in which the    }
  begin                                         { window appears.                                }
    BeginUpdateScreen(theScreen);               {Drawing area reduced to the specified screen    }

    {insert your color-dependent drawing code here}

    EndUpdateScreen;                            {End the drawing for this monitor, and restore   }
  end;                                          { the window's visible (drawing) region.         }

  {insert your color-independent drawing code here}
```

- By default, when your application gets this event, Tools Plus has already done the following for you:

```
CurrentWindow(Event.Window);                    {Make the affected window the current grafPort   }
BeginUpdate(WindowPointer(Event.Window));       {Drawing will occur only within the area that    }
                                                { needs refreshing.                              }
```

  When your application gets this event, all it has to do is draw to the window as required. When your event handler finishes executing, Tools Plus automatically does the following:

```
EndUpdate(WindowPointer(Event.Window));         {End the update for the window and restore the   }
                                                { window's visible (drawing) region.             }
```

- If you need more control over window drawing when your application responds to this event, first create the window by adding the wManualUpdate option to the window's spec parameter. When your application gets this event, it should respond in the following manner:

```
CurrentWindow(Event.Window);                    {Make the affected window the current grafPort   }
BeginUpdate(WindowPointer(Event.Window));       {Drawing will occur only within the area that    }
                                                { needs refreshing.                              }

{insert your window drawing code here (see earlier example) }

EndUpdate(WindowPointer(Event.Window));         {End the update for the window                   }
CurrentWindowReset;                             {Reset the current window to be the same as      }
                                                {   the active window (optional)                 }
```

  In the example above, BeginUpdate temporarily sets the window's visRgn (visible region where drawing occurs) to the intersection of the visRgn and updateRgn (region requiring updating). This effectively restricts drawing to the area that needs updating. During a doPreRefresh event, the affected area may include Tools Plus objects that will be automatically refreshed after your event handler finishes executing. During a doRefresh event, the affected area excludes Tools Plus objects that have already been drawn thereby preventing you from drawing over them. EndUpdate restores drawing to the rest of the window.

- It is possible that several windows will need to be refreshed simultaneously if, for instance, a closing window exposed three windows behind it. Tools Plus reports a doPreRefresh event followed by a doRefresh for each window that needs refreshing.
- Unlike ordinary Macintosh applications, this event is not generated when a window is first opened. If you need this to happen, you can add the wRefresh option to the window's spec parameter when opening a window. Alternatively, you can open the window as hidden, create any Tools Plus objects, then display the window.
- This event is not generated for desk accessories, the Dialog Manager, alerts, or Dynamic Alerts. These events are handled automatically.
- When responding to this event, your application should limit itself to activities that pertain only to refreshing the specified window. Some of the things to avoid are creating, modifying, deleting, moving, resizing, showing and hiding user interface elements.

Valid Event Record Fields

`Event.Window` **Window Number**: Window number that needs refreshing.

Programming Tips:

1. If your application does a lot of drawing in a window, you may be able to speed up the refreshing process by checking if each object *needs* to be redrawn. On an object-by-object basis, use Tools Plus's RectIsVisible or RgnIsVisible routines to determine if the object needs to be redrawn. If not, your application can save time by not redrawing that object.
2. If you need to draw things to the window such as line art, pictures, or other dynamic imagery, it may be a good idea to do this drawing only in response to a doPreRefresh or doRefresh event. When you first create your window, create it as hidden and create any Tools Plus user interface elements such as buttons and list

boxes. Then display the window which will generate a doPreRefresh event, Tools Plus objects are drawn, then a doRefresh event is generated. To the user, it just looks like everything is coming up quickly at the same time.

3 If your application follows the second tip (above), your window refreshing routine can paint the window with a color or pattern, then draw text and/or lines on top.

4 It may take some time to refresh your window if you need to draw a large picture or a number of pictures. A worst-case example is having a 32-bit picture as the window's backdrop. You can speed up refreshing by breaking up the picture into smaller pictures and redrawing only those that need refreshing. Tools Plus's RectIsVisible and RgnIsVisible will help you determine if a picture needs refreshing.

5. If you have a floating palette or toolbar that has a number of picture buttons, you can make it refresh much faster by doing the following in response to a doPreRefresh event:
   1 When you get a doPreRefresh event, draw a picture of the picture buttons. The picture will likely have all picture buttons drawn in a deselected state. This image will give the illusion that the entire palette is redrawn instantly.
   2 Set your window's clip region to exclude the image of the picture buttons. This prevents Tools Plus from redrawing the picture buttons.
   3 If you can, include selected picture buttons in the clip region to let Tools Plus redraw only those buttons.

   Tools Plus redraws the picture buttons that need refreshing and are within the clip region you have defined.
   Do the following in response to a doRefresh event:
   1 Reset the window's clip region to include the entire window.
   2 If you were not able to perform step 3 above, first deselect, then select the required picture buttons (using SelectPictButton) to overwrite the picture of the deselected buttons.

---

doPrintDocuments event

This event indicates that the Finder (or another application) has requested that your application print one or more documents. This can occur as a result of the user selecting one or more document that were created by your application then choosing the File menu's Print command from the Finder, or a client application making the request of your application via an Apple Event. From a high level, the code you use to step through the list of documents that need to be opened is as follows:

C

```
for (theIndex = 1; theIndex <= CountNumberOfFiles(); theIndex++)
  if (GetIndexFileFSS(theIndex, &myFSS))
    MyPrintDoc(&myFSS);
```

Pascal

```
for theIndex := 1 to CountNumberOfFiles do
  if GetIndexFileFSS(theIndex, myFSS) then
    MyPrintDoc(myFSS);
```

You will likely want to modify the above code to add logic that stops going through the file list if your application cannot print any more files, if it encounters a error, or if the user halts printing. The GetIndexFileFSS routine retrieves an FSSpec record that can be used to open a document in System 7 or later. For 680x0 applications running on System 6 or older, or those 680x0 applications that are not Apple Event aware, use the GetIndexFile routine in place of GetIndexFileFSS.

Programming Considerations

- This event may be reported only if your application is Apple Event aware (i.e., its 'SIZE' resource is set with the "High level event aware" flag on) and your application is runing under System 7 or later.
- If your 680x0 application is running under System 6 or older, or it is not Apple Event aware, the ProcessEvents routine may report a doPrintDocuments event if one or more documents need to be printed when your application is first launched. This simulates the effect of an "print documents" Apple Event when Apple Events are not available.
- Call the SetEventError routine if your application fails to print or chooses not to print all the requested documents for any reason.

- While your application is responding to the doPrintDocuments event, Tools Plus suspends the reporting of Apple Events. That way, you can be sure your application will have the opportunity to print all the requested documents before it gets additional requests to open documents, print documents, or quit.
- It is best if you do not open a window for each file that needs to be printed. If you must open a window, open it off screen so that the user does not see it.
- It is likely that opening, displaying and printing documents will be a lengthy process, so it's a good idea to display a modal window with an appropriate message (such as "Printing {*file name*}. To cancel printing, type ⌘-.") and periodically call Process1EventWhileBusy to give other processes some processing time. The modal window will prevent the user from activating other applications or using pull-down menus while your application opens and prints the required documents.
- In an Apple Event aware applications running on System 7 or later, you can override Tools Plus's default reporting of doPrintDocuments events by installing your own Apple Event Handler routine for the "print documents" event.

Valid Event Record Fields

`Event.Event`	**The Event Manager's Event Record**: Event record as retrieved from the Event Manager. This record is set to zeros if the ProcessEvents routine synthesized a doPrintDocuments event.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down. This record is set to zeros if the ProcessEvents routine synthesized a doPrintDocuments event.

---

doQuitApplication event

This event is reported if the Finder or another application wants your application to quit. This event may come as a result of the user shutting down or restarting their Macintosh, or immediately following a doPrintDocuments event that launched your application.

Programming Considerations

- When your application gets a doQuitApplication event, it should go through the same shutdown process as though the user had selected your application's File menu's Quit item. Specifically, if any open documents have been modified, ask the user if the changes should be saved before quitting, save the documents and preferences, close windows and so on. When your application's shutdown process is completed, call the QuitToolsPlus routine to inform Tools Plus that your application has finished processing events. If your application fails to call the QuitToolsPlus routine, then Tools Plus will understand this to mean your application did not quit, and it is still running. You can also optionally call the SetEventError routine with a relevant error code if want to provide feedback to the calling application why your application could not quit.
- If your application is suspended (i.e., it is not active), the doQuitApplication event brings your application to the foreground and activates it.
- If your application is Apple Event aware (i.e., its 'SIZE' resource is set with the "High level event aware" flag on), and it is running under System 7 or later, Tools Plus reports the "quit application" Apple Event as a doQuitApplication event.
- If your application is not Apple Event aware, or if it is running under System 6 or older, this event will not normally be reported. The one exception is for 680x0 applications that are launched as a result of the user selecting one or more documents, and choosing the File menu's Print command from the Finder. In this case, your application will first receive a doPrintDocuments event followed by a doQuitApplication event.
- In an Apple Event aware applications running on System 7 or later, you can override Tools Plus's default reporting of a doQuitApplication event by installing your own Apple Event Handler routine for the "quit application" event.

Valid Event Record Fields

`Event.Event`	**The Event Manager's Event Record**: Event record as retrieved from the Event Manager. This record is empty if the ProcessEvents routine synthesized a doQuitApplication event.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down. This record is empty if the ProcessEvents routine synthesized a doQuitApplication event.

doRefresh event

A window's contents need to be refreshed (redrawn) after Tools Plus has refreshed its own objects.

Your application needs to respond to the doRefresh event and/or a doPreRefresh event only if you have drawn objects in the window. Tools Plus's user interface elements (such as buttons, panels, pop-up menus, editing fields, etc.) are automatically refreshed. A doRefresh event is reported when a window is completely or partially obscured, then becomes uncovered and needs to be redrawn. Tools Plus takes care of redrawing buttons, panels, scroll bars, editing fields, list boxes, pop-up menus, picture buttons and custom controls. Your application must take care of refreshing anything else, such as icons, pictures, lines, or text that it may have drawn to the window. When your application receives this event, Tools Plus will have already drawn its objects.

When a doRefresh event is reported, the window's "update region" defines the exact area that needs to be refreshed, excluding the objects drawn by Tools Plus (buttons, panels, picture buttons, scroll bars, editing fields, list boxes, pop-up menus, and custom controls). Although you can redraw everything in the window, the actual drawing is limited to the parts of the window that are visible and which need refreshing. Tools Plus changes the window's update region to exclude its own objects, thus preventing you from accidentally drawing over them.

Your application can ignore the doRefresh event if you have chosen to do all the necessary drawing in response to a doPreRefresh event, or if the only objects in the window are drawn by Tools Plus (buttons, panels, picture buttons, scroll bars, editing fields, list boxes, pop-up menus, and custom controls).

Programming Considerations

(see doPreRefresh event)

Valid Event Record Fields

`Event.Window`	**Window Number**: Window number that needs refreshing.

Programming Tips:

(see doPreRefresh event)

doResume event

Indicator that your application has been resumed.

The doResume event reports that your application has become the active application. This occurs under MultiFinder or System 7 (or later) after your application has become suspended (see the doSuspend event).

Valid Event Record Fields

`Event.Event`	**The Event Manager's Event Record**: Event record as retrieved from the Event Manager.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down.

Programming Considerations

- If your application is cutting, copying or pasting specialized objects (i.e., not text that is handled by text editing fields), you may want to check the clipboard to see if it contains the specialized objects that you can paste, then copy them to your local scrap.

---

doScrollBar event

Indicator that user clicked in a scroll bar.

The doScrollBar event reports that the user has clicked somewhere in a scroll bar or a custom CDEF that is made to behave like a scroll bar by Tools Plus. The scroll bar has 5 parts: up button, down button, “page up” region, “page down” region, and the thumb. In all cases, your application should respond to the scroll bar event by scrolling the window’s contents (or portion thereof) and drawing any parts that have been newly revealed due to scrolling.

Programming Considerations

- If the scroll bar’s thumb was moved, your application can obtain the current value of the scroll bar by calling GetScrollBarVal. By using this value, your application can determine how much to scroll.
- If the scroll bar’s up button, down button, “page up” region, or “page down” region was clicked, your application must scroll the window by the correct amount, then change the scroll bar’s value by using SetScrollBarVal.
- If the up arrow or “page up” region is clicked, subtract the correct amount from the current value. If the down arrow or “page down” region is clicked, add the correct amount to the current value.
- If the user holds the mouse button down in the up arrow, “page up” region, down arrow, or “page down” region, Tools Plus returns a series of doScrollBar events as readily as your event handler can respond to them. Your application doesn’t have to be aware of this, but it is nice to know that scroll bar events will not pile up in the queue.
- If the user moves the scroll bar’s thumb and returns it to its original place during the same drag, an event is not generated.
- This event will never occur when the watch cursor is displayed, since scroll bars cannot be clicked.
- Scroll bars that are part of a list box are handled automatically.
- A doScrollBar event is not generated when the user clicks scroll bars in a dialog box or desk accessory. These events are handled automatically.

Valid Event Record Fields

`Event.Window`	**Window Number**: Window number containing the affected scroll bar (frontmost standard window, tool bar, or a floating palette)
`Event.ScrollBar.Num`	**Scroll Bar Number**: Scroll bar number that was clicked by the user.
`Event.ScrollBar.Part`	**Scroll Bar Part**: Part of scroll bar that was clicked by user.

```
CONST                               {Scroll Bar parts                      }
               inUpButton   = 20;   {up arrow of a scroll bar              }
               inDownButton = 21;   {down arrow of a scroll bar            }
               inPageUp     = 22;   {"page up" region of a scroll bar      }
               inPageDown   = 23;   {"page down" region of a scroll bar    }
               inThumb      = 129;  {thumb of a scroll bar                 }
```

---

doSuspend event

Indicator that your application is about to be suspended.

The doSuspend event reports that your application will be suspended (or become a "background" application) as soon as the event handler finishes executing. This occurs under MultiFinder or System 7 (or later) when a desk accessory or other application is launched, or when an inactive window belonging to a desk accessory or another application is activated.

Programming Considerations

- If your application is cutting, copying or pasting specialized objects (i.e., not text that is handled by editing fields), copy this material from your local scrap to the clipboard.

Valid Event Record Fields

`Event.Event`	**The Event Manager's Event Record**: Event record as retrieved from the Event Manager.
`Event.Modifiers`	**Modifier Flags**: Flags indicating if the Caps Lock, Shift, Option, Command or Control modifier keys were down.

---

doTimer event

Indicator that a Timer event was generated.

The doTimer event reports that a Timer is reporting a timed event, and it is calling your event handler routine to respond appropriately.

Programming Considerations

- A Timer may not report its event exactly at the expected time if the system is too busy to allow this to happen.
- If a Timer is set to report its events at a specific frequency (eg: 90 events per minute), a heavy processor workload or a high Timer frequency may prevent the Timer from ever catching up to where it should be. If this happens, do not use the timerLockTimerToCount option.
- When the timerLockTimerToCount option is not used, the event record's Event.Timer.Count field indicates the number of events that *should* have been reported even though the system may have been too busy to call your event handler routine each time.

Valid Event Record Fields

`Event.Window`	**Window Number**: Window for which the event is being reported. Will be zero (0) is the Timer is not associated with a window.
`Event.Timer.Num`	**Timer Number**: Timer number that generated the Timer event.
`Event.Timer.Count`	**Timer's Event Count**: Sequential event counter (i.e., the number of times this Timer *should* have reported an event).
`Event.Timer.NextTime`	**Next time Timer will report an event**: Time in ticks from boot time when this Timer will report its next event.

---

doZoomWindow event

Indicator that user has changed a window's size and/or location by zooming.

The doZoomWindow event reports that the user has changed a window's size and/or position by clicking the "zoom box." This will always occur on the active window. Applications will typically ignore this event, or treat it similarly to the doGrowWindow event.

Programming Considerations

- Your application can call the WindowStatus routine to obtain the window's new width and height in pixels, as well as its new location in the screen's global co-ordinates.
- The window's entire contents are erased during the zoom. Tools Plus will report a doRefresh event for this window. Responding to the doRefresh event will regenerate the window's contents.
- This event will never occur when the watch cursor is displayed, since the zoom box cannot be clicked.
- A doZoomWindow event is not generated when the user click's a desk accessory's zoom box. The process is handled automatically.

Valid Event Record Fields

`Event.Window` **Window Number**: Window number that was zoomed (frontmost standard window).

"Field To Event" Cross reference

The following table provides a quick cross reference between each field in the event record, and which events make use of that field.

Field	Events
`Event.Button`	doButton, doClickToFocus, doKeyInControl, doPictButton
`Event.Event`	doChgWindow, doClick (first mouse down only), doClickControl, doClickToFocus, doListBox, doManualEvent, doNothing, doOpenApplication, doOpenDocuments, doPrintDocuments, doQuitApplication, doResume, doSuspend
`Event.Field`	doAutoKey, doChgInField, doClickToFocus, doKeyDown
`Event.Key…`	doAutoKey, doKeyDown, doKeyInControl, doKeyUp
`Event.ListBox…`	doClickToFocus, doListBox
`Event.Menu…`	doMenu, doPopUpMenu
`Event.Modifiers`	doAutoKey, doButton, doClickControl, doClickToFocus, doGoAway, doKeyDown, doKeyInControl, doKeyUp, doListBox, doManualEvent, doMenu, doNothing, doOpenApplication, doOpenDocuments, doPictButton, doPopUpMenu, doPrintDocuments, doQuitApplication, doResume, doSuspend
`Event.Mouse…`	doClick
`Event.ScrollBar…`	doClickToFocus, doKeyInControl, doScrollBar
`Event.Timer…`	doTimer
`Event.What`	(all events) doAutoKey, doButton, doChgInField, doChgMonitorSettings, doChgWindow, doClick, doClickControl, doClickDesk, doClickToFocus, doGoAway, doGrowWindow, doKeyDown, doKeyInControl, doKeyUp, doListBox, doManualEvent, doMenu, doMoveCursor, doMoveWindow, doNothing, doOpenApplication, doOpenDocuments, doPictButton, doPopUpMenu, doPrintDocuments, doQuitApplication, doRefresh, doResume, doScrollBar, doSuspend, doTimer, doZoomWindow
`Event.Window`	doAutoKey, doButton, doChgInField, doChgWindow, doClick, doClickControl, doClickToFocus, doGoAway, doGrowWindow, doKeyDown, doKeyInControl, doKeyUp, doListBox, doMoveWindow, doPictButton, doPopUpMenu, doRefresh, doScrollBar, doTimer, doZoomWindow

17 Color Drawing & Multiple Monitors

If your application does not take advantage of the Macintosh's color capabilities (as determined when initializing Tools Plus by way of the InitToolsPlus routine), you can skip this section. This chapter deals entirely with drawing in color, as well as with the implications of making your application compatible with Macintoshes that have multiple monitors. You should already be familiar with the relevant material in Inside Macintosh to understand the concepts behind color drawing and Color QuickDraw.

Several routines used for highlighting and unhighlighting areas are also useful to developers who create a black and white interface. Those routines are: UseHiliteColor, UseHiliteText, HiliteRect and HiliteRgn. Also, if your application is merely testing for the presence or absence of Color QuickDraw, you can just use the HasColorQuickDraw routine.

If you are using the simplest form of color drawing, that is drawing objects or text using fixed colors and letting Color QuickDraw resolve all color discrepancies between monitors (or different settings on the same monitor), you don't *have* to read this section. You should read this section if your application behaves differently depending any of the following:

- The number of available colors or shades of gray (black and white, 256 colors, 256 grays, millions of colors, etc.)
- Whether Color QuickDraw is available or unavailable on the Macintosh that is running your app.
- The orientation, or changes in orientation of multiple monitors (how one monitor is positioned in relation to another)
- Menu bar's height is changed (can be done by the user in Mac OS 8.5 or later)
- Changes in monitor resolution

Within this section, all references to the term "color" also include "gray-scale," or shades of gray unless otherwise stated. The term "color" does not include "black and white" such as the original 9 inch monochrome Macintosh monitors, or when a monitor is set to "black and white" in the Monitors Control Panel.

Note: All Tools Plus user interface elements (such as picture buttons and pop-up menus, etc.) take care of themselves when drawing on various monitor settings and multiple monitor setups. You don't have to do anything to make them work. This section is for drawing objects yourself.

Using One Monitor

The majority of Macintoshes have only a single monitor, but even with a single monitor, your application may choose to behave differently depending on whether the user has set the monitor to 1-bit (black and white), 2-bits (4 colors), 4-bits (16 colors), 8-bits (256 colors), 16-bits (thousands of colors) or 24-bits (millions of colors), or to gray-scale of equivalent depth. Realize that the user can change a monitor's settings in real time by using the appropriate Control Panel or Control Strip. Also, third party products exist that let the user change the monitor's settings without your application being suspended. An example is a product in which a "monitor settings" pop-up menu appears when the user control-clicks.

Never assume that the user has a specific monitor, or that the monitor is set to a specified number of colors. Before drawing or refreshing a window's contents, your application should determine the monitor's *depth* in pixels (by using the ScreenDepth routine), and possibly if the monitor is set to colors or gray-scale (by using the ScreenHasColors routine), then draw accordingly.

Using Multiple Monitors

With Tools Plus, you can make your application "multi-monitor capable" almost as simply as making it respond appropriately to the number of colors available on the main monitor. This is accomplished by drawing the *color-dependent* contents of a window once for each monitor the window intersects. If the window is entirely on one monitor, its contents are drawn only once. If it intersects two monitors, its contents are drawn once for the first monitor and a second time for the second monitor. The portions of the window that are *color-independent* (i.e., they are drawn

the same no matter what the monitor's color capabilities are), need to be drawn only once.

Logical Screens

Fortunately, Tools Plus has several routines that make the task of drawing a window's contents on multiple monitors an uncomplicated process, that is by drawing on *logical* screens instead of physical monitors. A logical screen is the combined area of all available monitors that have the same [1] screen depth (number of colors) and [2] setting of "colors" or "grays."

If your application is running on a Macintosh with three monitors, each being set to 256 colors, the NumberOfScreens routine reports 1 logical screen, that being the sum of all three monitors. Your application can then update all three monitors simultaneously. Conversely, if two of those monitors are set to 256 colors and the third one is set to black and white, NumberOfScreens reports 2 logical screens. Your application can then update the pair of 256 color monitors simultaneously (1st logical screen) then the black and white monitor (2nd logical screen). Note that NumberOfScreens differentiates between color and gray-scale monitors, even if they are set to the same pixel depth.

The following example illustrates a typical application's source code used to refresh a window. It accounts for multiple monitors, and for objects that are color-dependent (they are drawn differently depending on the monitor's settings), as well as color-independent (drawn identically, regardless of the monitor's settings).

```
BeginUpdate(WindowPointer(Event.Window));       {Drawing will occur only within the area that    }
                                                { needs refreshing (includes all monitors).      }
for theScreen := 1 to NumberOfScreens do        {Repeat drawing for each monitor in which the    }
  begin                                         { window appears, one screen at a time.          }
    BeginUpdateScreen(theScreen);               {Drawing area reduced to the specified screen    }

    {insert your color-dependent drawing code here}

    EndUpdateScreen;                            {End the drawing for this monitor, and restore   }
  end;                                          { the window's visible (drawing) region.         }

{insert your color-independent drawing code here}

EndUpdate(WindowPointer(Event.Window));         {End the update for the window                   }
```

Note that this programmatic style is only a recommendation. You may decide to have self-contained color-dependent routines accessed from within sections of color-independent code.

BeginUpdate is a Macintosh Toolbox routine that sets the specified window's visible region to be the same as the update region. This limits drawing to the region of the window that needs updating (refreshing). You need to call BeginUpdate and EndUpdate only if you are responding to a doRefresh or doPreRefresh event, and only if you used the wManualUpdate option when you opened the window.

The *NumberOfScreens* routine reports the number of logical screens (not physical monitors) that have unique settings.

Call *BeginUpdateScreen* just before you begin drawing a window's color-dependent contents. BeginUpdateScreen temporarily saves a copy of the current window's visible region (visRgn), and replaces it with an intersection of the window's visible region and the specified logical screen. This limits drawing to the area of the window that is visible only on the specified logical screen. You can use the ScreenDepth and ScreenHasColors routines to determine the color characteristics of the specified logical screen.

Call *EndUpdateScreen* after you have completed drawing a window's color-dependent contents. It restores the window's visible region (visRgn) to its original value, that being prior to being altered by BeginUpdateScreen.

EndUpdate is a Macintosh Toolbox routine that restores the specified window's visible region to its original value before you called the toolbox's BeginUpdate, that being the entire part of the window that is visible. This restores drawing ability to the entire window. You need to call BeginUpdate and EndUpdate only if you are responding to a doRefresh or doPreRefresh event, and only if you used the wManualUpdate option when you opened the window.

If you want to optimize the drawing of color-dependent windows, your application may want to check the current window's visRgn after calling BeginUpdateScreen. If the visRgn is empty, it means the window does not intersect the specified logical screen, and therefore does not need updating. Use Tools Plus's RgnIsVisible or RectIsVisible routines to determine if an object you are about to draw is visible. If it is not, you can likely save time by not drawing it.

Physical Monitors

Tools Plus also includes routines that help your application determine information about the physical monitors that are attached and running on your Macintosh. NumberOfMonitors reports the number of monitors that are attached. MonitorDepth reports a single monitor's depth in pixels. MonitorhasColors reports if a monitor is set to display colors or gray-scale/black and white. MonitorGDevice returns a handle to a monitor's Graphics Device, for developers who want to do their own, more sophisticated graphics manipulation. MainMonitorNumber returns the monitor number that contains the menu bar (although it can be hidden), and optionally a tool bar.

Detecting Monitor and Screen Changes

Tools Plus is optimized for maximum performance, so it does not constantly check to see if the monitor settings have changed. Mac OS does not have any mechanism for reporting changes in these settings, so Tools Plus quickly sets an internal flag whenever the settings *may* have changed, and that is each time it gets an event. Whenever your application uses a routine that must report on information pertaining to physical monitors or logical screens, Tools Plus first checks to see if the settings may have changed, and if so, it recalculates the required information as per the current settings by calling the CheckForMonitorChanges routine. This way, your application can call twenty routines in a row that report on monitor and screen settings, and the relevant information is only calculated once.

When the CheckForMonitorChanges routine detects that monitor settings have changed, it will report a doChgMonitorSettings event to your application. If your application has a tool bar, then it may be moved or resized automatically in order to ensure that it stays under the menu bar (which the user may move to another monitor), and that it runs across the entire width of the main monitor (whose dimensions may be changed by the user). In these cases, Tools Plus will report a doMoveWindow or doGrowWindow for your tool bar window.

Tools Plus checks for monitor changes just before it reports a doPreRefresh event to your application, and when the user resumes your suspended application. The only time that your application won't be immediately notified of a change in monitor settings, is if it has no open windows and the user changes monitor setting without suspending your application, as is possible with a Control Strip.

Changing Screen Settings

Tools Plus automatically recognizes when the user changes monitor settings via the "Monitors" Control Panel (in any version of the System). This includes [1] changing the orientation of multiple monitors in the "Monitors" Control Panel, [2] changing a monitor between color, gray-scale, or black and white, and [3] changing the number of colors or shades of gray that the monitor will display. Internally, a call to NumberOfScreens detects the change and recalculates the logical screen table. A doRefresh event is then generated, informing your application that all windows need to be redrawn incorporating the new settings.

Note: When your application is running in your development environment, it may not detect the changes in screen settings when the "Monitors" Control Panel is used. This is a limitation of your development environment and not Tools Plus.

HasColorQuickDraw

Determine if the Macintosh running your application has Color QuickDraw.

C
```
pascal Boolean HasColorQuickDraw (void);
```

Pascal
```
function HasColorQuickDraw: BOOLEAN;
```

HasColorQuickDraw informs your application if the Macintosh on which it is running has Color QuickDraw, and Tools Plus makes use of it. If your application has elected not to take advantage of Color QuickDraw (as specified when calling the InitToolsPlus routine at initialization), or if Color QuickDraw is not available, HasColorQuickDraw returns *false*.

Color drawing routines can only be used if the Mac has Color QuickDraw, so your application should not call any of the Macintosh toolbox's color routines unless Color QuickDraw is present. Tools Plus takes care of handling its own drawing routines.

NumberOfScreens

Determine the number of logical screens available on the Macintosh running your application.

C
```
pascal short NumberOfScreens (void);
```

Pascal
```
function NumberOfScreens: INTEGER;
```

NumberOfScreens reports the number of logical screens (not physical monitors) that use unique color settings. Internally, this routine maintains a logical screen table that has one entry for each monitor with unique color or gray-scale settings. If three monitors are available, two having 256 colors and the third with 16 shades of gray, the logical screen table will contain two entries: the first being comprised of the combined region of the two 256 color monitors, and the second being the gray-scale monitor.

Use NumberOfScreens to determine the number of times a window's color-dependent contents have to be drawn (once for each logical screen).

NumberOfScreens returns a value of 1 if the Macintosh running your application doesn't have Color QuickDraw, if color drawing has been disabled by the InitToolsPlus routine, or if the current grafPort is an off-screen bitmap or printing grafPort.

NumberOfScreens is optimized to rebuild the internal tables only if the user changes monitor settings.

BeginUpdateScreen

Begin updating the portion of the current window that is on the specified logical screen.

C
```
pascal void BeginUpdateScreen (short TheScreen);
```

Pascal
```
procedure BeginUpdateScreen (TheScreen: INTEGER);
```

TheScreen specifies the logical screen number on which the drawing will occur. This number must be between 1 and the value returned by the NumberOfScreens routine.

Call BeginUpdateScreen just before you begin drawing the current window's color-dependent contents. BeginUpdateScreen temporarily stores a copy of the current window's visible region (visRgn) and replaces it with an intersection of the window's visible region and the specified logical screen. This limits drawing to the area of the window that is visible only on the specified logical screen.

Subsequent calls to the ScreenDepth and ScreenHasColors routines will refer to the logical screen specified by theScreen.

Each call to BeginUpdateScreen must be balanced by a call to EndUpdateScreen. Also, BeginUpdateScreen and EndUpdateScreen cannot be nested (that is, you must call EndUpdateScreen before the next call to BeginUpdateScreen).

If the Macintosh running your application doesn't have Color QuickDraw, or if color drawing has been disabled by the InitToolsPlus routine, or if theScreen is not a valid logical screen number, BeginUpdateScreen does nothing.

EndUpdateScreen

End updating the portion of the current window that is on the specified logical screen.

C
```
pascal void EndUpdateScreen (void);
```

Pascal
```
procedure EndUpdateScreen;
```

Call EndUpdateScreen after you have completed drawing a window's color-dependent contents. It restores the window's visible region (visRgn) to its original value, that being prior to being altered by BeginUpdateScreen.

Each call to BeginUpdateScreen must be balanced by a call to EndUpdateScreen. Also, BeginUpdateScreen and EndUpdateScreen cannot be nested (that is, you must call EndUpdateScreen before the next call to BeginUpdateScreen).

If BeginUpdateScreen did not successfully modify the current window's visible region, EndUpdateScreen does nothing.

ScreenDepth

Determine the number of colors (or shades of gray) available on the current logical screen.

C
```
pascal short ScreenDepth (void);
```

Pascal
```
function ScreenDepth: INTEGER;
```

The ScreenDepth routine reports the current logical screen's depth (in bits) as set by the "Monitors" control panel. Its value determines the number of colors (or shades of gray) that are seen on the logical screen. ScreenDepth is useful for optimizing programs to take advantage of whatever color capability a monitor has. The table below shows the number of colors that are available per ScreenDepth value.

ScreenDepth Value	Colors/Grays Available
1	black & white
2	4
4	16
8	256
16	thousands
24	millions
32	millions+

ScreenDepth returns a valid value only if it is situated between calls to BeginUpdateScreen and EndUpdateScreen, since BeginUpdateScreen sets the current logical screen number. If ScreenDepth is called outside a BeginUpdateScreen / EndUpdateScreen structure, it reports on the main screen (the one containing the menu bar).

If the Macintosh running your application doesn't have Color QuickDraw, or if color drawing has been disabled by the InitToolsPlus routine, or if BeginUpdateScreen did not successfully modify the current window's visible region, ScreenDepth returns a value of 1.

ScreenHasColors

Determine if a screen is set to draw in color (not gray-scale or black & white).

C
```
pascal Boolean ScreenHasColors (void);
```

Pascal
```
function ScreenHasColors: BOOLEAN;
```

ScreenHasColors returns *true* if the current logical screen has been set by the “Monitors” control panel to display in color, and if Tools Plus has been initialized to make use of color via the InitToolsPlus routine. Use ScreenHasColors if you want to determine if a multi-bit screen is color or gray-scale.

ScreenHasColors returns a valid value only if it is situated between calls to BeginUpdateScreen and EndUpdateScreen, since BeginUpdateScreen sets the current logical screen number. If ScreenHasColors is called outside a BeginUpdateScreen / EndUpdateScreen structure, it reports on the main screen (the one containing the menu bar).

If the Macintosh running your application doesn’t have Color QuickDraw, or if color drawing has been disabled by the InitToolsPlus routine, or if BeginUpdateScreen did not successfully modify the current window’s visible region, ScreenHasColors returns *false*.

CheckForMonitorChanges

Recalculate settings for physical monitors and logical screens.

C
```
pascal void CheckForMonitorChanges (void);
```

Pascal
```
procedure CheckForMonitorChanges;
```

CheckForMonitorChanges recalculates the settings for all monitors to determine logical screens, and to have this information available for subsequent calls by NumberOfScreens, ScreenDepth, ScreenHasColors, NumberOfMonitors, MonitorDepth, MonitorHasColors, MonitorGDevice, and MainMonitorNumber routines.

If monitor changes are detected, they are reported to your application as a doChgMonitorSettings event. In this case, Tools Plus automatically moves and/or resizes your toolbar to make sure it stays below the menu bar and runs across the width of the main monitor.

NumberOfMonitors

Determine the number of physical monitors available on the Macintosh running your application.

C
```
pascal short NumberOfMonitors (void);
```

Pascal
```
function NumberOfMonitors: INTEGER;
```

NumberOfMonitors reports the number of physical monitors that are running on your Macintosh. It returns a value of 1 if the Macintosh running your application doesn’t have Color QuickDraw. This routine is optimized to rebuild the internal monitor and logical screen tables only if the user changes monitor settings.

MonitorDepth

Determine the number of colors (or shades of gray) available on a monitor.

C
```
pascal short MonitorDepth (short MonitorNumber);
```

Pascal
```
function MonitorDepth (MonitorNumber: INTEGER): INTEGER;
```

MonitorNumber is the monitor's ID as displayed in the Monitors Control Panel or equivalent. It identifies a target monitor. A monitor's ID is set when the Macintosh starts up, and does not change.

The MonitorDepth routine reports the target monitor's depth (in bits) as set by the Monitors Control Panel or equivalent. Its value determines the number of colors (or shades of gray) that are seen on the monitor. MonitorDepth is useful for optimizing programs to take advantage of whatever color capability a monitor has. The table below shows the number of colors that are available per MonitorDepth value.

MonitorDepth Value	Colors/Grays Available
1	black & white
2	4
4	16
8	256
16	thousands
24	millions
32	millions+

If the Macintosh running your application doesn't have Color QuickDraw, or if the specified *MonitorNumber* is not with the range of 1 through *NumberOfMonitors*, then MonitorDepth returns a value of 1.

MonitorHasColors

Determine if a monitor is set to draw in color (not gray-scale or black & white).

C
```
pascal Boolean MonitorHasColors (short MonitorNumber);
```

Pascal
```
function MonitorHasColors (MonitorNumber: INTEGER): BOOLEAN;
```

MonitorNumber is the monitor's ID as displayed in the Monitors Control Panel or equivalent. It identifies a target monitor. A monitor's ID is set when the Macintosh starts up, and does not change.

MonitorHasColors returns *true* if the target monitor has been set to display in color. Use MonitorHasColors if you want to determine if a multi-bit monitor is color or gray-scale.

If the Macintosh running your application doesn't have Color QuickDraw, or if the specified *MonitorNumber* is not with the range of 1 through *NumberOfMonitors*, then MonitorHasColors returns a value of *false*.

MonitorGDevice

Get a handle to a monitor's Graphics Device.

C
```
pascal GDHandle MonitorGDevice (short MonitorNumber);
```

Pascal
```
function MonitorGDevice (MonitorNumber: INTEGER): GDHandle;
```

MonitorNumber is the monitor's ID as displayed in the Monitors Control Panel or equivalent. It identifies a target monitor. A monitor's ID is set when the Macintosh starts up, and does not change.

MonitorGDevice returns a handle to the monitor's Graphics Device for developers who need more sophisticated functionality. If the Macintosh running your application doesn't have Color QuickDraw, or if the specified *MonitorNumber* is not with the range of 1 through *NumberOfMonitors*, then MonitorGDevice returns a value of *nil*.

MainMonitorNumber

Determine the main monitor number (the one with the menu bar).

C
```
pascal short MainMonitorNumber (void);
```

Pascal
```
function MainMonitorNumber: INTEGER;
```

MainMonitorNumber reports the main monitor number which contains the menu bar (which may be hidden by your application). It returns a value of 1 if the Macintosh running your application doesn't have Color QuickDraw. This routine is optimized to rebuild the internal monitor and logical screen tables only if the user changes monitor settings.

RectIsVisible

Determine if a rectangle falls within a window's (or grafPort's) visible region.

C
```
pascal Boolean RectIsVisible (const Rect *TheRect);
```

Pascal
```
function RectIsVisible (TheRect: RECT): BOOLEAN;
```

If your application does a lot of drawing in a window, you may be able to speed up the refreshing process by checking if each object *needs* to be redrawn. On an object-by-object basis, use RectIsVisible (or RgnIsVisible) to determine if the object falls within the window's visible region (visRgn). If it does not, your application can save time by not redrawing that object. This routine works on the current grafPort which can be an off-screen bitmap or printing grafPort.

TheRect specifies a rectangle in the current window's local co-ordinates.

RectIsVisible returns *true* if the specified rectangle lies within (in whole or on part) the current window's visible region (visRgn). Otherwise, it returns *false*. This routine works equally well inside or outside a BeginUpdate/EndUpdate structure, or a BeginUpdateScreen/EndUpdateScreen structure. The following example shows how you can use RectIsVisible:

```
for theScreen := 1 to NumberOfScreens do       {Repeat drawing for each monitor in which the    }
  begin                                        { window appears.                                }
    BeginUpdateScreen(theScreen);              {Drawing area reduced to the specified screen    }

    if RectIsVisible(myObjRect) then           {If the rect is visible on the specified screen… }
      {insert your color-dependent drawing code here}

    EndUpdateScreen;                           {End the drawing for this monitor, and restore   }
  end;                                         { the window's visible (drawing) region.         }

  if RectIsVisible(myObjRect) then             {If the rect needs refreshing…                   }
    {insert your color-independent drawing code here}
```

Note carefully that a window's visible region has a different meaning depending on whether BeginUpdate and BeginUpdateScreen are used or not, and therefore, RectIsVisible will return different results. The table below explains this.

Where is RectIsVisible called?		Window's visRgn Encloses…	RectIsVisible means…
outside of BeginUpdate EndUpdate structure	outside of BeginUpdateScreen EndUpdateScreen structure	The current window's visible area (i.e., not off the monitor or obscured by other windows, palettes, menu bar, or tool bar).	Can you see any part of the area enclosed by the rectangle?
outside of BeginUpdate EndUpdate structure	inside **BeginUpdateScreen EndUpdateScreen** structure	The current window's visible area that is part of the logical screen number specified in BeginUpdateScreen.	Can you see any part of the area enclosed by the rectangle within the logical screen number specified in BeginUpdateScreen?
inside **BeginUpdate EndUpdate** structure	outside of BeginUpdateScreen EndUpdateScreen structure	The current window's area that needs refreshing (i.e., it has been newly exposed and objects inside this area must be redrawn)	Does any area enclosed by the rectangle need to be refreshed?
inside **BeginUpdate EndUpdate** structure	inside **BeginUpdateScreen EndUpdateScreen** structure	The current window's area that needs refreshing *and* is part of the logical screen number specified in BeginUpdateScreen.	Does any area enclosed by the rectangle *and* which is located in the logical screen specified by BeginUpdateScreen need to be refreshed?

Also see: RgnIsVisible.

RgnIsVisible

Determine if a specific region falls within a window's (or grafPort's) visible region.

C
```
pascal Boolean RgnIsVisible (RgnHandle TheRgn);
```

Pascal
```
function RgnIsVisible (TheRgn: RGNHANDLE): BOOLEAN;
```

RgnIsVisible is identical to the RectIsVisible routine, except that it accepts a region handle in place of a rectangle.

GetFrontRGB

Get a window's foreground color.

C
```
pascal void GetFrontRGB (RGBColor *Color);
```

Pascal
```
procedure GetFrontRGB (var Color: RGBColor);
```

Color is the current port's foreground color. If Color QuickDraw is not available on the Macintosh running your application, or if Color QuickDraw is not used (as specified when using the InitToolsPlus routine), Color returns with a value set to black (red, green and blue components all set to 0).

GetFrontRGB is functionally equivalent to the toolbox's GetForeColor routine, except that it also works on Macintoshes without Color QuickDraw. This means you can program assuming that Color QuickDraw is available, and your application will still run on a Macintosh without Color QuickDraw.

GetBackRGB

Get a window's background color.

C
```
pascal void GetBackRGB (RGBColor *Color);
```

Pascal
```
procedure GetBackRGB (var Color: RGBColor);
```

Color is the current port's background color. If Color QuickDraw is not available on the Macintosh running your application, or if Color QuickDraw is not used (as specified when using the InitToolsPlus routine), Color returns with a value set to white (red, green and blue components all set to 65535).

GetBackRGB is functionally equivalent to the toolbox's GetBackColor routine, except that it also works on Macintoshes without Color QuickDraw. This means you can program assuming that Color QuickDraw is available, and your application will still run on a Macintosh without Color QuickDraw.

SetFrontRGB

Set a window's foreground color.

C
```
pascal void SetFrontRGB (const RGBColor *Color);
```

Pascal
```
procedure SetFrontRGB (Color: RGBColor);
```

Color is the current port's new foreground color. Color QuickDraw remaps the specified color to the nearest match it can find for the current port. If Color QuickDraw is not available on the Macintosh running your application, or if Color QuickDraw is not used (as specified when using the InitToolsPlus routine), SetFrontRGB does nothing.

SetFrontRGB is functionally equivalent to the toolbox's RGBForeColor routine, except that it also works on Macintoshes without Color QuickDraw. This means you can program assuming that Color QuickDraw is available, and your application will still run on a Macintosh without Color QuickDraw.

SetBackRGB

Set a window's background color.

C
```
pascal void SetBackRGB (const RGBColor *Color);
```

Pascal
```
procedure SetBackRGB (Color: RGBColor);
```

Color is the current port's new background color. Color QuickDraw remaps the specified color to the nearest match it can find for the current port. If Color QuickDraw is not available on the Macintosh running your application, or if Color QuickDraw is not used (as specified when using the InitToolsPlus routine), SetBackRGB does nothing.

SetBackRGB is functionally equivalent to the toolbox's RGBBackColor routine, except that it also works on Macintoshes without Color QuickDraw. This means you can program assuming that Color QuickDraw is available, and your application will still run on a Macintosh without Color QuickDraw.

SetRGB

Store a color's components in an RGB Color record.

C
```
pascal void SetRGB (RGBColor *Color, long Red, long Green, long Blue);
```

Pascal
```
procedure SetRGB (var Color: RGBColor; Red, Green, Blue: LONGINT);
```

SetRGB simply copies the R, G, and B components (supplied by your application) into a color record. It is functionally equivalent to the following code:

```
Color.red := Red;
Color.green := Green;
Color.blue := Blue;
```

Using this routine generates about 8 bytes more object code than the equivalent procedural instructions, but it makes your code more readable and it saves Pascal programmers from having to typecast the color components to integers.

Color is an RGB color record being populated by the individually specified color components.

Red is the RGB value of the color's red component (0 to 65535).

Green is the RGB value of the color's green component (0 to 65535).

Blue is the RGB value of the color's blue component (0 to 65535).

RedrawRect

Erase a rectangular area using the window's backdrop color, then force its contents to be redrawn.

C
```
pascal void RedrawRect (const Rect *TheRect);
```

Pascal
```
procedure RedrawRect (TheRect: RECT);
```

RedrawRect is the perfect way to erase an object, especially if that object overlaps others. The specified rectangle is filled with the current window's backdrop color, then it is invalidated. This causes a doPreRefresh and doRefresh event to be generated, which in turn makes your application refresh objects within the rectangle. Tools Plus objects inside the rectangle are automatically refreshed.

TheRect specifies the target rectangle in the current window's local co-ordinates.

Also see: RedrawRgn.

RedrawRgn

Erase a region using the window's backdrop color, then force its contents to be redrawn.

C
```
pascal void RedrawRgn (RgnHandle TheRgn);
```

Pascal
```
RedrawRgn (TheRgn: RgnHandle)
```

RedrawRgn is identical to RedrawRect, except that a region handle is passed as a parameter to specify a region.

Also see: RedrawRect.

GetDimColor

Calculate a disabled color.

C
```
pascal Boolean GetDimColor (const RGBColor *Front, const RGBColor *Back,
              RGBColor *DimColor);
```

Pascal
```
function GetDimColor (Front: RGBColor; Back: RGBColor;
              var DimColor: RGBColor ): boolean;
```

GetDimColor calculates an object's color when it is disabled. If you use GetDimColor inside a BeginUpdateScreen/ EndUpdateScreen block, the calculation is performed on the logical screen specified by BeginUpdateScreen. If you use it outside of the block the calculation is performed on the main monitor.

Front is the object's foreground RGB color when it is enabled.

Back is the background RGB color on which the object is placed.

DimColor is the calculated color of the object when it is disabled.

The routine's value returns as *true* if a disabled color is available. If a disabled color is not available, *DimColor* if set to the same value as *Front* and the routine returns with a value of *false.* Your application must dither the object if a disabled color is not available. The following example shows how to use this routine:

```
for theScreen := 1 to NumberOfScreens do   {Cycle through each logical screen                  }
  begin                                    {                                                   }
    BeginUpdateScreen(theScreen);          {Restrict drawing to the specified logical screen   }
    PenColorNormal;                        {Set black on white for monochrome monitor.  PatCopy }
                                           {  mode makes pen draw using solid foreground color. }
    SetFrontRGB(ObjectColor);              {If Color QuickDraw is used, set foreground and      }
    SetBackRGB(ObjBackColor);              {  background colors.  Tools Plus ignores these if   }
                                           {  Color QuickDraw is not available or is not used.  }
                                           {Calculate the disabled color. Does the object need  }
                                           {  to be dithered?…                                  }
    dithered := not GetDimColor(ObjectColor, ObjBackColor, DimTextColor);  {                    }
    if dithered then                       {If the object must be dithered…                     }
      PenPat(Gray);                        {  switch to 50% gray pattern to dither the object.  }
    SetFrontRGB(DimTextColor);             {Set dim color (may be same as ObjectColor)          }
    {Draw the object here. The gray pattern makes all pen drawing use a dithered combination of }
    {  the foreground color and background color.  The following code is used to dither an     }
    {  object _after_ it is drawn.  This is needed in situations where it is not possible to    }
    {  dither the object as it is being drawn, such as text before System 7's grayishTextOr text}
    {  transfer mode was available.                                                             }
    if dithered then                       {If the object needs to be dithered…                 }
      begin                                {                                                    }
        PenPat(Gray);                      {Use 50% gray pattern to dither the object           }
        PenMode(patBic);                   {PatBic paints using the background wherever there is}
                                           {  a black pixel in the gray pattern.                }
        PaintRgn(ObjectRgn);               {Dither the object's region                          }
      end;                                 {                                                    }
    EndUpdateScreen;                       {Remove restricted drawing                           }
  end;                                     {                                                    }
```

PenColorNormal

Reset the current window's pen to the default values.

C
```
pascal void PenColorNormal (void);
```

Pascal
```
procedure PenColorNormal;
```

PenColorNormal is a color equivalent of QuickDraw's PenNormal routine in that it resets the pen to its initial state (1x1 size, patCopy mode, black pattern). Additionally, if the Macintosh running your application has Color QuickDraw, and if color drawing has been enabled by the InitToolsPlus routine, the foreground color is set to black and the background color is set to white.

You can use the PenColorNormal routine in place of PenNormal throughout your application, regardless if Color QuickDraw is available or not.

UseHiliteColor

Set the current window's foreground color to the system's highlight color, and the background color to white.

C
```
pascal void UseHiliteColor (void);
```

Pascal
```
procedure UseHiliteColor;
```

When your application needs to draw using the system's highlight color, the UseHiliteColor routine calls PenColorNormal (which resets the pen to its initial state of 1x1 size, patCopy mode, black pattern), then it sets the current window's foreground color to the system's highlight color, and the background color to white. All subsequent drawing will be done using the highlight color.

This routine works perfectly regardless of your monitor's settings, the presence of multiple monitors, or if Color QuickDraw is available or not. QuickDraw (or Color QuickDraw if it's available) maps the highlight color to the closest approximation available on the target monitor.

You can use PenColorNormal to restore the window's foreground and background colors to the default black and white.

Also see: UseHiliteText, HiliteRect and HiliteRgn.

UseHiliteText

Set the current window's text drawing mode for drawing on the highlight color.

C
```
pascal void UseHiliteText (void);
```

Pascal
```
procedure UseHiliteText;
```

Use the UseHiliteText routine just before your application needs to draw text on the system's highlight color. Typically, your application would use both UseHiliteColor and UseHiliteText within the same piece of code.

This routine works perfectly regardless of your monitor's settings, the presence of multiple monitors, or if Color QuickDraw is available or not. If UseHiliteText is used outside a BeginUpdateScreen / EndUpdateScreen structure, it sets the text drawing mode to be appropriate for the highlight color used on the main monitor. When used inside a BeginUpdateScreen / EndUpdateScreen structure, it sets the text drawing mode to be appropriate for the highlight color used on the logical screen specified by BeginUpdateScreen.

```
UseHiliteColor;                                    {Draw using the highlight color               }
{draw your highlights here (foreground color is the highlight color) }
```

```
  for theScreen := 1 to NumberOfScreens do       {Repeat drawing for each monitor in which the    }
    begin                                        { window appears.                                }
      BeginUpdateScreen(theScreen);              {Drawing area reduced to the specified screen    }
      UseHiliteText;                             {Set text drawing mode for drawing on the        }
                                                 { highlight color as it appears on this screen.  }
      MoveTo(20, 200);                           {Position the pen for drawing the text           }
      DrawString('Highlighted Text');            {Draw text on the highlighted color              }
      TextMode(srcOr);                           {Restore default text drawing mode               }
      EndUpdateScreen;                           {End the drawing for this monitor, and restore   }
    end;                                         { the window's visible (drawing) region.         }
```

UseHiliteText sets the text drawing mode and the window's foreground colors such that black text is drawn on the window's highlight color. When the window's highlight color is very dark or black, UseHiliteText sets the text drawing mode to srcBic resulting in white characters upon the highlight color.

After you finish drawing text, you can restore the text drawing mode to the default setting using TextMode(srcOr).

Also see: UseHiliteColor, HiliteRect and HiliteRgn.

HiliteRect

Highlight or unhighlight a rectangle on the current window, and prepare to draw text in it.

C
```
pascal void HiliteRect (const Rect *TheRect, Boolean Hilite);
```

Pascal
```
procedure HiliteRect (TheRect: Rect; Hilite: boolean);
```

HiliteRect is an intelligent combination of the UseHiliteColor and UseHiliteText routines. It is used to highlight or unhighlight a rectangular area (likely a cell in a spreadsheet or a line in a custom list), and prepare for drawing text in that area.

TheRect is the affected area specified as a rectangle.

Hilite specifies if the rectangle is to be highlighted or unhighlighted. You can use the constants *on* or *off* for this purpose.

When a rectangle is highlighted (*Hilite* = *true*), the rectangle is filled with the system's highlight color. QuickDraw (or Color QuickDraw if it is available) maps the highlight color to the closest approximation available on the target monitor. PenColorNormal is called to restore the window's pen settings to their defaults (1x1 size, patCopy mode, black pattern, black foreground and white background). Then text drawing mode is set to srcOr for drawing black characters upon the highlight color, or srcBic if the highlight color is black (or it translates to black on the target monitor) to produce white characters on the black highlight.

When the rectangle is unhighlighted (*Hilite* = *false*), the rectangle is filled with white. PenColorNormal is called to restore the window's pen settings to their defaults (1x1 size, patCopy mode, black pattern, black foreground and white background). Then text drawing mode is set to srcOr mode for drawing black characters regardless of the background.

After using HiliteRect, you can draw the text in the affected area using DrawString (or an equivalent). You may also draw icons or other objects over the highlighted color. You can restore the text drawing mode to its default setting using TextMode(srcOr) after you have finished drawing in the rectangle.

This routine works perfectly regardless of your monitor's settings, the presence of multiple monitors, or if Color QuickDraw is available or not. If HiliteRect is used outside a BeginUpdateScreen / EndUpdateScreen structure, it sets the text drawing mode to be appropriate for the highlight color used on the main monitor. When used inside a BeginUpdateScreen / EndUpdateScreen structure (the recommended way), it sets the text drawing mode to be appropriate for the highlight color used on the logical screen specified by BeginUpdateScreen.

```
  for theScreen := 1 to NumberOfScreens do        {Repeat drawing for each monitor in which the    }
    begin                                         { window appears.                                }
      BeginUpdateScreen(theScreen);               {Drawing area reduced to the specified screen    }
      HiliteRect(myRect, true);                   {Highlight the rectangle, and set text drawing   }
                                                  { mode for drawing on the highlight color as it  }
                                                  { appears on this screen.                        }
      MoveTo(20, 200);                            {Position the pen for drawing the text           }
      DrawString('Highlighted Text');             {Draw text on the highlighted color              }
      TextMode(srcOr);                            {Restore default text drawing mode               }
      EndUpdateScreen;                            {End the drawing for this monitor, and restore   }
    end;                                          { the window's visible (drawing) region.         }
```

Also see: UseHiliteColor, UseHiliteText and HiliteRgn.

Programming Tips:

1 For the best results, highlight objects only when the window displaying them is active. You can use the WindowIsActive routine in conjunction with the *Hilite* parameter to accomplish this, such as
   HiliteRect (myRect, HiliteFlag and WindowIsActive(CurrentWindowNumber));

2. If you are following tip 1, make sure your application responds to a doActivate event (window was activated) and doDeactivate event (window was deactivated) by highlighting or unhighlighting selected items.

HiliteRgn

Highlight or unhighlight a region on the current window, and prepare to draw text in it.

C
```
pascal void HiliteRgn (RgnHandle TheRgn, Boolean Hilite);
```

Pascal
```
procedure HiliteRgn (TheRgn: RgnHandle; Hilite: boolean);
```

HiliteRgn is an intelligent combination of the UseHiliteColor and UseHiliteText routines. It is used to highlight or unhighlight a region and prepare for drawing text in that area. This routine is a functional equivalent to HiliteRect, except that it accepts a region instead of a rectangle as a parameter.

TheRgn is the affected area specified as a region.

Hilite specifies if the region is to be highlighted or unhighlighted. You can use the constants *on* or *off* for this purpose.

Also see: UseHiliteColor, UseHiliteText and HiliteRect.

GetColorPenState

Get the current window's pen settings.

C
```
pascal void GetColorPenState (ColorPenState *ThePenState);
```

Pascal
```
procedure GetColorPenState (var ThePenState: ColorPenState);
```

GetColorPenState is a color equivalent of QuickDraw's GetPenState routine in that it obtains a copy of the current window's pen location, size, pattern, and display mode in *ThePenState*. Additionally, if the Macintosh running your application has Color QuickDraw, and if color drawing has been enabled by the InitToolsPlus routine, the foreground and background color are also stored in *ThePenState*. These settings can be restored later with SetColorPenState.

This routine is useful when calling subroutines that operate in the current window but must change the graphics pen. Each such routine can save the pen's state, perform whatever tasks it needs to do, then restore the original pen state immediately before returning. The ColorPenState record is defined as such:

```
ColorPenState = record          {Color equivalent for a 'PenState' record          }
    PenState: PenState;         {Standard QuickDraw pen state                      }
    ForegroundColor: RGBColor;  {Window's foreground color                         }
    BackgroundColor: RGBColor;  {Window's background color                         }
end;
```

You can use the GetColorPenState routine in place of GetPenState throughout your application, regardless if Color QuickDraw is available, or if it is used (as specified when calling InitToolsPlus).

SetColorPenState

Get the current window's pen settings.

C
```
pascal void SetColorPenState (const ColorPenState *ThePenState);
```

Pascal
```
procedure SetColorPenState (ThePenState: ColorPenState);
```

SetColorPenState is a color equivalent of QuickDraw's SetPenState routine in that it sets the current window's pen location, size, pattern, and display mode according to the values stored in *ThePenState*. Additionally, if the Macintosh running your application has Color QuickDraw, and if color drawing has been enabled by the InitToolsPlus routine, the foreground and background colors are also set.

This is usually done at the end of a routine that has altered the pen parameters and wants to restore them to their original state that existed at the beginning of the routine. (See GetColorPenState.)

You can use the SetColorPenState routine in place of SetPenState throughout your application, regardless if Color QuickDraw is available, or if it is used (as specified when calling InitToolsPlus).

18 User Notification

This chapter describes Tools Plus's implementation of the Macintosh toolbox's *Notification Manager*. The Notification Manager is used to tell the user that there is something happening they need, or want to be aware of in an *inactive* application. This feature is available in System 6 (when running MultiFinder) and System 7 or higher. An example of the Notification Manager at work is when Print Monitor tells your application it needs a sheet of paper to be inserted for a manual page feed.

Tools Plus does more than just simplify access to the Notification Manager. User notification is automatic whenever your application displays a Dynamic Alert box. If your application is inactive when it uses the AlertBox routine, Tools Plus notifies the user by displaying a notification dialog such as the one shown below.

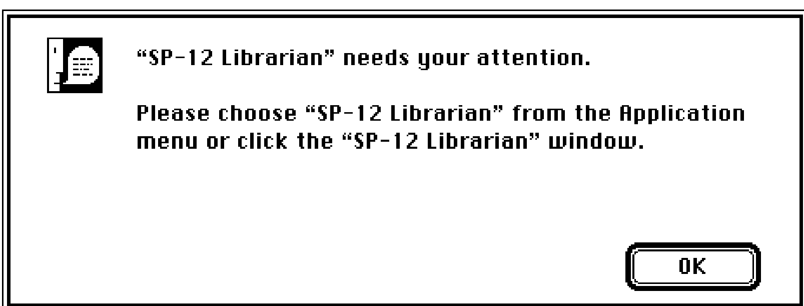
Your application can customize how the user is notified by using the SetNotification routine. The user can be notified by any of the following means (including and combination).

- Display a flashing icon in the Application menu (System 7 or higher) or Apple menu (System 6)
- Play a sound (or the system's default error sound)
- Display an alert with a message in it that requires the user to click the OK button before continuing.

To comply with Macintosh user interface guidelines, you should think of notifications as three levels:

1. Display a flashing icon
2. Display a flashing icon and play a sound
3. Display a flashing icon, optionally play a sound, and display a message

Ideally, your application should let the user decide if and how they want to be notified, as demonstrated by the Print Monitor's notification options.



Default notification dialog displayed while your application is inactive and needing attention

Notifying the User

Tools Plus automatically accesses the Macintosh toolbox's Notification Manager if your application is inactive and it uses the AlertBox routine. A default notification message is displayed, as shown on the previous page.

Your application can use the SetNotification routine to specify a small icon ('SICN' resource) that will flash in the menu bar during notification. SetNotification also specifies the sound that is used (if any), and the message that is displayed in a notification alert (if any) when the user is notified.

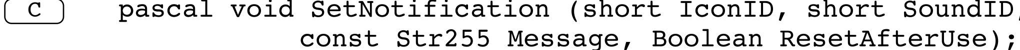
If your application wants to notify the user without using a Dynamic Alert, it can do so with the PostNotification routine. If your application is inactive, PostNotification notifies the user as per the settings specified by SetNotification.

When your application is activated, the notification is cleared.

SetNotification

Define the settings for notifying the user.

C



```
pascal void SetNotification (short IconID, short SoundID,
            const Str255 Message, Boolean ResetAfterUse);
```

Pascal

```
procedure SetNotification (IconID, SoundID: INTEGER; Message: STRING;
            ResetAfterUse: BOOLEAN);
```

IconID is the resource ID for an ‘SICN’ small icon that is flashed in the menu bar during notification. Tools Plus loads this resource into memory and locks it while posting the notification. If you don’t want to display an icon, use the NoIcon constant.

SoundID is the resource ID for an ‘snd ’ sound that is played when the user if first notified. Tools Plus loads this resource into memory and locks it while posting the notification. You can use the nmSysBeep constant if you want to play the default system error sound, or nmSilentNote if you don’t want to play a sound at notification.

Message is the string that is displayed in the notification alert. You can use the nmDefaultMsg constant if you want to use the default notification message, or the nmNoMsg constant if you don’t want a notification alert displayed. The message can contain two string variables that are replaced by Tools Plus:

^0 is replaced by your application’s name
^1 is replaced by the symbol in System 6, and the word “Application” under System 7 or higher

These two variables can be combined to produce tailored messages that adapt automatically to System 6 and System 7 or higher. Tools Plus’s default message is:

“^0” needs your attention.

Please choose “^0” from the ^1 menu or click the “^0” window.

In all cases, “^0” is replaced with your application’s name. The “^1” is replaced by text that is dependent on the system version on which your application is running. Therefore, the default string is automatically changed to:

System 6: “AppName” needs your attention.

Please choose “AppName” from the menu or click the “AppName” window.

System 7+: “AppName” needs your attention.

Please choose “AppName” from the Application menu or click the “AppName” window.

ResetAfterUse specifies if the notification settings should be reset to their default values after notification is posted. If *ResetAfterUse* is *true*, the settings revert to their default values. If *ResetAfterUse* is *false*, the specified settings remain in effect until they are changed by SetNotification. You can use the nmResetWhenDone and nmKeepSettings constants for this item.

Also see: PostNotification.

```
CONST                                {Notification constants…                 }
               nmSysBeep       = -1;    {NOTIFICATION: Use System Error sound      }
               nmSilentNote    = 0;     {              Silent notification         }
               nmDefaultMsg    = ' ';   {              Use default message         }
               nmNoMsg         = '';    {              Don't display an alert      }
               nmResetWhenDone = true;  {              Reset to defaults after used }
               nmKeepSettings  = false; {              Keep settings after used    }
               NoIcon          =-32768; {ICONS:        No icon used                }
```

Note: If your application is displaying an icon during notification, and/or plays a sound other than the default system error sound, the ‘SICN’ and ‘snd ’ resources are locked in memory by Tools Plus until your application is activated. Therefore, you can flag these resources as “purgeable” to save memory.

PostNotification

Notify the user that your application needs attention.

C
```
pascal Boolean PostNotification (void);
```

Pascal
```
function PostNotification: BOOLEAN;
```

If your application is inactive, PostNotification notifies the user that your application needs attention. If SetNotification has been used to specify notification settings, then those settings are used. Otherwise, Tools Plus's default settings are used.

If your application is inactive when PostNotification is called, the routine returns *true*. If your application is active, or it is running under System 5, or it is running under System 6's Finder (not MultiFinder), PostNotification does nothing and return's *false*.

19 Dynamic Alerts

Dynamic alerts are very similar to the Macintosh's alerts, only they are much better in most situations. They automatically change size and shape to accommodate the text that is displayed in them, and they are always centered on the main monitor. It's like having hundreds of custom alerts available, without having to design any of them!

When a dynamic alert is displayed, it optionally beeps the user and the cursor is changed to the Macintosh arrow. The alert box is automatically sized to accommodate the text specified by your application, and it is centered on the main monitor to be aesthetically pleasing. An icon can optionally be displayed in the top left corner of the alert. Your application also specifies the combination of buttons that appear at the bottom of the alert.

The AlertBox routine automatically processes mouse clicks and typing events, and applies them to the alert. When the user clicks one of the buttons, or types Return or Enter to activate a default button, the alert box closes and control is returned to your application. The AlertBox routine returns the value of the button that was selected by the user.

Multitasking in Dynamic Alerts

When your application calls the AlertBox routine, Tools Plus opens a window, populates it appropriately, and sets up a private window event handler to handle mouse clicks, typing, and refreshing events for the alert. While the dynamic alert is displayed, Tools Plus calls your application's event handler routine(s) to do such things as process doRefresh events for your application's windows, doDeactivate events, and possibly other events.

By default, Tools Plus does not report doNothing events to your application when a dynamic alert is displayed because your application may display a dynamic alert in response to a doNothing event. In such cases, this may cause your application to recursively redisplay the same alert.

If you write your event handler routine in such a way that it senses if a dynamic alert is displayed and thereby prevents the recursive redisplaying of an alert, then you can have Tools Plus report doNothing events to your event handler while a dynamic alert is displayed by using the SetAlertBoxNullEvents routine. Doing this allows your application to continue to carry on with background processing in response to a doNothing event, even while a dynamic alert is displayed. You can use the AlertBoxCount routine to determine how many dynamic alerts are displayed.

Tools Plus allows the simultaneous displaying of up to three (3) dynamic alerts. The following is an example of how it may be possible to have multiple dynamic alerts open simultaneously:

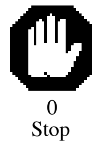
- Your application allows the reporting of doNothing events while a dynamic alert is displayed (see the SetAlertBoxNullEvents routine)
- The user starts searching a database for records. The search runs as a background process in response to doNothing events.
- The user closes one of your application's document windows, to which your application responds by displaying a dynamic alert: “Save changes before closing this document?” (Yes, No, Cancel buttons are available). This is the *first* alert displayed. While the alert is waiting for the user's response, your application continues to receive doNothing events and it continues to search the database.
- The database search finishes without finding any entries, and it displays a *second* dynamic alert: “No records found.” (Ok button is available)

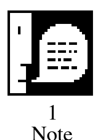
At this time, two dynamic alerts are displayed with the most recent one being active. As soon as the user dismisses the second alert, the first alert is activated and the user can decide if they want to save the document before closing it.

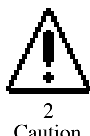
Icons

When Tools Plus draws the icon in the dynamic alert, it does so by accessing an *icon family* and by being sensitive to the settings of the monitor on which the icon is being displayed. This means that it displays the best available icon (cicn, icl8, icl4, ICN#, or ICON resource) for the target monitor. You can define 'cicn' or 'icl8' icons for use with monitors set to 8-bits or higher, 'icl4' icons for monitors set to 4-bits or higher, and 'ICN#' or 'ICON' icons for monitors set to 1-bit or higher. For more details about how icons are displayed, see the DrawIcon routine which is used by AlertBox to draw the icon.

The System file includes 3 icons that are ready for your use. These are the "stop", "note", and "caution" icons as illustrated below. Constants have been defined to let you use these system icons without system compatibility concerns.


0
Stop


1
Note


2
Caution

Text

The text displayed by dynamic alerts is automatically split into multiple lines by using word-wrap if necessary. AlertBox adjusts the box's width to make the length of multiple lines as similar as possible. A Carriage Return (ASCII code $0D) can be used within the text to start a new line. You can use the ReturnKey constant to make your program more readable. This lets you form an alert box with multiple lines of text by using a single line of source code in your program.

A dynamic alert's text is left-aligned unless it has no buttons, in which case the text is centered.

Buttons

Dynamic alerts can display up to three buttons, including an optional default button. The default button is outlined with a border and is automatically selected if the user presses the Return or Enter key. Several common button combinations have been defined for you as constants (samples provided later). Later in this manual, you will be shown how to define your own button combinations.

The user can select a button on a dynamic alert by using a command key equivalent. The Escape key or ⌘-. (Command-period) can be used to select the "Cancel" button or a language-dependent equivalent. All other buttons can be selected by using the command key in conjunction with the first character of the button's title.

Routine's Value

The AlertBox routine returns with a value that indicates the button that was clicked by the user. If you display an alert with no buttons, AlertBox returns with a value of 1 when the alert is clicked by the user. Constants are defined to let your application match the routine's value to a button name more easily. A value of -1 indicates that the alert could not be opened, either due to a severe memory shortage, or more likely, because 3 dynamic alerts are already open concurrently, and your application likely has recursive code that, if allowed, would continue to open dynamic alerts infinitely.

Automatic User Notification

If your application is running under System 6's MultiFinder or System 7 or higher, dynamic alerts automatically make use of the Macintosh toolbox's Notification Manager. The Notification Manager is used to tell the user that there is something happening they need, or want to be aware of in an *inactive* application. An example of the Notification Manager at work is when Print Monitor tells your application it needs a sheet of paper to be inserted for a manual page feed.

If your application is inactive when it uses the AlertBox routine, Tools Plus notifies the user by displaying a notification dialog. Your application can customize the notification by using the SetNotification routine.

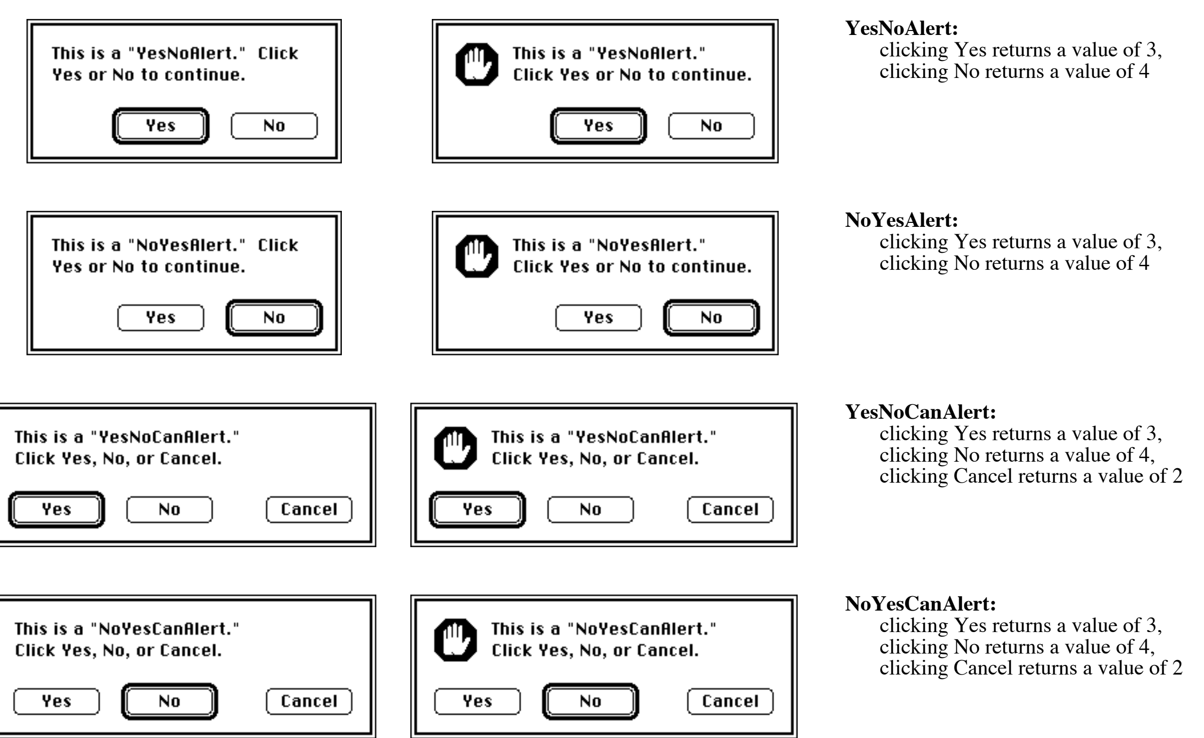
Appearance Manager

There are two ways you can make dynamic alerts Appearance Manager savvy. The easiest way is to initialize Tools Plus (InitToolsPlus routine) with the initAppearanceManagerSavvy option. This automatically replaces all references to classic System 7 style buttons and windows with 3D equivalents made available by the Appearance Manager. Your dynamic alerts will automatically use the correct window, background theme, buttons, and font as specified by the Appearance Manager's current "theme." Alternatively, you can use the SetAlertBoxPrefs routine to manually specify window colors, fonts, and button appearance details.

The Appearance Manager's background is applied to the alert if you leave the background color at its default color, white. If you specify a different background color, the Appearance Manager's background is not applied to the dynamic alert.

Alert Samples

The following are examples of predefined dynamic alert types. In each case, the left column shows a specific type without an icon, and the right column with an icon. Your application is not limited to the alerts shown below. You can define your own custom button combinations, and button names as well.



YesNoAlert:
clicking Yes returns a value of 3,
clicking No returns a value of 4

NoYesAlert:
clicking Yes returns a value of 3,
clicking No returns a value of 4

YesNoCanAlert:
clicking Yes returns a value of 3,
clicking No returns a value of 4,
clicking Cancel returns a value of 2

NoYesCanAlert:
clicking Yes returns a value of 3,
clicking No returns a value of 4,
clicking Cancel returns a value of 2

AlertBox

Display a dynamic alert box.

C
```
pascal short AlertBox (short theIcon, const Str255 AlertText,
            long AlertCode);
```

Pascal
```
function AlertBox (theIcon: INTEGER; AlertText: STRING;
            AlertCode: LONGINT): INTEGER;
```

TheIcon is the icon ID that is displayed in the alert. If the icon ID you specify does not exist, then the noteIcon is displayed in its place. Your application can specify any icon it wants, providing the icon resource exists in either the System file or your application. For more details about how the icon is drawn, see the DrawIcon routine which is used by AlertBox to draw the icon. If your application will run on a Macintosh with an Appearance Manager, use either a 'cicn' icon, or an icon suite that includes an 'icl8' icon, an optional 'icl4' icon, and an 'ICN#' icon with mask. The Appearance Manager checks for an 'cicn' before an icon suite. The Appearance Manager also includes 3D icons for the standard Stop, Note, and Caution icons.

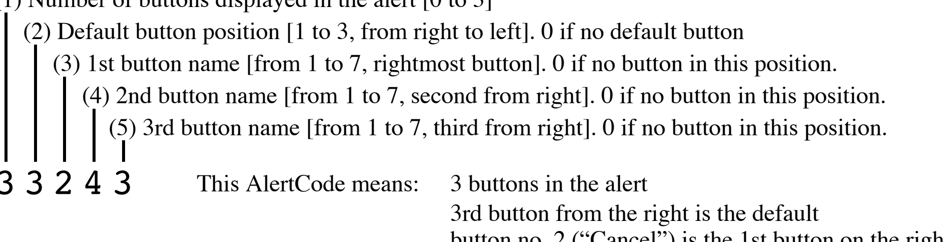
AlertText is the text that is displayed in the dynamic alert. A Carriage Return (ASCII code $0D) can be used in the text to start a new line. You can use the ReturnKey constant to make your program more readable. A dynamic alert's text is left-aligned unless the alert has no buttons, in which case the text is centered.

AlertCode specifies the button layout that appears in the dynamic alert. See "Custom Button Combinations" (below) to define your own button layouts.

The AlertBox routine returns with a value that indicates which button was clicked by the user. If AlertBox displays an alert with no buttons, it returns with a value of 1 when the user clicks in the alert. Constants are defined to let your application match the routine's value to a button name more easily. The AlertBox routine returns a value of -1 if it cannot display the alert for any reason such as a severe memory shortage or the maximum number of dynamic alerts are already open.

Custom Button Combinations

AlertCode lets you specify the button layout that appears on dynamic alerts. This code is a long integer whose value is broken into 5 single-digit numbers, each of which specifies something about the button combination:



Single button alerts have the button centered between the left and right side of the alert box. Two button alerts have their buttons placed side by side in the bottom right corner of the alert. Three button alerts have the first button placed on the bottom right side of the box, and the next two are paired off further to the left.

If you initialize Tools Plus using the initAppearanceManagerSavvy option, and providing you don't specify a background color for dynamic alerts (using the SetAlertBoxPrefs routine), dynamic alerts adopt the Appearance Manager's background theme. In cases when you want to override this behavior and have a plain white dynamic alert, add the alertPlainBackdrop option to the *AlertCode*.

If the value of *AlertCode* is negative, the dynamic alert doesn't beep when displayed. We suggest that you define your own custom alert codes as constants, and use those constants to make your source code more readable.

Advanced Techniques

If your application uses a lot Alerts, or you want to make as efficient use of memory as possible, you can store all the alerts' messages as string resources and write your own routine that accepts a string resource ID from the calling application, then loads the string resource, and hands it off to the AlertBox routine.

If you want to get even more advanced, you can add some extra data bytes to the end of your string resource, representing the icon number and button combination. Your application would call *your* alert routine and provide it with a string resource ID. Your alert routine would then load the string resource, read the data bytes and parse them into the *theIcon* and *AlertCode* parameters for the AlertBox routine. By using this concept, you could process an alert with a call that looks like the following:

```
UserButton := MyAlert(501);
```

The number 501 would be your string resource ID, and UserButton would return the button that was selected by the user. You still have the advantage of using a dynamic alert, and you can enjoy even greater simplicity.

Also see: AlertBox3 and AlertButtonName to rename buttons in dynamic alerts.

```
CONST                                  {Icon IDs for alert icons…                }
              NoIcon        =-32768    {no icon displayed in alert box           }
              stopIcon      = 0;       {stopIcon will automatically access       }
              noteIcon      = 1;       {   ID = 3 in system files version 5 or   }
              cautionIcon   = 2;       {   6.                                    }
                                       {Dynamic alert buttons… }
              OkAltBut      = 1;       {OK                                       }
              CanAltBut     = 2;       {Cancel                                   }
              YesAltBut     = 3;       {Yes                                      }
              NoAltBut      = 4;       {No                                       }
              ContAltBut    = 5;       {Continue                                 }
              SkipAltBut    = 6;       {Skip                                     }
              QuitAltBut    = 7;       {Quit                                     }
                                       {Dynamic alert button combinations…       }
              NoButtonAlert =     0;   {No buttons                               }
              OkAlert       = 11100;   {OK                (OK default)           }
              CanAlert      = 11200;   {Cancel            (Cancel default)       }
              OkCanAlert    = 22210;   {OK + Cancel       (OK default)           }
              CanOkAlert    = 21210;   {OK + Cancel       (Cancel default)       }
              YesNoAlert    = 22430;   {Yes/No            (Yes default)          }
              NoYesAlert    = 21430;   {Yes/No            (No default)           }
              YesNoCanAlert = 33243;   {Yes/No + Cancel   (Yes default)          }
              NoYesCanAlert = 32243;   {Yes/No + Cancel   (No default)           }
              alertPlainBackdrop = 1000000; {Do not fill alert with Appearance Manager's }
                                       {  theme.                                 }
```

Note: (System 5 and System 6's Finder only) It is possible for your application to call AlertBox when none of its windows are active. An example of this occurs as follows: [1] user enters text in an editing field on window "a", [2] user opens a desk accessory, [3] user clicks on another window ("b") belonging to your application, [4] your application determines that the user cannot activate window "b" until the field in window "a" is corrected, so it displays a dynamic alert stating so.
Whenever a dynamic alert is called, it automatically insures that the frontmost window belonging to your application is active before the alert is displayed.

Programming Tips:

1 If you want your application to be in "full color" by displaying colorized stop, note, and caution icons in your alerts, include the demo application's 'icl8', 'icl4' and 'ICON' icons (ID numbers 0, 1 and 2). AlertBox will display the icon that is best for your monitor settings.

AlertBox3

Display a dynamic alert box using temporary button titles.

C
```
pascal short AlertBox3 (short theIcon, const Str255 AlertText,
            long AlertCode, const Str255 But1, const Str255 But2,
            const Str255 But3);
```

Pascal
```
function AlertBox3 (theIcon: INTEGER; AlertText: STRING;
            AlertCode: LONGINT; But1, But2, But3: STRING): INTEGER;
```

AlertBox3 is identical to the AlertBox routine, except that it accepts an additional three strings to temporarily replace the titles of the first three (of a possible nine) buttons. The dynamic alert displays the button titles you specify then it reverts to the original titles after the alert is closed.

But1 through *But3* are strings that represent temporary replacement titles for buttons 1 through 3.

This routine is helpful if your application frequently creates alerts with custom button titles.

Note: Make sure your *AlertCode* references only button numbers 1, 2 and 3 in places where you want to temporarily substitute the standard button titles with the titles you specify in the routine.

AlertButtonName

Change the button title on dynamic alert boxes.

C
```
pascal void AlertButtonName (short Button, const Str255 Title);
```

Pascal
```
procedure AlertButtonName (Button: INTEGER; Title: STRING);
```

Tools Plus provides seven different button titles that may be used in various combinations on dynamic alert boxes. Although button titles and button number constants are provided for only seven buttons (as detailed in AlertBox), a total of nine buttons are available for use.

Button specifies the button number (from 1 to 9) that is affected.

Title specifies the button's title that will appear on all subsequent dynamic alerts. Each button's size changes to accommodate the button's title width, so it is important that you test your alerts to ensure that they look good. The specified title stays in effect until it is explicitly changed by your application. If a null string is provided, the button resumes its default title (the default titles are listed earlier in this chapter).

Your application can rename any or all of the button titles as required, but keep in mind that the constants defined for the AlertCode and buttons will not work correctly because buttons have been renamed.

GetAlertBoxPrefs

Get preferences for dynamic alerts.

C
```
pascal void GetAlertBoxPrefs (TPAlertBoxPrefs *Prefs);
```

Pascal
```
procedure GetAlertBoxPrefs (var Prefs: TPAlertBoxPrefs);
```

The GetAlertBoxPrefs routine retrieves preferences settings for dynamic alerts in a preferences record. You can then change individual items in the record and save the new settings with SetAlertBoxPrefs.

The *Prefs* record contains preferences for dynamic alerts and is defined as such:

C

```
struct TPAlertBoxTextPrefs {      /* Settings for message displayed in the alert:  */
   short Font;                    /*  · Font                                       */
   short Size;                    /*  · Font's size                                */
   short Style;                   /*  · Font's style                               */
   RGBColor Color;                /*  · Font's color                               */
   };
typedef struct TPAlertBoxTextPrefs TPAlertBoxTextPrefs;
struct TPAlertBoxButtonPrefs {    /*Settings for button displayed in the alert:    */
   short ProcID;                  /*  · ProcID                                     */
   short Font;                    /*  · Font                                       */
   short Size;                    /*  · Font's size                                */
   short Style;                   /*  · Font's style                               */
   RGBColor FrameColor;           /*  · Frame's color                              */
   RGBColor BodyColor;            /*  · Body's color                               */
   RGBColor TextColor;            /*  · Text's color                               */
   RGBColor BackColor;            /*  · Background color (ignored by most CDEFs)   */
   };
typedef struct TPAlertBoxButtonPrefs TPAlertBoxButtonPrefs;
struct TPAlertBoxPrefs {          /*Dynamic Alert's preferences                    */
   RGBColor Backdrop;             /* · Window's backdrop color                     */
   TPAlertBoxTextPrefs Text;      /* · Text preferences                            */
   TPAlertBoxButtonPrefs Button1; /* · Prefs for button in 1st position (right)    */
   TPAlertBoxButtonPrefs Button2; /* · Prefs for button in 2nd position            */
   TPAlertBoxButtonPrefs Button3; /* · Prefs for button in 3rd position            */
   };
typedef struct TPAlertBoxPrefs TPAlertBoxPrefs;
```

Pascal

```
TPAlertBoxTextPrefs = record       {Settings for message displayed in the alert:   }
   Font: integer;                  {  · Font                                       }
   Size: integer;                  {  · Font's size                                }
   Style: Style;                   {  · Font's style                               }
   Color: RGBColor;                {  · Font's color                               }
end;
TPAlertBoxButtonPrefs = record     {Settings for button displayed in the alert:    }
   ProcID: integer;                {  · ProcID                                     }
   Font: integer;                  {  · Font                                       }
   Size: integer;                  {  · Font's size                                }
   Style: Style;                   {  · Font's style                               }
   FrameColor: RGBColor;           {  · Frame's color                              }
   BodyColor: RGBColor;            {  · Body's color                               }
   TextColor: RGBColor;            {  · Text's color                               }
   BackColor: RGBColor;            {  · Background color (ignored by most CDEFs)   }
end;
TPAlertBoxPrefs = record           {Dynamic Alert's preferences                    }
   Backdrop: RGBColor;             { · Window's backdrop color                     }
   Text: TPAlertBoxTextPrefs;      { · Text preferences                            }
   Button1: TPAlertBoxButtonPrefs; { · Prefs for button in 1st position (right)    }
   Button2: TPAlertBoxButtonPrefs; { · Prefs for button in 2nd position            }
   Button3: TPAlertBoxButtonPrefs; { · Prefs for button in 3rd position            }
end;
```

SetAlertBoxPrefs

Set preferences for dynamic alerts.

C

```
pascal void SetAlertBoxPrefs (const TPAlertBoxPrefs *Prefs);
```

Pascal

```
procedure SetAlertBoxPrefs (Prefs: TPAlertBoxPrefs);
```

The SetAlertBoxPrefs routine stores preferences settings for dynamic alerts. See the GetAlertBoxPrefs routine to retrieve the preferences before making changes, and to see the TPAlertBoxPrefs record layout.

The *Prefs* record contains preferences for dynamic alerts and is defined as such:

The following example shows how to change preferences for dynamic alerts:

```
GetAlertBoxPrefs(Prefs);                  {Get dynamic alert preferences          }
Prefs.Text.Font := geneva;                {Message will be displayed using Geneva 9pt  }
Prefs.Text.Size := 9;                     {                                       }
Prefs.Button1.ProcID := 32000 + useWFont; {1st (right most) button has procID of 32000 }
                                          {  (for CDEF resource ID = 2000) and it uses }
                                          {  a custom font.                       }
Prefs.Button1.Font := helvetica;          {1st button uses Helvetica 10pt         }
Prefs.Button1.Size := 10;                 {                                       }
```

```
Prefs.Button2 := Prefs.Button1;          {2nd button uses same settings as 1st button  }
SetAlertBoxPrefs(Prefs);                 {Store dynamic alert preferences              }
```

SetAlertBoxNullEvents

Allow/disallow doNothing events while a dynamic alert is open.

C
```
pascal void SetAlertBoxNullEvents (Boolean AllowNulls);
```

Pascal
```
procedure SetAlertBoxNullEvents (AllowNulls: BOOLEAN);
```

The SetAlertBoxNullEvents routine allows or disallows the reporting of doNothing events while a dynamic alert is displayed. By default, doNothing events are not reported to your event handler when a dynamic alert is open. The “Multitasking in Dynamic Alerts” section at the beginning of this chapter details the ramifications of allowing or disallowing doNothing event reporting when a dynamic alert is open.

AllowNulls specifies if doNothing events are reported to your event handler while a dynamic alert is displayed. A setting of *true* allows reporting, while a value of *false* disallows it. You can use the constants *on* and *off* for this purpose.

AlertBoxCount

Determine the number of dynamic alerts that are open.

C
```
pascal short AlertBoxCount (void);
```

Pascal
```
function AlertBoxCount: INTEGER;
```

The *AlertBoxCount* routine returns the number of dynamic alerts that are open. Your application may need to know this if it takes steps to prevent the recursive redisplaying of dynamic alerts, as described early in this chapter in the “Multitasking in Dynamic Alerts” section. Your application can also compare this value to the MaxAlertBoxes constant (which represents the maximum number of dynamic alerts that can be open simultaneously) to determine if it is possible to open more alerts.

```
CONST         MaxAlertBoxes = 3;           {Maximum number of Dynamic Alerts that can be }
                                           {  open concurrently.                         }
```

20 Miscellaneous Routines

This section contains some routines that simplify and enhance the programming experience. Some routines were developed specifically to replace similar parts of the Macintosh's toolbox by providing simpler or more useful counterparts, such as Tools Plus's DrawIcon. Other routines were written to mimic existing Macintosh user interface features to which programmers do not have access, such as the "zoom lines" that are seen when the Finder opens a document, or the standard Macintosh thermometer that indicates an application's progress. And yet other routines are included because they were needed for the internal working of Tools Plus, and we thought they would be useful to you as well.

StrInBox

Draw a string in a bounding rectangle without creating a field.

C
```
pascal void StrInBox (short left, short top, short right, short bottom,
             const Str255 Text, long DispSpec, short Just);
```

Pascal
```
procedure StrInBox (left, top, right, bottom: INTEGER;
             Text: STRING; DispSpec: LONGINT; Just: INTEGER);
```

StrInBox is a substitute for the toolbox's TextBox routine. It offers several improvements not found in TextBox:
- the concept of a single-line display area
- text color options
- background options including: "clear before display" and "transparent background"
- sensitivity to multiple monitors with different settings
- sensitivity to presence or absence of Color QuickDraw.

The string is drawn using the current window's font, size and style settings (as set by the TextFont, TextSize, and TextFace routines).

Left, *top*, *right*, and *bottom* define a rectangle in local co-ordinates that determines the display area and location in the current window. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom). The rectangle must be wide enough for at least 1 character. For best results, the height of the rectangle should be the same as a font's height (font height can be determined by calling the GetFontInfo routine and adding *Ascent* + *Descent* + *Leading*). If multiple lines are displayed, the height should be in increments of the font height. If the rectangle's height is not greater than the font's height, a *single-line* display area exists and word wrap is not applied to the text.

Text is the string that is displayed in the rectangle.

DispSpec specifies how the text and its background are drawn. The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result.

Choose only one of the following background options…

`teWhiteBack`	Before the Text is drawn, the area enclosed by the rectangle is erased with white. This is the default, so omitting all options implies using this one.
`teBackdrop`	Before the Text is drawn, the area enclosed by the rectangle is erased with the window's backdrop color.

`teColorBack`	Before the Text is drawn, any portion of the rectangle that is on a monitor with a bit depth of 4-bits or more is erased using the window's background color. Otherwise, the rectangle is erased using white. If Color QuickDraw is unavailable (or unused), the rectangle is erased using white.
`teNoBack`	The rectangle's area is unaffected before the text is drawn, thus having the appearance of drawing on a transparent background. This option is best for drawing text over patterns or pictures.

Choose only one of the following text color options…

`teBlackText`	Text is black. This is the default, so omitting all options implies using this one.
`teColorText`	If any portion of the text is drawn on a monitor with a bit depth of 4-bits or more, it is drawn using the window's foreground color. Otherwise, the text is black. If Color QuickDraw is unavailable (or unused), the text is black.

Just specifies if a field is left aligned, right aligned, or centered. See the relevant constants at the end of this section.

Also see: StrInBoxRect, TextInBox and TextInBoxRect.

☞ **Warning**: StrInBox must be called outside a BeginUpdateScreen / EndUpdateScreen structure.

```
CONST                                  {Background Drawing:                                      }
               teWhiteBack   = $0000;  {  White background (default)                             }
               teBackdrop    = $0001;  {  Draw on backdrop color                                 }
               teColorBack   = $0002;  {  Color background                                       }
               teNoBack      = $0008;  {  Transparent background                                 }
                                       {Text Drawing:                                            }
               teBlackText   = $0000;  {  Black text (default)                                   }
               teColorText   = $0010;  {  Foreground colored text                                }
                                       {Combined text and background constants:                  }
               teBlackOnBackdrop = teBlackText + teBackdrop;    {Black text on backdrop          }
               teBlackOnWhite    = teBlackText + teWhiteBack;   {Black text on white             }
               teBlackOnColor    = teBlackText + teColorBack;   {Black text on color             }
               teBlackOnClear    = teBlackText + teNoBack;      {Black text, no background       }
               teColorOnBackdrop = teColorText + teBackdrop;    {Color text on backdrop          }
               teColorOnWhite    = teColorText + teWhiteBack;   {Color text on white             }
               teColorOnColor    = teColorText + teColorBack;   {Color text on color             }
               teColorOnClear    = teColorText + teNoBack;      {Color text, no background       }
                                       {Text alignment:                                          }
               teJustLeft    = 0;      {  Left aligned (default)                                 }
               teJustCenter  = 1;      {  Centered                                               }
               teJustRight   =-1;      {  Right aligned                                          }
```

StrInBoxRect

Draw a string in a bounding rectangle without creating a field.

C
```
pascal void StrInBoxRect (const Rect *Bounds, const Str255 Text,
             long DispSpec, short Just);
```

Pascal
```
procedure StrInBoxRect (Bounds: RECT; Text: STRING;
             DispSpec: LONGINT; Just: INTEGER);
```

StrInBoxRect is identical to the StrInBox routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

TextInBox

Draw text in a bounding rectangle without creating a field.

C
```
pascal void TextInBox (short left, short top, short right, short bottom,
            const Ptr TextPtr, short Length, long DispSpec, short Just);
```

Pascal
```
procedure TextInBox (left, top, right, bottom: INTEGER; TextPtr: PTR;
            Length: INTEGER; DispSpec: LONGINT; Just: INTEGER);
```

TextInBox is identical to StrInBox except instead of passing a string, you pass the following parameters:

TextPtr is a pointer to the text being displayed.

Length is the length of the text being displayed.

TextInBoxRect

Draw text in a bounding rectangle without creating a field.

C
```
pascal void TextInBoxRect (const Rect *Bounds, const Ptr TextPtr,
            short Length, long DispSpec, short Just);
```

Pascal
```
procedure TextInBoxRect (Bounds: RECT; TextPtr: PTR; Length: INTEGER;
            DispSpec: LONGINT; Just: INTEGER);
```

TextInBoxRect is identical to the TextInBox routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

DrawPict

Draw a picture (PICT resource).

C
```
pascal void DrawPict (short BaseID,
            short left, short top, short right, short bottom, long DispSpec);
```

Pascal
```
procedure DrawPict (BaseID: INTEGER;
            left, top, right, bottom: INTEGER; DispSpec: LONGINT);
```

DrawPict is a replacement for the toolbox's DrawPicture routine. It offers several improvements not found in DrawPicture:

- selective scaling or clipping to a viewing rectangle
- drawing a picture using its original size (without decoding the resource)
- background options including "clear before display" and "transparent background"
- sensitivity to multiple monitors and settings (different PICTs can be used for different monitor settings)
- sensitivity to presence or absence of Color QuickDraw
- shifting a picture within a clipped viewing rectangle (see DrawShiftPict).

BaseID specifies the base 'PICT' resource ID number. If you are drawing a single PICT resource, *BaseID* represents the ID number of that resource. If you are displaying different pictures depending on the monitor's settings, *BaseID* is the resource number of the Black and White picture (all other pictures in the set are numbered higher than this one). See "Resource IDs" later in this section for a detailed description of 'PICT' resource numbering.

Left, *top*, *right*, and *bottom* define a rectangle in the current window's local co-ordinates where the picture is displayed. These parameters can be seen as two corners; the upper left-hand corner (left,top) and the bottom right-hand corner (right,bottom) of a display rectangle. You can optionally retain the picture's original proportions (as detailed later by *DispSpec*), in which case the picture's upper left-hand corner is drawn at the co-ordinates specified by *left* and *top*.

DispSpec specifies how the picture is displayed (clipping, scaling, background options, resource searching, etc.) The value for this 4-byte long integer can be specified either by adding a set of constants to obtain the desired result, or by using a specially defined variant record. See the "Appearance and Behavior" section for details.

PICT Resource IDs

You do not need to read this section if you are displaying a single 'PICT' resource. This section applies only if you need to display a different picture depending on the monitor's settings.

Up to six versions of the same picture can be created, one for each of the different monitor settings shown below. By default, DrawPict looks for the PICT resource that is best suited for the monitor's settings. This is true even if the picture straddles multiple monitors. If the best suited picture is not available, DrawPict looks for an equivalent picture that is normally used when the monitor is at the next lower setting (gray is a lower setting than color). You can override this behavior by telling DrawPict to use a *higher* quality image and have QuickDraw remap the colors. For example, you may decide to create only a Black and White (1-bit) picture and an 8-bit color equivalent, and when the monitor is set to 4-bit gray or color, or 8-bit gray, to use the 8-bit color picture instead of dropping down to the Black and White one.

The following chart describes the PICT resource numbering scheme for creating pictures that are sensitive to monitor settings (this also takes Macs with multiple monitors into account):

	'PICT' Resource ID	
Screen Depth	**Gray**	**Color**
B&W	Base ID	
4-bit	Base ID + 1	Base ID + 2
8-bit	Base ID + 3	Base ID + 4
16+ bit		Base ID + 5

Please observe Macintosh user interface guidelines by creating at least the Black and White picture (with a resource ID number equal to *BaseID*).

Appearance and Behavior

DispSpec specifies how the picture is displayed (clipping, scaling, background options, resource searching, etc.) The value for this 4-byte long integer can be specified either by adding a set of constants to obtain the desired result, or by using a specially defined variant record, as illustrated below.

`pictScale1PICT` — Use this constant *alone* to draw a single PICT, and to have it scaled to the specified rectangle. This is the default way of displaying a picture, and is functionally equivalent to the toolbox's DrawPicture routine.

`pictUsePictRect` — Use the resource's rectangle for drawing the picture. This option ensures that the picture is drawn using exactly the same height and width as when it was created. The picture's upper left-hand corner is drawn at the co-ordinates specified by *left* and *top*.

`pictClipToRect` — Clip the picture within the rectangle specified by *left*, *top*, *right* and *bottom*. If you are not using pictClipToRect or pictUsePictRect, the picture is scaled to fit into the rectangle.

`pictOnBackdrop`	Clear the display rectangle using the window's backdrop color before drawing the picture. This is useful for clearing out a previous picture or making sure the image is displayed on a background that is the same color as the window's backdrop. Without this option, your picture will be drawn on top of whatever exists in the display rectangle as though the new picture has a transparent background.
`pictOnWhite`	Clear the display rectangle with white before drawing the picture. This is useful for clearing out a previous picture or making sure the image is displayed on a white background. Without this option, your picture will be drawn on top of whatever exists in the display rectangle as though the new picture has a transparent background.
`pictOnColor`	Clear the display rectangle using the window's background color before drawing the picture. This is useful for clearing out a previous picture or making sure the image is displayed on the window's background color. Without this option, your picture will be drawn on top of whatever exists in the display rectangle as though the new picture has a transparent background.
`pictMultiPICT`	Use this constant if you are displaying different pictures depending on the monitor's settings.
`pictBWplus`	When used in conjunction with pictMultiPICT, this option tells DrawPict to use a higher resolution picture if a Black and White resource can't be found (the Black and White picture has a resource ID equal to *BaseID*).
`pictGray4plus`	When used in conjunction with pictMultiPICT, this option tells DrawPict to use a higher resolution picture if a 4-bit gray scale resource can't be found (the 4-bit gray scale picture has a resource ID equal to *BaseID* + 1).
`pictColor4plus`	When used in conjunction with pictMultiPICT, this option tells DrawPict to use a higher resolution picture if a 4-bit color resource can't be found (the 4-bit color picture has a resource ID equal to *BaseID* + 2).
`pictGray8plus`	When used in conjunction with pictMultiPICT, this option tells DrawPict to use a higher resolution picture if an 8-bit gray scale resource can't be found (the 8-bit gray scale picture has a resource ID equal to *BaseID* + 3).
`pictColor8plus`	When used in conjunction with pictMultiPICT, this option tells DrawPict to use a higher resolution picture if an 8-bit color resource can't be found (the 8-bit color picture has a resource ID equal to *BaseID* + 4).

As an example, if you want to draw a picture that uses multiple PICTs (depending on the monitor's settings), and you want to retain the picture's original proportions, you should use the combined constants pictMultiPICT + pictUsePictRect. Alternatively, a C structure and a Pascal variant record are available to help you define the *DispSpec* in a more intuitive way, as shown below:

C

```
union TPDrawPictSpec {                          /*Picture drawing appearance & behavior          */
                                                /* specifications in 2 formats…                  */
   struct{                                      /* · Parsed into components:                     */
      short bits31to16;                         /*     (reserved bits)                           */
      unsigned short bit15 :1;                  /*     (reserved bit)                            */
      unsigned short bit14 :1;                  /*     (reserved bit)                            */
      unsigned short bit13 :1;                  /*     (reserved bit)                            */
      unsigned short bit12 :1;                  /*     (reserved bit)                            */
      unsigned short bit11 :1;                  /*     (reserved bit)                            */
      unsigned short Color8plus :1;             /*    Use better pict if 8-bit color not avail   */
      unsigned short Gray8plus :1;              /*    Use better pict if 8-bit gray not avail    */
      unsigned short Color4plus :1;             /*    Use better pict if 4-bit color not avail   */
      unsigned short Gray4plus :1;              /*    Use better pict if 4-bit gray not avail    */
      unsigned short BWplus :1;                 /*    Use better pict if B&W not available       */
      unsigned short MultiPICT :1;              /*    Use different PICT/monitor setting         */
      unsigned short DrawOnColor : 1;           /*    Clear with background color before drawing? */
      unsigned short DrawOnWhite : 1;           /*    Clear with white before drawing?           */
      unsigned short DrawOnBackdrop : 1;        /*    Clear with backdrop color before drawing?  */
      unsigned short ClipToRect :1;             /*    Clip picture to rectangle?                 */
   unsigned short UsePictRect :1;               /*    Use picture's rectangle?                   */
   } Bits;                                      /*                                               */
   long Num;                                    /* · Long equivalent                             */
};                                              /*                                               */
typedef union TPDrawPictSpec TPDrawPictSpec;
```

Pascal

```
TPDrawPictSpec = packed record                 {Picture drawing appearance and behavior          }
                                               {  specifications in 2 formats…                   }
    case integer of                            {                                                 }
     0: (                                      { · Parsed into components:                       }
       bits31to16: integer;                    {       (reserved bits)                           }
       bit15, bit14, bit13: boolean;           {       (reserved bits)                           }
       bit12, bit11: boolean;                  {       (reserved bits)                           }
       Color8plus: boolean;                    {     Use better pict if 8-bit color not available }
       Gray8plus: boolean;                     {     Use better pict if 8-bit gray not available  }
       Color4plus: boolean;                    {     Use better pict if 4-bit color not available }
       Gray4plus: boolean;                     {     Use better pict if 4-bit gray not available  }
       BWplus: boolean;                        {     Use better pict if B&W not available         }
       MultiPICT: boolean;                     {     Use different PICT per monitor setting       }
       DrawOnColor: boolean;                   {     Clear with background color before drawing?  }
       DrawOnWhite: boolean;                   {     Clear with white before drawing?             }
       DrawOnBackdrop: boolean;                {     Clear with backdrop color before drawing?    }
       ClipToRect: boolean;                    {     Clip picture to rectangle?                   }
       UsePictRect: boolean;                   {     Use picture's rectangle?                     }
     );                                        {                                                 }
     1: (                                      { · Integer equivalent:                           }
       Num: longint;                           {     Specification longint                       }
     );                                        {                                                 }
   end;
```

As an example, lets draw a picture that uses multiple images (depending on the monitor's setting), clears the destination rectangle before drawing, and uses an 8-bit color image in place of 4-bit color or gray or 8-bit gray. The following code sample illustrates how this is done:

```
procedure DoItNow;
  var
    DispSpecs: TPDrawPictSpec;              {Declare the variable used for the Display Specs   }
  begin
    DispSpecs.Num := 0;                     {Initialize all the bits to zero values            }
    DispSpecs.MultiPICT := true;            {Use multiple PICTs                                }
    DispSpecs.DrawOnWhite := true;          {Clear destination rect with white before drawing  }
    DispSpecs.Gray4plus := true;            {Use better image if 4-bit gray PICT not found     }
    DispSpecs.Color4plus := true;           {Use better image if 4-bit color PICT not found    }
    DispSpecs.Gray8plus := true;            {Use better image if 8-bit gray PICT not found     }
    DrawPict(myBaseID, 10, 10, 275, 300, DispSpecs.Num);  {Draw the picture                    }
```

You can use whatever you like best as the *DispSpec*, a single constant, several constants added together, a variable, or the short or 4-byte long integer component of a structure or variant record.

Also see: DrawPictRect, DrawShiftPict, and DrawShiftPictRect.

☞ **Warning**: DrawPict must be called outside a BeginUpdateScreen / EndUpdateScreen structure.

Note: To avoid icon and picture conflicts while you are developing your application, avoid resource numbers that are used by your development environment (THINK C or THINK Pascal). THINK C and THINK Pascal sometimes supply their *own* resources in place of those in your resource file whenever resources numbers coincide.

You can create and edit resources with a resource editor such as Apple's ResEdit. Remember to use ID numbers 128 or higher. The rest are reserved numbers.

```
CONST                                  {Picture drawing behavior & appearance Specs: }
        pictScale1PICT     = $0000;    {Scale PICT to specified rectangle            }
        pictUsePictRect    = $0001;    {Use picture's rectangle?                     }
        pictClipToRect     = $0002;    {Clip picture to rectangle?                   }
        pictOnBackdrop     = $0004;    {Clear with backdrop color before drawing?    }
        pictOnWhite        = $0008;    {Clear with white before drawing?             }
        pictOnColor        = $0010;    {Clear with backgnd color before drawing?     }
        pictMultiPICT      = $0020;    {Use different PICT per monitor setting       }
        pictBWplus         = $0040;    {Use better pict if B&W not available         }
        pictGray4plus      = $0080;    {Use better pict if 4-bit gray not available  }
        pictColor4plus     = $0100;    {Use better pict if 4-bit color not available }
        pictGray8plus      = $0200;    {Use better pict if 8-bit gray not available  }
        pictColor8plus     = $0400;    {Use better pict if 8-bit color not available }
```

DrawPictRect

Draw a picture (PICT resource) in a bounding rectangle.

C
```
pascal void DrawPictRect (short BaseID, const Rect *Bounds, long DispSpec);
```

Pascal
```
procedure DrawPictRect (BaseID: INTEGER; Bounds: RECT; DispSpec: LONGINT);
```

DrawPictRect is identical to the DrawPict routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

DrawShiftPict

Draw a picture (PICT resource) offset in its frame.

C
```
pascal void DrawShiftPict (short BaseID,
              short left, short top, short right, short bottom,
              short dh, short dv, long DispSpec);
```

Pascal
```
procedure DrawShiftPict (BaseID: INTEGER;
              left, top, right, bottom: INTEGER;
              dh, dv: INTEGER; DispSpec: LONGINT);
```

DrawShiftPict is identical to the DrawPict routine, except that it accepts additional parameters for offsetting the picture within the specified rectangle.

Dh and *dv* specify the horizontal and vertical number of pixels by which the picture is offset. If *dh* and *dv* are positive, the offsetting is to the right and down. If either are negative, offsetting is in the opposite direction.

DrawShiftPictRect

Draw a picture (PICT resource) offset in its frame.

C
```
pascal void DrawShiftPictRect (short BaseID, const Rect *Bounds,
            short dh, short dv, long DispSpec);
```

Pascal
```
procedure DrawShiftPictRect (BaseID: INTEGER; Bounds: RECT;
            dh, dv: INTEGER; DispSpec: LONGINT);
```

DrawShiftPictRect is identical to the DrawShiftPict routine, except that it accepts the *Bounds* rectangle in place of the individual left, top, right and bottom co-ordinates.

DrawIcon

Draw an icon and optionally make it appear selected (darkened) or disabled (grayed).

C
```
pascal void DrawIcon (short theIcon, short left, short top,
            Boolean EnabledFlag, Boolean SelectedFlag);
```

Pascal
```
procedure DrawIcon (theIcon, left, top: INTEGER;
            EnabledFlag, SelectedFlag: BOOLEAN);
```

TheIcon is the icon ID that is displayed. This includes *any* of the standard Macintosh icons: cicn, icl8, icl4, ICN#, ICON, ics8, ics4, ics# and SICN.

Left and *top* define the top left-hand corner in the current window's local co-ordinates where the icon is drawn.

The *EnabledFlag* indicates if the icon is drawn as enabled or not. An enabled icon is displayed normally, whereas a disabled icon is dimmed or "grayed out." The two constants that can be used for this flag are *enabled* and *disabled.* In nearly all cases, you will want to draw the icon as enabled. If you were to superimpose the icon with a zone from a cursor table (to make it a click-sensitive area and behave like a button), you may want to draw it as disabled to signify that it can't be selected.

The *SelectedFlag* indicates if the icon is to be drawn as selected or not. An unselected icon is displayed normally, whereas a selected icon is darkened. The two constants that can be used for this flag are *selected* and *notSelected.* In nearly all cases, you will want to draw the icon as notSelected. If you were to superimpose the icon with a zone from a cursor table (to make it a click-sensitive area and behave like a button), you may want to draw it as selected.

Intelligent Icon Drawing

The DrawIcon routine is flexible in that it can draw color and black & white icons of various types under any conditions. It does so by using the most appropriate icon in an *icon family* (detailed below), depending on the number of colors displayed, and whether the icon is selected, not selected, enabled, or disabled.

If your application runs on a Macintosh with multiple monitors, DrawIcon automatically draws the icon correctly, even if the icon straddles multiple screens. For this reason, DrawIcon must be placed outside any BeginUpdateScreen / EndUpdateScreen structures.

Icon Family

DrawIcon can draw any of the following icon types:

cicn	variable size 8-bit color icon + monochrome (1-bit) equivalent + mask
icl8	32x32 pixel 8-bit color icon used primarily by the Finder as your application's icon
icl4	32x32 pixel 4-bit color icon used primarily by the Finder as your application's icon
ICN#	32x32 pixel monochrome (1-bit) icon + mask used primarily by the Finder as your application's icon
ICON	32x32 pixel monochrome (1-bit) icon
ics8	16x16 pixel 8-bit color icon used primarily by the Finder as your application's icon
ics4	16x16 pixel 4-bit color icon used primarily by the Finder as your application's icon
ics#	16x16 pixel monochrome (1-bit) icon + mask used primarily by the Finder as your application's icon
SICN	16x16 pixel monochrome (1-bit) icon*

*Although SICN resources can contain several icons, DrawIcon only draws the first one in the resource.

When a specific icon ID is shared by two or more of the icon types listed above, those related icons are called an "icon family." The simplest example of an icon family is found in the bundle (BNDL) resource in almost all Macintosh applications. The bundle resource, usually just called a bundle, is an organization of icons used by the Finder to display your application (and its related documents) on the desk top and in folders. The icon family with an ID of 128 is usually the icon that depicts your application, where icl8 is the 8-bit icon, icl4 is the 4-bit icon, and ICN# is the 1-bit icon and mask.

DrawIcon makes use of icon families to display the best available icon. You can create and edit icons with a resource editor such as Apple's ResEdit. When you create icons, remember to use ID numbers 128 or higher. The rest are reserved numbers.

Note: To avoid icon conflicts while you are developing your application, do not create icon numbers that are used by your development environment (THINK C or THINK Pascal). THINK C and THINK Pascal sometimes supply their *own* icons in place of those in your resource file whenever icon numbers coincide.

Warning: If you are using a 'cicn' (variable size color) icon that may be displayed on a Macintosh that doesn't have Color QuickDraw, make sure the icon's size is set to at least 9 pixels wide (although the actual image and mask can be smaller). A bug in the Macintosh's ROMs causes a crash when CopyBits tries to work on a BitMap that is 8 pixels wide or less. Tools Plus circumvents this bug by not displaying the 'cicn'.

Icon Selection

When you specify an icon ID, DrawIcon selects an appropriate icon from the available icon family based on [1] the number of colors available on the screen, and [2] whether the icon is selected or not. If Color QuickDraw is not available on the Macintosh running your application (Macintosh 512KE and Macintosh Plus only), or if your application has chosen *not* to use Color QuickDraw when executing the InitToolsPlus routine, DrawIcon will behave as though you are using a monochrome (black and white) monitor.

When selecting an appropriate icon from the specified icon family (as depicted in the table below), DrawIcon first attempts to find a large icon. Failing that, it will search through the small icons. When searching for an icon, DrawIcon determines the number of colors available on the screen. It then checks if the icon is selected or not. Once these two criteria have been determined, DrawIcon tries to find the optimum icon type in the icon family. If the optimum icon type is not available, DrawIcon descends to the second best choice, and so on down the list. DrawIcon will use icons intended for a lesser number of colors (i.e., from 256 to 16) if an appropriate icon isn't available to take advantage of all the colors the screen has to offer.

Icon Size	Screen Depth (in bits)	Available Colors	Icon Selection Order	
			Not Selected	Selected
Large or Variable	8 (or more)	256 (or more)	icl8 cicn	icl8 cicn
	4	16	icl4 cicn	
	1 or 2	4 or black & white	ICN# ICON cicn†	ICN# ICON cicn†
Small	8 (or more)	256 (or more)	ics8	ics8
	4	16	ics4	
	1 or 2	4 or black & white	ics# SICN	ics# SICN

†Black and white portion only.

If, for example, your application is running on an 8-bit monitor and it is trying to display a selected icon with an ID of 130, DrawIcon will always look for icon types whose ID is 130, and will first try to display an icl8. If an icl8 can't be found, it will try to find a cicn. If neither of those icons can be found, DrawIcon will continue down the list to the icons best suited for a 4-bit monitor. Since there aren't any in the "selected" column, it will descend again to 2-bit monitors, where it will try to find ICN#, ICON and cicn icon types. If no large icon is found, DrawIcon repeats the process for small icons by first searching for an ics8, then an ics# and SICN icon.

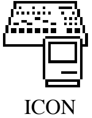
Drawing the Icon, Selecting, Disabling, and Masking

All icon types are drawn within a limited square, such as the 32x32 bit ICON or 16x16 bit ics4. When the icon is displayed by DrawIcon, the icon's image *and* the background within the boundaries of the square are drawn on the window. You can liken this process to using MacPaint and selecting the image by using the frame selection tool (not the lasso), then copying that image and pasting it on the destination window. As you will notice, the background is also pasted.

Selecting the icon darkens it when using color or gray-scale, except for icl4 icons which can't be darkened (in which case a substitute monochrome icon is used and darkened). On black and white monitors, the icon image is inverted (black turns to white and white turns to black).

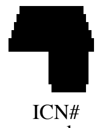
Disabling the icon overlays a gray pattern to make the icon look "grayed out." You can specify the appearance of disabled icons by using the DefaultIconLook routine.

Each of these processes, drawing, selecting, and disabling, is done using the icon's bounding square. If you want to limit the process to a smaller area, say, for example, to an area that is identical to an oddly shaped icon image, you can incorporate an icon mask. All drawing, selecting and disabling is performed only within the area defined by the icon mask. The cicn, ICN#, and ics# icons have integrated masks. If you use an icl8 or icl4 icon, also include an ICN# icon with the same ID, and DrawIcon will use its mask. The same applies for ics8 and ics4 icons which rely on the ics# icon for a mask. The example below demonstrates the difference a mask makes.

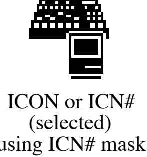


ICON
not selected

ICON
(selected)
no mask



ICN#
mask



ICON or ICN#
(selected)
using ICN# mask

Creating Your Own Icons

You can create your own icons using a resource editor, such as Apple's ResEdit. When creating cicn, icl8 and ics8 icons with ResEdit, you have a choice of two pallets: "Apple icon colors" or "Standard 256 colors." If your icon is going to be drawn as selected, use only the Apple icon colors, since they will guarantee that your icon can be darkened properly. Note that icons that are displayed as both selected and disabled are usually difficult to read, since simultaneously darkening and grayed out often makes the image illegible.

You can also replace the system's standard Stop, Note, and Warning icons with your own customized icons, simply by creating icons with the same IDs and including them in your application.

Also see: DrawSICN and DrawSICNmode.

☞ **Warning**: DrawIcon must be called outside a BeginUpdateScreen / EndUpdateScreen structure.

DrawSICN

Draw an SICN (small indexed) icon.

C
```
pascal void DrawSICN (short theIcon, short left, short top, short index);
```

Pascal
```
procedure DrawSICN (theIcon, left, top, index: INTEGER);
```

DrawSICN is similar to DrawIcon except that is handles a very specific case: the small indexed icon, the 'SICN' resource. SICNs can have one or more 16x16 pixel black and white images in each resource. The DrawIcon routine displays only the first image whereas this routine can display any of them. The image is drawn using the srcCopy mode.

TheIcon is the SICN resource ID that is displayed.

Left and *top* define the top left-hand corner in the current window's local co-ordinates where the icon is drawn.

Index specifies which image in the resource is drawn (1 being the first, 2 being the second, etc.) If *index* does not represent an image that exists on the resource, DrawSICN does nothing.

Also see: DrawSICNmode.

DrawSICNmode

Draw an SICN (small indexed) icon using any transfer mode.

C
```
pascal void DrawSICNmode (short theIcon, short left, short top, short index,
           short mode);
```

Pascal
```
procedure DrawSICNmode (theIcon, left, top, index, mode: INTEGER);
```

DrawSICNmode is identical to DrawSICN except that is requires an additional *mode* parameter that specifies the transfer mode. The constants you can use for the transfer mode are srcCopy, srcOr, srcXor, srcBic, notSrcCopy, notSrcOr, notSrcXor, or notSrcBic.

DefaultIconLook

Specify the default appearance for disabled icons.

C
```
pascal void DefaultIconLook (long IconSpec);
```

Pascal
```
procedure DefaultIconLook (IconSpec: LONGINT);
```

The DefaultIconLook routine specifies the appearance of disabled icons that are drawn by DrawIcon (Tools Plus also uses DrawIcon whenever it needs to draw an icon). Your application should use DefaultIconLook early on to ensure that all icons have the same appearance. If your application uses DefaultIconLook later in its execution, it will only affect icons that are drawn (or refreshed) after the change is made.

IconSpec specifies the default appearance for all icons. The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result See the section below for details. Tools Plus's picture buttons also rely on this setting to draw disabled buttons, however, each picture button can have its own settings which do not use the default.

Default Appearance for Disabled Icons

IconSpec specifies the default appearance for all icons. The value for this 4-byte long integer can be specified by adding a set of constants to obtain the desired result, as illustrated below. The following table illustrates the three possible dimming effects (only one can be chosen at a time), plus an option that preserves the icon's border when it is drawn as disabled. 8-bit and 1-bit (black and white) examples are provided to show you the effect in various environments. The table is based on a single icon, the standard "printer" icon in color, and black and white.

	Appearance of Disabled Icon							
	Border *is* affected by disabling				Border unaffected (`DfltIconLeaveBorder`)			
	8-bit (Color or Gray)		Black & White		8-bit (Color or Gray)		Black & White	
Dimming Options (use one only)	not selected	selected	not selected	selected	not selected	selected	not selected	selected
Enabled icons (no disabling effect applied)								
`DfltIconDimBlackLtPat` Overlay image with black color using a "light gray" (25%) pattern.								
`DfltIconDimWhiteLtPat` Overlay image with white color using a "light gray" (25%) pattern.								
`DfltIconDimWhitePat` Overlay image with white color using a "medium gray" (50%) pattern.								

The optional DfltIconLeaveBorder constant is used to preserve the icon's border by limiting the dimming effect to an area that is 1 pixel smaller than the icon's mask (icons that do not have a mask, such as the 'ICON', have an effective mask that is the icon's entire area).

If your application ever has the need to globally change the default appearance and have it instantly reflected in all the windows, add the DfltIconUpdateNow constant to the IconSpec. This option forces a doRefresh event for all open windows in your application, thereby insuring that all icons are redrawn using the new default. Picture buttons using the default settings are also updated.

```
CONST                                       {Disabled icons' default appearance…      }
              DfltIconDimBlackLtPat =  $01;  {  Overlay Black color w/ Lt Gray pattern }
              DfltIconDimWhiteLtPat =  $02;  {  Overlay White color w/ Lt Gray pattern }
              DfltIconDimWhitePat   =  $04;  {  Overlay White color w/ Gray pattern    }
              DfltIconLeaveBorder   =  $20;  {Leave border when selected or disabled   }
              DfltIconUpdateNow     =$8000;  {Update all windows with changes          }
```

Maintaining Indexed String ('STR#') Structures

An indexed string record is a structure that contains from 0 to 32767 Pascal strings, each of which can be from 0 to 255 characters in length. Unlike an ordinary array of strings, an indexed string structure is *dynamic*, in that new strings can be added to the structure and existing strings can be deleted. Another significant difference is that the amount of memory required to store each string in the structure is less than in a traditional array of strings -- an indexed string structure uses one byte per character in the string (plus a length byte) whereas an array allocates the maximum string length.

Although indexed string structures are much more memory-efficient than string arrays, accessing their data is slower. The performance penalty is relatively minor, but it increases as the *number* of strings in the structure increases.

An indexed string record has the same structure as an 'STR#' resource. It begins with a 2-byte integer (short) whose value represents the number of strings in the structure, followed by the strings' data. Because the indexed string record is dynamic (its size can grow or shrink as its data changes), it is always referenced by a handle. Always use Tools Plus routines to maintain the data in an indexed string record.

There are two ways to get a handle to an indexed string record: [1] load an 'STR#' resource, or [2] use Tools Plus's NewIndexStringHandle routine to create a new, empty indexed string record. In both cases, a handle to the record is returned to you. You can then use Tools Plus routines to:

NewIndexStringHandle	create a new indexed string record
GetIndexString	read a string
SetIndexString	change the value of a strings
InsertIndexString	add a new string to the structure (insert between records or append to end)
DeleteIndexString	delete a string from the structure
CountIndexString	count the number of strings in the structure

☞ **Warning**: Tools Plus routines perform no integrity checks on your indexed string record (it is assumed that your record is not corrupted). Create 'STR#' resources using a reputable resource editor. Use only Tools Plus routines to modify the indexed string record.

NewIndexStringHandle

Create a new indexed string record.

C
```
pascal Handle NewIndexStringHandle (short NumberOfStrings);
```

Pascal
```
function NewIndexStringHandle (NumberOfStrings: INTEGER): HANDLE;
```

NewIndexStringHandle creates a new indexed string record and allocates the minimum space required to store the number of strings specified by the *NumberOfStrings* parameter. The number of bytes allocated by this routine are NumberOfStrings + 2.

A handle to the indexed string record is returned. It is best to never lock this handle since there is never a need to do so.

CountIndexString

Determine the number of strings in an indexed string record.

C
```
pascal short CountIndexString (Handle hRec);
```

Pascal
```
function CountIndexString (hRec: HANDLE): INTEGER;
```

hRec is a handle to an indexed string record (as created by NewIndexStringHandle), or a handle to an 'STR#' resource.

The routine returns the number of strings in the indexed string record. This routine does not actually count the number of strings in the structure. It simply returns the value of a counter at the beginning of the record.

GetIndexString

Get a string from an indexed string record.

C
```
pascal void GetIndexString (Handle hRec, short Index, Str255 Text);
```

Pascal
```
procedure GetIndexString (hRec: HANDLE; Index: INTEGER; var Text: Str255);
```

hRec is a handle to an indexed string record (as created by NewIndexStringHandle), or a handle to an 'STR#' resource.

Index specifies the string you want to receive (1 = first string, 2 = second string, etc). This number should be in the range of 1 to the value of CountIndexString.

Text is the string obtained from the indexed string record belonging to the *hRec* handle. If the value of *index* does not represent a string that exists in the record, *Text* is returned as a null string.

SetIndexString

Store a string in an indexed string record.

C
```
pascal void SetIndexString (Handle hRec, short Index, const Str255 Text);
```

Pascal
```
procedure SetIndexString (hRec: HANDLE; Index: INTEGER; Text: Str255);
```

hRec is a handle to an indexed string record (as created by NewIndexStringHandle), or a handle to an 'STR#' resource.

Index specifies the string you want to update with a new value (1 = first string, 2 = second string, etc). This number should be in the range of 1 to the value of CountIndexString.

Text is the new value for a string that exists in the indexed string record belonging to the *hRec* handle. If the value of *index* does not represent a string that exists in the record, SetIndexString does nothing.

☞ **Warning**: Make sure that the *hRec* handle is not locked when passing it to this routine (this handle *never* needs to be locked). If *hRec* is a handle to an 'STR#' resource, also make sure it is not flagged as "protected." Failure to observe this warning may result in corrupted data or unexpected results.

InsertIndexString

Insert a new string in an indexed string record, or append a new string to the end of the record.

C
```
pascal void InsertIndexString (Handle hRec, short Index, const Str255 Text);
```

Pascal
```
procedure InsertIndexString (hRec: HANDLE; Index: INTEGER; Text: Str255);
```

hRec is a handle to an indexed string record (as created by NewIndexStringHandle), or a handle to an 'STR#' resource.

Index specifies where the new string is added (1 = first string, 2 = second string, etc). Specify a value of CountIndexString + 1 to add a new string to the end of the record.

Text is the new string being added to the indexed string record belonging to the *hRec* handle. If the value of *index* is not valid, InsertIndexString does nothing.

The fastest way to add a set of strings to the end of the list is to make sure that NewIndexStringHandle does not allocate space for unused strings. For example, to add 100 new strings, use NewIndexStringHandle(0) to create an empty 'STR#' structure, then use InsertIndexString 100 times to add the new strings in sequence 1 to 100.

☞ **Warning**: Make sure that the *hRec* handle is not locked when passing it to this routine (this handle *never* needs to be locked). If *hRec* is a handle to an 'STR#' resource, also make sure it is not flagged as "protected." Failure to observe this warning may result in unexpected results.

DeleteIndexString

Delete a string from an indexed string record.

C
```
pascal void DeleteIndexString (Handle hRec, short Index);
```

Pascal
```
procedure DeleteIndexString (hRec: HANDLE; Index: INTEGER);
```

The specified string is deleted from the indexed string record. This is not the same as setting a string to null (where the string is merely set to "" to occupy less memory). When a string is deleted from an indexed string record, a subsequent call to CountIndexString will be decreased by 1.

hRec is a handle to an indexed string record (as created by NewIndexStringHandle), or a handle to an 'STR#' resource.

Index specifies which string is deleted (1 = first string, 2 = second string, etc). This number should be in the range of 1 to the value of CountIndexString.

☞ **Warning**: If *hRec* is a handle to an 'STR#' resource, also make sure it is not flagged as "protected." Failure to observe this warning may result in unexpected results.

BitMaps and PixMaps

Tools Plus provides routines for creating and destroying black and white and color bitmaps (called PixMaps). Within this section, the term *BitMap* is used to refer to a black and white bitmap, while *PixMap* is used for color bitmaps. The term *bitmap* refers to a generic bitmap object, either black and white or color. A detailed explanation of the need for bitmaps and their use is beyond the scope of this document. For complete information on BitMaps and PixMaps, see the Inside Macintosh reference manuals, or Apple's Macintosh Technical Note #120.

A BitMap, in its simplest terms, is a record for storing a bitmapped image. PixMaps are similar except they store color images. Two very common uses for bitmaps are:

- Temporarily copying an image from a window or screen: An example of this occurs when a pull-down menu temporarily obscures various objects such as parts of a window and its contents, desk accessories and the desk top. Just before the pull-down menu descends, a copy of the area it will cover is copied to a bitmap. When the pull-down menu withdraws, the image is copied from the bitmap back to its original location, thereby restoring the image.
- Animation: Animation is achieved by drawing a composite image on a bitmap, the image being made up of a background and the animated objects. Several bitmaps may exist: one for the background, one or more containing the various objects being animated, and one that is used as a scratch pad. When an animated object moves, your application copies a small part of the background bitmap to the scratch pad bitmap thereby covering up the object at its old location (i.e., the object is erased by the background). The object's image is copied from the object bitmap to the scratch pad bitmap in its new location. Finally, the sum of the object's old location and new location are copied from the scratch pad bitmap to the window. All the user sees is that the object has moved from its old location to a new one, without seeing the layer by layer buildup of the image or any screen flicker.

Creating a bitmap

Your application uses the CreateBitMap routine to create an off screen BitMap or PixMap. There are several different ways to create a bitmap depending on your application's needs. The simplest method is to create a 1-bit deep black and white BitMap. Your application simply specifies the BitMap's co-ordinates and CreateBitMap does the work.

Color bitmaps are a bit more complex because a PixMap needs a color table to correctly map RGB colors to the available number of colors that can be represented by the bit depth of the PixMap. The CreateBitMap routine has several ways of using a graphics device (GDevice) as the basis for creating a color PixMap. In each case, the PixMap makes a copy of the graphics device's color table for itself. The methods for using a GDevice to create a PixMap are as follows:

- Use the GDevice with the greatest depth that intersects a specified rectangle: This method of creating a PixMap is best when you want to temporarily store part of a window in a PixMap then later restore it. Your application specifies a rectangle in the current window's local co-ordinates and CreateBitMap uses the GDevice belonging to the monitor with the greatest pixel depth that intersects the rectangle.
- Use a specific GDevice: Your application can create a PixMap based on a specific GDevice, likely one you have created with your own color table. This method is most suited for developers who create a custom GDevice.
- Use the current GDevice: This is similar to the method above, except that it uses the current graphics device instead of the one with the greatest depth.
- Use the maximum GDevice. A PixMap is created using the GDevice with the greatest bit depth.

When your application creates a bitmap, the CreateBitMap routine returns a GrafPtr to the bitmap and in the case of a color bitmap, it also returns a handle to the GDevice that was used to create the bitmap. Your application can treat the GrafPtr like a pointer to a window, except that drawing occurs in the off screen bitmap and is not visible.

Drawing to a bitmap

Before you draw to a bitmap, determine the current grafPort and GDevice by using the toolbox's GetPort and GetGDevice. You will be changing one or both of these items, so it's a good idea to store their current settings so that you can restore them after you finish drawing to the bitmap.

Next, just as required when drawing to a window, you must make the bitmap the current grafPort by using SetPort and specify the bitmap's GrafPtr. This tells QuickDraw that subsequent drawing operations take place on the bitmap. You can now think of the bitmap as a window and draw accordingly utilizing the clip region, text settings and so on.

When using a color bitmap, some Color QuickDraw routines perform calculations based on the current graphics device's color table. An example of this is drawing disabled text using System 7's grayishTextOr text transfer mode. When text is drawn using grayishTextOr mode, the current graphics device's color table is used to calculate a suitable intermediate color or an appropriate dithering pattern. If your application uses routines that depend on the current GDevice, or if you are uncertain of this and you just want to be safe, set the current GDevice using SetGDevice and specify the GDHandle returned by CreateBitMap.

When you finish drawing to a bitmap, restore the original grafPort and GDevice by using the toolbox's SetPort and SetGDevice with the values you obtained prior to drawing. You will thereby insure a stable environment for the Mac OS, extensions such as QuickTime, and third party software products.

Copying to a bitmap or to a window

Images can be copied from a window to a bitmap, from bitmap to bitmap, and from a bitmap to a window. The easiest way to do this is to first make the target the current grafPort. This process is identical to preparing to draw in the target window or bitmap, as described in the previous section titled "drawing to a bitmap." Set the destination's foreground and background colors to black and white respectively to ensure accurate color mapping between the source and destination. Finally, use the toolbox's CopyBits routine to copy the image from the source to the destination. The line below is an example of how CopyBits is used to copy an image from a bitmap to a window:

```
CopyBits(bitmapPtr^.portBits, Wptr^.portBits, sourceRect, destRect, srcCopy, maskRgn);
```

CreateBitMap

Create a BitMap or a PixMap.

C
```
pascal Boolean CreateBitMap (GrafPtr *newPort, GDHandle *hGDevice,
            const Rect *Bounds, short TypeOfMap);
```

Pascal
```
function CreateBitMap (var newPort: GrafPtr; var hGDevice: GDHandle;
            var Bounds: RECT; TypeOfMap: INTEGER ): BOOLEAN;
```

This routine can create a bitmap in a variety of ways:

- black and white BitMap
- color PixMap based on an area in a window
- color PixMap based on a specified GDevice
- color PixMap based on the current GDevice
- color PixMap based on the GDevice with the greatest pixel depth

NewPort returns a pointer to a newly created and initialized GrafPort or CGrafPort (color grafPort). This grafPort contains the required BitMap or PixMap. A value of nil is returned if the required bitmap could not be created.

HGDevice is the graphics device (GDevice) upon which a PixMap is based. Normally, you specify nil. If you are creating a PixMap based on a known GDevice (i.e., when you include then bitmapFromGDevice option in the TypeOfMap parameter), specify the handle to the desired GDevice in *hGDevice*. This routine returns hGDevice set to the graphics device upon which the PixMap is based. A nil value is returned when a black and white BitMap is created.

Bounds specifies the dimensions of the bitmap being created. Tools Plus creates a bitmap with a width that is rounded up to a long integer (4 byte) boundary to provide the best performance. When *TypeOfMap* has a value of bitmapFromWindow, this rectangle also specifies an area in the current window that is being "matched" by the bitmap. The bitmap is created based on the maximum pixel depth within this rectangle. The resulting bitmap will have the same co-ordinates specified by *Bounds*. You can change these co-ordinates by using the toolbox's SetOrigin.

TypeOfMap specifies what kind of bitmap is created, and how it is created. One of the following constants should be used:

`bitmapBW`	Create a black and white BitMap using the co-ordinates specified by the Bounds rectangle.
`bitmapFromWindow`	If Color QuickDraw is available and being used (as defined by InitToolsPlus), create a color PixMap based on the GDevice with the greatest pixel depth that intersects the specified rectangle (Bounds) on the current window. If Bounds doesn't intersect the window, a bitmap is not created.
`bitmapFromGDevice`	If Color QuickDraw is available and being used (as defined by InitToolsPlus), create a color PixMap based on the GDevice specified by *hGDevice*. The PixMap has the same pixel depth as the specified GDevice, and has a copy of the GDevice's color table.
`bitmapFromCurrentGDevice`	If Color QuickDraw is available and being used (as defined by InitToolsPlus), create a color PixMap based on the current GDevice. The PixMap has the same pixel depth as the current GDevice, and has a copy of the GDevice's color table.
`bitmapFromMaxGDevice`	If Color QuickDraw is available and being used (as defined by InitToolsPlus), create a color PixMap based on the GDevice with the greatest pixel depth. The PixMap has the same pixel depth as the GDevice, and has a copy of the GDevice's color table.

This routine returns *true* if the bitmap was created. If the routine returns with a value of *false*, no bitmap was created. Bitmaps are typically not created because there is insufficient memory.

☞ **Warning**: Use GetPort to obtain the current grafPort before creating a bitmap or using SetPort to make a bitmap the current grafPort. When you are finished drawing to the bitmap, restore the original grafPort. Failure to do so may make parts of Tools Plus or other software malfunction.

```
CONST                                     {Methods and types of bitmaps…               }
        bitmapBW                 = 0;     {B&W bitmap (1-bit)                          }
        bitmapFromWindow         = 1;     {Color bitmap based on window's max pixel depth }
        bitmapFromGDevice        = 2;     {Color bitmap based on a specified GDevice     }
        bitmapFromCurrentGDevice = 3;     {Color bitmap based on the current GDevice     }
        bitmapFromMaxGDevice     = 4;     {Color bitmap based on maximum GDevice         }
```

DestroyBitMap

Destroy a BitMap or PixMap.

C
```
pascal void DestroyBitMap (GrafPtr oldOffscreen);
```

Pascal
```
procedure DestroyBitMap (oldOffscreen: GrafPtr);
```

OldOffscreen is a bitmap created by CreateBitMap or any other method of creating bitmaps. The following operations are performed on black and white BitMaps:

```
ClosePort(oldOffScreen);
DisposePtr(oldOffscreen^.portBits.baseAddr);
DisposePtr(oldOffscreen);
```

The following operations are performed on color PixMaps:

```
CloseCPort(oldOffscreen);
DisposeHandle(oldOffscreen^.portPixMap^^.pmTable);
DisposePtr(oldOffscreen^.portPixMap^^.baseAddr);
DisposePtr(oldOffscreen);
```

BitMap2Region

Convert a BitMap or PixMap to a region.

C
```
pascal OSErr BitMap2Region (RgnHandle Region, BitMap bMap);
```

Pascal
```
function BitMap2Region (Region: RgnHandle; bMap: BitMap): OSErr;
```

BitMap2Region is identical to the toolbox's BitMapToRegion routine, except that this one is available to applications running under any system version whereas BitMapToRegion is available only under System 7 or later. There is no need to use this routine if your source code always runs on System 7 or later (as is the case with PowerMacs). If your application is running on System 7 or later, BitMap2Region calls the toolbox's BitMapToRegion routine. In the PowerMac Tools Plus libraries, BitMap2Region simply calls the toolbox's BitMapToRegion routine and there is no alternate code.

Region is a handle to a valid region. The region is modified to be an equivalent to the supplied bitmap. Note that the region's co-ordinates match those of the bitmap (i.e., the top left corner is not necessarily set to 0,0. If the region's size exceeds the 32K limit it returns empty.

BMap is either a BitMap or PixMap record. When supplying a PixMap record, the bit depth must be 1 or the region will return empty and an error code will result.

The routine returns an OS error code resulting from the conversion. The possible error codes are:

noErr	(0)	No error
pixmapTooDeepErr	(-148)	Pixel map record is deeper than 1 bit per pixel
rgnTooBigErr	(-500)	Bitmap would convert to a region that is bigger than 32K

SystemVersion

Determine what version of the System file is being used.

C
```
pascal double SystemVersion (void);
```

Pascal
```
function SystemVersion: EXTENDED;
```

SystemVersion determines the version of the System file being used by your application (located on your startup disk). The System file's version is expressed as three numeric components separated by periods (i.e., 7.0.1). Because a floating-point number has a single decimal, the second period is omitted to present the decimal equivalent of 7.01.

Because this routine's source code is compiled as part of your application (it can be found in the ToolsPlus.c file for C programmers, and the ToolsPlus.p interface file for Pascal programmers), it is compiled according to your project's compiler settings for 680x0 processor optimization and/or math co-processor optimization. Therefore, SystemVersion returns a floating-point number that can be compared to constants in your source code, such as the following example:

```
if (SystemVersion >= 7.0) then
```

Programming Tips:

1 If your application uses SystemVersion often, or in ways in which processing speed is important, you should realize that floating-point operations are slower than integer operations, and they consume more memory. Instead of using the SystemVersion routine throughout your application, you can use the _SYSV routine which simply returns the System file version as an integer (i.e., system 7.0.1 is returned as 701). Then throughout your program, you can use constants to compare against the integer-formatted system version. The following example illustrates this:

```
const
  System6     = 600;                             {Use these constants to compare the system    }
  System7     = 700;                             {  file's version to specific version         }
  System7_0_1 = 701;                             {  numbers.                                   }
var
  SysVersion: integer;                           {Use this variable as the System file         }
                                                 {  version throughout your application.       }
begin
  SysVersion := _SYSV;                           {Get System version as an integer (i.e.       }
                                                 {  System 7.0.1 returns as 701)               }
  if (SysVersion >= System6) and (SysVersion <= System7) then  {This line checks for all       }
    begin                                        {  versions between 6.0 and 7.0 inclusive…    }
```

2 A second method, which is faster still and more memory conservative, makes all evaluations at the beginning of the application, then uses the *result* throughout.

```
const
  System6     = 600;                             {Use these constants to compare the system    }
  System7     = 700;                             {  file's version to specific version         }
  System7_0_1 = 701;                             {  numbers.                                   }
var
  SysVersion: integer;                           {Use this variable as the System file         }
                                                 {  version throughout your application.       }
  Sys6_to_Sys7: boolean;                         {Is system between 6.0 and 7.0?               }
  Sys7_plus: boolean;                            {Is system 7.0 or greater?                    }
begin
  SysVersion := _SYSV;                           {Get System version as an integer (i.e.       }
                                                 {  System 7.0.1 returns as 701)               }
  Sys6_to_Sys7 := ((SysVersion >= System6) and (SysVersion <= System7));  {Set boolean         }
  Sys7_plus := (SysVersion >= System7);          {  variables with values that are determined  }
                                                 {  now.  You'll use these booleans through-   }
                                                 {  out your application.                      }
  if Sys6_to_Sys7 then                           {This line checks for all system versions     }
                                                 {  between 6.0 and 7.0 inclusive.  It runs    }
                                                 {  quicker and takes up less memory than      }
                                                 {  comparing numbers.                         }
```

GetToolsPlusVersion

Determine what version of Tools Plus is being used.

C
```
pascal void GetToolsPlusVersion (Str255 Version);
```

Pascal
```
procedure GetToolsPlusVersion (var Version: Str255);
```

GetToolsPlusVersion determines the version of Tools Plus libraries being used by your application.

Version returns with a string containing the Tools Plus version number formatted in the following manner:

3.1.6a where	3	= Major version number	(significant revisions/enhancements to Tools Plus)
	1	= Minor release number	(minor revisions/enhancements)
	6	= Maintenance release number	(bug fixes, incidental changes)
	a	= Very minor patch	(recompile with newer compiler, typographical errors)

The Tools Plus version number is meant to be used for display purposes only, such as in an “About…” dialog. Your application is compiled with a *known* version of Tools Plus, so there is never any need to have conditional code based on the available version of Tools Plus unlike the Macintosh’s System version.

Beep

Play the System Error sound.

C
```
pascal void Beep (void);
```

Pascal
```
procedure Beep;
```

Beep replaces the Macintosh toolbox's SysBeep routine. On the Macintosh 512KE, Plus and SE, the parameter passed to the SysBeep routine specifies the duration (in clock ticks) for which the "Simple Beep" sound is played. If any other sound is selected (by the Sound control Panel), or if the sound is played on a computer other than the Macintosh 512KE, Plus and SE, the parameter is ignored.

Using Beep instead of SysBeep ensures that the System Error sound is played for the correct duration regardless of the computer on which it is running. It also saves you typing, as well as two bytes of application per use.

In later versions of System 7, the Sound Manager plays the System Error sound asynchronously, so when your application calls the Beep routine, control is immediately returned to your application to allow it to perform other tasks such as opening a window while the System Error sound plays out. This enhancement may have some implications if you expect a rapid response from the user as a result of the beep because your application can queue several beeps before the first one finishes playing. For example, a fast typist can types five illegal key strokes in the time it takes the first error sound to play.

Also see: BeepSynch.

BeepSynch

Play the System Error sound synchronously.

C
```
pascal void BeepSynch (void);
```

Pascal
```
procedure BeepSynch;
```

BeepSynch is identical to the Beep routine except it plays the System Error sound synchronously, meaning that control is returned to your application when the System Error sound finishes playing. This is useful in situations where you do not want to queue up a number of beeps as a result of the user's actions, such as when the user types several illegal key strokes. What you can do is maintain a global variable that keeps track of when the last System Error sound completed playing by calling the toolbox's TickCount routine immediately after BeepSynch. Then when each illegal keystroke is detected, play BeepSynch only if the keystroke occurred after the completion of the most recent System Error sound (the keystroke's time is reported by Tools Plus in Event.Event.when). From the user's perspective, the single System Error sound applies to the first detected keystroke and to all illegal keystrokes that that were made while the sound was playing.

Wait

Wait for a specified time.

C
```
pascal void Wait (long ClockTicks);
```

Pascal
```
procedure Wait (ClockTicks: LONGINT);
```

ClockTicks specifies the time that is to be waited in sixtieths of a second, or clock ticks. The Wait routine returns to the calling application after the specified number of clock ticks have transpired. If the number is less than or equal to zero, Wait returns immediately.

One example of using this routine is when an application is first launched. After initializing Tools Plus, your application may decide to display an identification window that tells the user about the program and version number. While the window is displayed, your application can open files and obtain resources. You may decide to display the window for no less than 3 seconds, in which case you would first obtain the current number of ticks since startup from the TickCount routine, display the identification window, then perform your other duties. When these tasks are completed, call TickCount again to determine how much time has transpired, then use Wait to delay for 180 ticks (3 seconds x 60 ticks), less the number of ticks that have passed since the window was first opened.

Warning: Use Wait judiciously because events will still be accumulating in the event queue while your application is waiting, and older events may be lost. Also, other applications won't be getting any processing time while your application is waiting.

SynchToVideo

Synchronize to video (wait until vertical retrace occurs before continuing).

C
```
pascal void SynchToVideo (void);
```

Pascal
```
procedure SynchToVideo;
```

When displaying objects in quick succession, such as during object animation or when dragging a line across a window, an undesirable side effect called "strobing" is often seen. Strobing manifests as white flickers, missing steps in a sequence of images, or other image instability. Call SynchToVideo immediately before erasing/displaying an image. SynchToVideo waits until the vertical retrace is executed, then it returns control to your application. The duration of the wait can be as much as 1/60 of a second.

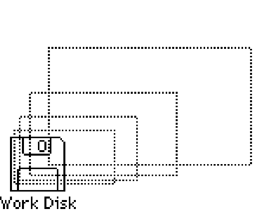
ZoomLines

Draw "zoom lines" from one rectangle to another.

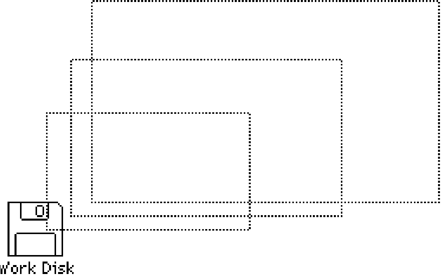
C
```
pascal void ZoomLines (const Rect *OldRect, const Rect *NewRect, short Zoom);
```

Pascal
```
procedure ZoomLines (OldRect, NewRect: RECT; ZoomType: INTEGER);
```

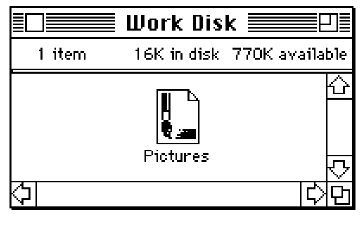
The drawing of "zoom lines" creates an illusion of transition between two objects. A good example of zoom line use is in the Finder. Whenever a document, application, or disk is opened, zoom lines expand from the object. This creates an illusion that the object is *opening*. Zoom lines also create an apparent screen depth, since they make an object appear to zoom towards the user when it is opening, and zoom back down onto the screen when it closes. This process is called "zooming in" or "zooming out," as illustrated as follows:



Step 1 -- Lines start emanating from the starting rectangle, which in this case is a floppy disk. Several zoom lines are visible at the same time.



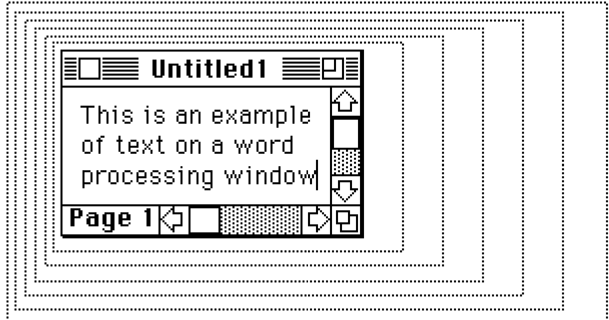
Step 2 -- At first, the zoom lines move slowly and are closely spaced, since they are in the “distance.” They accelerate as they approach their final size. The lines are dotted on a white background, and black on the desk top’s gray background.



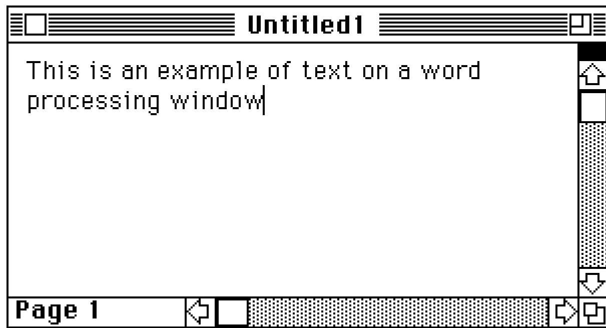
Step 3 -- When the final size is reached, the zoom lines disappear and the new object, which in this example is an open window, appears.

The example above demonstrates “zooming out.” The process of “zooming in” is reversed: the expanded object disappears and zoom lines are drawn from the co-ordinates of the expanded object down to the co-ordinates of the closed object.

Another type of zoom lines is available, and that is “zooming across.” This involves a single object’s transition from one shape to another. An example of zooming across is when a window’s “zoom box” is clicked. The window changes from a user state to a standard state (or vice versa). When zooming across, the zoom lines move at a constant rate from the old co-ordinates to the new. Because no acceleration or deceleration occurs, screen depth appears to be constant. Zooming across is illustrated below:



Step 1 -- A single rectangle moves at a constant rate, steadily making the transition between the old co-ordinates and the new. The rectangle is dotted on a white background, and white on the desk top’s gray background.



Step 2 -- The last rectangle has the same co-ordinates as the new rectangle. The old object disappears and is recreated using the new co-ordinates.

OldRect and *NewRect* specify the original and destination rectangles in the screen’s global co-ordinates. Transition always occurs starting at *OldRect* and ending at *NewRect*, regardless if you are zooming in, zooming out, or zooming across.

ZoomType specifies the type of zoom operation being performed. The constants for this are: ZoomIn, ZoomOut and ZoomAcross.

```
CONST                         {Zoom Types                             }
             ZoomIn     =-1;  {zoom down to an object                 }
             ZoomOut    = 1;  {zoom up from an object                 }
             ZoomAcross = 0;  {transition from one object to another  }
```

DrawThermometer

Draw the standard Macintosh progress thermometer in the current window.

C
```
pascal void DrawThermometer (const Rect *DisplayRect, long Value,
            long Maximum);
```

Pascal
```
procedure DrawThermometer (DisplayRect: RECT; Value, Maximum: LONGINT);
```

DrawThermometer draws the standard Macintosh progress thermometer, such as the one seen in the Finder when a file is being copied or duplicated. The thermometer is drawn according to the system that your Macintosh is running, and the settings of the monitor(s) on which the thermometer is displayed. The thermometer drawn by this routine is always similar to the one seen in the Finder.

DisplayRect is the display rectangle in the current window's local co-ordinates. If the rectangle is taller than it is wide, a vertical thermometer is created. Otherwise a horizontal thermometer is drawn. If you are using Aaron, a system extension that runs under System 7 to give you the desk top icons, windows and controls as seen in Mac OS 8, set the display rectangle to be either 11 or 12 pixels high to make Aaron draw the thermometer using a Mac OS 8 style.

Value and *Maximum* are used to express the percentage completed (completed = value ÷ maximum). Progress is indicated in a left to right motion on horizontal thermometers, and from the bottom up on vertical ones. Use negative values to have the thermometer indicate progress in the reverse direction.

✍ **Note**: Even though DrawThermometer draws the thermometer very quickly, it may not be fast enough for applications that perform many thousands or even millions of operations during the thermometer's progression. In that case, use the toolbox's TickCount routine to time calls to DrawThermometer at a less frequent rate, perhaps once per second.

✍ **Note**: If your application determines that the Appearance Manager is present with the HasAppearanceManager routine, create your thermometer using NewScrollBar and use the kControlProgressBarProc procID (80). A standard System 7 thermometer is also available as a CDEF in SuperCDEFs.

☞ **Warning**: DrawThermometer must be called outside a BeginUpdateScreen / EndUpdateScreen structure.

HasAppearanceManager

Determine if the Macintosh running your application has an Appearance Manager.

C
```
pascal Boolean HasAppearanceManager (void);
```

Pascal
```
function HasAppearanceManager: BOOLEAN;
```

HasAppearanceManager informs your application if the Macintosh on which it is running has an Appearance Manager. If it does then your application has access to Appearance Manager-savvy windows and controls as documented in Inside Macintosh. This is a good way to determine if you need to use a custom window definition (WDEF resource) such as the Infinity Windoid to create a floating palette, or a custom control definition (CDEF resource) such as SuperCDEFs to create a slider.

If the Macintosh running your application has an Appearance Manager, then the HasAppearanceManager routine returns with a value of *true*. This does not mean that standard windows and controls will be mapped to the Apple gray scale appearance, only that your application can make use of the additional window definitions and control definitions that are supplied by the Appearance Manager.

✍ **Note**: This routine always returns with a value of *false* if you initialize Tools Plus (using the InitToolsPlus routine) without the initAppearanceManagerSavvy option. Doing so helps simulate your application running on a Mac without the Appearance Manager.

Also see: HasAppearanceManagerRoutines, UsingAppearanceManager, ReplaceWindowProcID and

ReplaceControlProcID.

HasAppearanceManagerRoutines

Determine if your application has access to Appearance Manager routines.

C
```
pascal Boolean HasAppearanceManagerRoutines (void);
```

Pascal
```
function HasAppearanceManagerRoutines: BOOLEAN;
```

HasAppearanceManagerRoutines informs your application if it has access to the Appearance Manager's routines. You only need to be aware of this if your application takes advantage of the Appearance Manager's features beyond the simple use of its 3D buttons, scroll bars, and windows.

In 680x0 code, this routine always returns the same value as the HasAppearanceManager routine (i.e., if the Macintosh running your application has an Appearance Manager, then your application has access to its routines).

In PowerPC code, references to Appearance Manager routines made by Tools Plus libraries must be resolved when you link your application, either to real Appearance Manager routines found in the AppearanceLib (included with your compiler), or with dummy routines found in the ToolsPlus.c or ToolsPlus.p files. When you compile your application with the UseAppearanceManager #define (or Pascal compiler directive) set to 1, you will be required to include the AppearanceLib library, and your application will have access to the Appearance Manager's routines. In this case, HasAppearanceManagerRoutines returns with the same value as the HasAppearanceManager routine. On the other hand, if you leave the UseAppearanceManager #define (or Pascal compiler directive) undefined or you set it to 0, Tools Plus links in dummy Appearance Manager routines to allow your application to link with Tools Plus libraries. In this case, HasAppearanceManagerRoutines returns *false*, and your application should not call any Appearance Manager routines.

✍ **Note**: Your application should not use any Appearance Manager routines if HasAppearanceManagerRoutines returns *false*. It can still use the 3D buttons, scroll bars, and windows providing the Appearance Manager is available.

✍ **Note**: This routine always returns with a value of *false* if you initialize Tools Plus (using the InitToolsPlus routine) without the initAppearanceManagerSavvy option. Doing so helps simulate your application running on a Mac without the Appearance Manager.

Also see: HasAppearanceManager, UsingAppearanceManager, ReplaceWindowProcID and ReplaceControlProcID.

UsingAppearanceManager

Determine if the Macintosh running your application has an Appearance Manager and it is running.

C
```
pascal Boolean UsingAppearanceManager (void);
```

Pascal
```
function UsingAppearanceManager: BOOLEAN;
```

UsingAppearanceManager informs your application if the Macintosh on which it is running has an Appearance Manager, and that it is running. When the Appearance Manager is running it automatically maps standard System 7 window and control definitions to use the Apple gray scale appearance, which is essentially a 3D look. This effectively makes regular applications take on a 3D look, at least in terms of the windows' frames, buttons and scroll bars. The Appearance Manager can also be "turned off" by the user to put the Mac into a "System 7 compatibility" mode in which your application's window frames and controls are not altered (they look like regular, flat controls).

If the Macintosh running your application has an Appearance Manager and it is turned on (i.e., not in "System 7 compatibility" mode), then the UsingAppearanceManager routine returns with a value of *true*. In this case, your windows and controls will take on the Apple gray scale appearance providing that you used standard Apple procIDs for your windows and controls, and that you did not override the window and control definitions with custom WDEFs

and CDEFs that use the same procIDs as Apple.

If you want to make sure your application has a floating palette and/or 3D buttons regardless if the Appearance Manager is running, you will need a floating palette WDEF such as the Infinity Windoid and 3D controls like SuperCDEFs. You can switch between using standard window and control definitions and custom ones by using ReplaceWindowProcID and ReplaceControlProcID.

Note: This routine always returns with a value of *false* if you initialize Tools Plus (using the InitToolsPlus routine) without the initAppearanceManagerSavvy option. Doing so helps simulate your application running on a Mac without the Appearance Manager.

Also see: HasAppearanceManagerRoutines, HasAppearanceManager, ReplaceWindowProcID and ReplaceControlProcID.

SetToZero

Set all bytes in a record to zero.

C
```
pascal void SetToZero (const void *RecPtr, long RecSize);
```

Pascal
```
procedure SetToZero (RecPtr: PTR; RecSize: LONGINT);
```

SetToZero initializes any variable, record or structure to zeros. This effectively sets numbers to zero values, booleans to *false*, strings to a length of zero, and pointers and handles to nil.

RecPtr is a pointer to the first byte of the affected object. In C, you can specify an object's address by using the ampersand, such as &MyObject. In Pascal, @MyObject accomplishes the same thing.

RecSize is the size of the affected object specified in bytes. Starting at the address indicated by *RecPtr*, *RecSize* number of bytes are set to zero. If *RecSize* is less than one, no change occurs.

EqualMem

Determine if two structures are equal.

C
```
pascal Boolean EqualMem (const void *Ptr1, const void *Ptr2,
            long BytesToCheck);
```

Pascal
```
function EqualMem (Ptr1: PTR; Ptr2: PTR; BytesToCheck: LONGINT): BOOLEAN;
```

EqualMem determines if two structures are equal to each other. This routine can be used to test small structures such as Color QuickDraw's "color" records, or very large structures.

Ptr1 and *Ptr2* are pointers to the two objects being tested. In C, you can specify an object's address by using the ampersand, such as &MyObject. In Pascal, @MyObject accomplishes the same thing.

BytesToCheck is the number of bytes being tested. This number is usually the object's size specified in bytes. EqualMem makes no assumptions about the size of each record. It merely tests the specified number of bytes for equality, starting at the locations specified by *Ptr1* and *Ptr2*.

The routine's value returns *true* if the specified number of bytes in the first record matches those in the second.

Note: Record packing can insert unused bits or bytes into a record to align fields (or subrecords) to certain boundaries. For example, a 10 character string in THINK Pascal actually consumes 12 bytes of memory (1 length byte + 10 data bytes + 1 unused byte for alignment to an even boundary). To account for this, ensure that the two records are equal to each other *bit for bit* before making changes. You can do this by using BlockMove to copy one record into another.

☞ **Warning**: Structures containing strings that are altered will not be equated correctly because EqualMem does not ignore the “garbage bytes” that may exist between the last valid character and the end of the string’s record boundary.

Min

Determine the minimum value of two numbers.

C
```
pascal long Min (long Val1, long Val2);
```

Pascal
```
function Min (Val1, Val2: LONGINT): LONGINT;
```

Val1 and *Val2* are two numbers whose value can be of integer or longint type.

The routine’s value returns the lesser value of *Val1* or *Val2*.

Max

Determine the maximum value of two numbers.

C
```
pascal long Max (long Val1, long Val2);
```

Pascal
```
function Max (Val1, Val2: LONGINT): LONGINT;
```

Val1 and *Val2* are two numbers whose value can be of integer or longint type.

The routine’s value returns the greater value of *Val1* or *Val2*.

21 Multiple Languages

Tools Plus libraries support languages other than English (such as French, German, Italian, etc.) by letting you replace words and phrases that are displayed by Tools Plus. The best example of this is how Tools Plus handles your application's Edit menu. Tools Plus automatically maintains the first five items in the Edit menu (Undo, Cut, Copy, Paste and Clear) by enabling and disabling them appropriately, and by changing the Undo item's text to reflect the currently available command, such as "Can't Undo", "Undo Cut", "Redo Cut", etc. As you may notice, all these words and phrases are in the English language, but Tools Plus lets you replace those words with the language of your choice as your application starts up, and it lets you change languages while your program is running (just in case you wrote an application that lets the user change the language under the application's control).

Where do those words appear?

The table included later in this chapter is a comprehensive list of each of the phrases and words, and where they appear in Tools Plus. There are four areas where Tools Plus presents words or phrases that are dependent on a language:

- Tools Plus error and warning messages
- the Edit menu's Undo item (changes to "Undo Cut", "Redo Cut", "Undo Copy", Redo Copy", etc.)
- User Notification (the message delivered by the Notification Manager)
- Dynamic Alerts (default button names)

Changing the words

Your application can replace Tools Plus's standard English phrases and words with another language by including an 'STR#' resource containing the appropriate replacement strings (detailed later). When your application calls InitToolsPlus, Tools Plus looks for an 'STR#' resource with an ID of 32767. If it is found, this resource's strings are used to replace the English words found throughout Tools Plus.

If your application gives the user the ability to change languages under application control, use the ToolsPlusLanguage routine to specify another 'STR#' resource ID that contains the strings in the language of your choice.

The STR# Resource

Your application can have multiple 'STR#' resources, one for each language your application supports (other than English), plus 'STR#' resources of your own that won't be used for this purpose. ID number 32767 is reserved by Tools Plus as the default resource that is loaded when Tools Plus is initialized. If this resource is missing, Tools Plus will initialize using English words throughout.

If your application supports only the English language, do not include an 'STR#' resource with the ID of 32767. You won't save any memory by including a resource and shortening or removing the strings.

If your application is localized for a single language other than English, create a single 'STR#' resource with an ID of 32767, or make a copy of the one we've included in the "Optional Resources" folder.

If your application supports multiple languages other than English, create an 'STR#' resource for each supported language (or copy ours). If your application starts up with a default language, give the related 'STR#' resource an ID of 32767, and all other supported languages a unique ID (likely 32766 and descending).

It is a smart idea to flag these resources as non-purgeable, locked, protected and pre-loaded to ensure that they are always available to Tools Plus. Even though this may consume a few hundred additional bytes of memory, it won't fragment memory and it will free you from the worry of running out of memory when you can least afford to.

No	English Equivalent	Use in Tools Plus & Comments
1	**Can't Undo**	Edit menu's "Undo" item. Tools Plus automatically changes the item to the correct phrase.
2	**Undo**	
3	**Undo Cut**	
4	**Undo Copy**	
5	**Undo Paste**	
6	**Undo Typing**	
7	**Undo Clear**	
8	**Redo Cut**	
9	**Redo Copy**	
10	**Redo Paste**	
11	**Redo Typing**	
12	**Redo Clear**	
13	**OK**	Default button titles in Dynamic Alerts. Do not change the meaning of the first seven items (i.e., the first one must be an equivalent to "OK", the second must be an equivalent to "Cancel"). If you want to change buttons' titles in Dynamic Alerts, use the AlertButtonName routine to do so, our use the AlertBox3 routine.
14	**Cancel**	
15	**Yes**	
16	**No**	
17	**Continue**	
18	**Skip**	
19	**Quit**	
20		(reserved for future consideration - leave blank)
21		(reserved for future consideration - leave blank)
22	**"^0" needs your attention.** **Please choose "^0" from the ^1 menu or click the "^0" window.**	Default message displayed by the Notification Manager when your application is suspended and requires attention. ^0 is replaced by your application's name. ^1 is replaced by the symbol in System 6, and the word "Application" under System 7.
23	**ERROR: Parameter passed to a Tools Plus routine is not within the legal range of values.**	Error displayed by Tools Plus when your application passes a parameter to a Tools Plus routine, and the parameter's value is outside of a legal range of values. This message includes spaces to make it look good when displayed in Chicago 12pt.
24	**This application will run only on a Macintosh 512KE or higher.**	Error message displayed when calling InitToolsPlus if the Mac running your application is a Lisa (Mac XL), Mac 128K or standard Mac 512K, none of which are supported by Tools Plus. If your application is PowerMac native without 680x0 code, you can leave this string empty.
25	**Tools Plus libraries must all be the same version.**	Error message displayed when calling InitToolsPlus if the correct number of Tools Plus libraries are included in your application, but one or more libraries are from a different version of Tools Plus. If your application is PowerMac native without 680x0 code, you can leave this string empty.
26	**Low memory...** **Continue without "Undo/Redo"?**	Just before the user makes a change in a field, Tools Plus checks to see if there is enough free memory to (1) complete the edit, and (2) implement the Undo/Redo feature. One of these messages is displayed if there isn't enough free memory.
27	**WARNING...** **Not enough memory for this operation.**	
28	**WARNING... Low memory!**	Periodic reminder when the user is typing in a field and memory is getting low.

'STR#' resource used to change phrases and words in Tools Plus to another language

ToolsPlusLanguage

Change the language in which Tools Plus messages, phrases and words are displayed.

C
```
pascal void ToolsPlusLanguage (short ResID);
```

Pascal
```
procedure ToolsPlusLanguage (ResID: INTEGER);
```

ResID specifies the 'STR#' resource ID number that contains the strings used to replace the English phrases in Tools Plus. Use a value of 0 (zero) to revert to English.

This routine has an immediate effect on the Edit menu's "Undo" item. It also replaces Dynamic Alert button names with the new language's defaults, thereby overriding any changes you may have previously made.

22 Other Macintosh Features

This section of the user manual deals with parts of the Macintosh toolbox that you may or may not decide to incorporate into your application. They are:

- Alerts
- Dialogs
- Custom Controls
- Lists

Except for Custom Controls, Tools Plus provides viable alternatives to the Macintosh's native toolbox routines. This chapter details those alternatives.

Alerts

Macintosh alert boxes, as described in the Dialog Manager in Inside Macintosh, can be used with Tools Plus with no ill effects. Because alerts are modal windows, and the Macintosh's Dialog Manager takes care of creating and maintaining the alert's window as well as the objects in the window, no conflicts will arise.

You should consider using Tools Plus's AlertBox routine, which gives you several advantages that regular alert boxes don't:

- Because you don't have to use Apple's Resource Editor (ResEdit), you can create alert boxes as the need for them arises, and you don't have to leave your development environment to do it.
- Your application's source code will be more readable because the icon, text, and button configuration will all be stated in the same line of source code.
- The text in the dynamic alert box is self-aligning to make the text look aesthetically pleasing. You have to take care of this yourself when designing alert boxes as resources.
- Your alert boxes are *always* centered on the main monitor regardless of the Macintosh your application is running on or the System version it's running. Standard Macintosh alerts won't.

Dialogs

Tools Plus supports dialog ('DLOG') resources, but because it completely circumvents the Macintosh toolbox's Dialog Manager, Tools Plus does not inherit any of the difficulties or short-comings that are regularly encountered when working with dialogs. Tools Plus handles multiple modeless and modal dialogs with ease, regardless of their complexity. Full details about Tools Plus's implementation of dialogs can be found in this manual's Windows chapter.

We strongly recommend that you use only Tools Plus routines to create and maintain dialogs, and that you avoid using the Dialog Manager and its routines for this purpose. The reason for this is that the Dialog Manager is by far less sophisticated than Tools Plus is and it encumbers you with much more restrictive demands. If you *must* use the Dialog Manager, then do so only with modal dialogs.

Custom Controls

Custom controls (CDEFs) can be created by programmers and used in applications that are created with Tools Plus. If you want Tools Plus to automatically manage these controls to behave like push buttons, check boxes, radio buttons, scroll bars or sliders, create the control using Tools Plus's NewButton or NewScrollBar routine.

If you have custom controls that you want to handle yourself, you can create them using standard toolbox routines. Tools Plus recognizes then as "foreign" controls and does not act upon them. All interaction with the custom control must be handled by your application. When a mouse-down event occurs in a custom control that is flagged in this manner, Tools Plus generates a doClickControl event and provides the Event Manager's event to your application.

See the Special Routines chapter in this manual. It will explain the caution that needs to be exercised when using some of the Macintosh's toolbox routines.

Lists

The List Manager's routines may be incorporated into your application to handle sophisticated or custom lists. Your application must handle all events pertaining to the list and its controls. Beware of the toolbox's LClick routine. It modifies the control's reference value (the contrlRfCon field of the control record), so if you store anything in the reference constant of a list box's scroll bar(s), you may lose that value by calling LClick.

See the Special Routines chapter in this manual. It will explain the caution that needs to be exercised when using some of the Macintosh's toolbox routines.

In most situations, Tools Plus's lists will fulfill your application's needs for creating and maintaining a list. Considering how easy Tools Plus's lists are to create and use, why not use them instead of the Macintosh's List Manager?

Warning: Apple's List Manager (and the LDEF written by Apple) is limited to 32K of data. This means that your list can't contain more than thirty-two thousand characters. Third-party LDEFs address this issue. For the most comprehensive solution to your list and table needs, see Introduction chapter and read the section named "The List Manager, List Boxes, Tables and Beyond."

23 Memory

All of code in Tools Plus libraries, and the objects it creates exist in your application's heap (the memory you allocate to an application in the "Get Info" dialog under MultiFinder or System 7 or higher). The only exception to this is Tools Plus's global record that is used to keep its internal workings functioning. The Tools Plus global record is kept on your application's stack. Fortunately, the global record is less than 3K, so it consumes relatively little stack space of the 32K limit for 680x0 applications (the stack on a Power Mac is not limited to 32K).

Almost all the objects created by Tools Plus routines are relocatable, and are therefore accessed internally by handles. The use of relocatable objects eliminates memory fragmentation caused by non-relocatable objects (locked handles, or dynamically allocated objects created by using a pointer). The only non-relocatable objects created by Tools Plus are window records, but this is not a problem providing your application calls InitToolsPlus early in your application. Calling InitToolsPlus early allocates the window records in memory where they won't cause any heap fragmentation.

You should be careful to avoid creating dynamically allocated non-relocatable objects, if possible. This will prevent memory fragmentation and make your application or plug-in much more memory efficient. In cases where you cannot avoid creating dynamically allocated non-relocatable objects, such as when you create UPPs for event handler routines, you should create these objects as early in your application as possible to minimize or eliminate memory fragmentation.

Each routine (Tools Plus or otherwise) consumes stack memory just to be able to execute its own code. After the routine has completed execution, its stack memory is automatically released. The user interface elements created by Tools Plus routines (such as buttons, list boxes, etc.) continue to occupy memory in your application's heap until they are explicitly deleted, or their parent window is closed, or when your application ends.

It is your responsibility to ensure that an operation can be completed with the amount of memory that is available in your application. For example, testing can determine the exact amount of memory required to open a window with 5 buttons, two list boxes, and 40 editing fields. Your application should ensure that the memory required to create such a window can be allocated safely *before* the window is opened.

Testing Memory Requirements

The best way to test how much memory your application needs to perform an operation, is to so see how much memory is available before the operation is performed, and to compare that value to how much memory is available after the operation is performed. This should be done during the testing phase of your project in a fashion that resembles the steps below. The code detailed below is only temporary, and should be removed after you have the information you require. In the example below, we'll be opening a dialog with numerous user interface elements on it.

1. Define three long integer (32 bits) variables: memBefore, memAfter, memAfterPurge
2. Just before opening the dialog, call the toolbox's MaxMem routine. You can ignore the routine's parameters. MaxMem purges all purgeable objects from your heap.
3. Call the toolbox's FreeMem routine to determine the amount of free memory that is available to your application. Save this value in the memBefore variable.
4. Open the dialog and create any user interface elements and dynamic objects that are associated with that window. This operation consumes memory in two ways. First, objects that permanently consume memory, such as buttons and fields, will consume the memory they require. This memory is consumed for as long as the window stays open. Second, temporary (purgeable) objects may be loaded into memory. These objects are needed on a temporary "as needed" basis, typically when the window is first opened, and possibly when the window is refreshed. A 'PICT' resource is an example of such an object, providing it is flagged as "purgeable."
5. Call the toolbox's FreeMem routine to determine the amount of free memory that is available to your application. Save this value in the memAfter variable.
6. Call the toolbox's MaxMem routine. You can ignore the routine's parameters.
7. Call the toolbox's FreeMem routine to determine the amount of free memory that is available to your application. Save this value in the memAfterPurge variable.

You now have three pieces of information:

Memory needed to first open your window	= memAfter - memBefore
Memory needed to keep your window open	= memAfterPurge - memBefore
Additional memory needed to refresh your window	= memAfterPurge - memAfter

You can display this information in a temporary alert, then delete all the code you used to determine the window's memory consumption.

Testing for Memory Availability

If your application can operate safely and with certainty with the minimum amount of memory specified in its "Get Info" box in the Finder, then you never need to test for memory availability. To be sure of this, force your application to consume the maximum amount of memory and use the Finder's About This Macintosh feature to see how much free memory is available to your application. Forcing the use of application memory usually entails opening all windows, filling all editing fields with the maximum amount of text, then clicking in an inactive full field and typing and filling it again (to force the "undo buffer" to fill).

You will most likely choose not to be so generous with your use of memory, especially in larger applications that can potentially consume many megabytes of memory if it is available, and that can also operate with much smaller memory spaces if required. In such cases, your application needs to check for the availability of memory before it attempts to consume it.

Before your application attempts to open a window or perform some other function that will consume memory, we recommend that you determine if the available memory is sufficient to perform the required function. You want to ensure that there is enough memory to perform the required operation, and you also want to make sure that other items, such as other open windows, have sufficient memory to keep working properly (i.e., you don't cut into other item's work memory). You start off by having a global constant or #define that represents the worst-case example of required working memory (shown as "Additional memory needed to refresh your window" a few paragraphs earlier). If you have three windows requiring 5K, 8K and 16K of working memory respectively, then your worst-case is 16K. This figure represents memory that you always want to have free to be able to keep things running smoothly. We recommend that you make this figure at least 100K over the amount you believe to be true, just to add a margin of safety. We'll refer to this value as *workingMemoryNeeded* throughout this text.

Write a routine that takes a single parameter (the total amount of memory needed) and returns a boolean to indicate if that required memory is available. The following routines are provided as examples:

C

```
Boolean EnoughMemory (long ramRequired)
  {
  Boolean  ok;
  long     xLong;

  if (ramRequired <= FreeMem())            // Enough memory without compacting heap?
    ok = true
  else {
    xLong = MaxMem(xLong);                 // Purge dynamic objects, compact memory
    ok = (ramRequired <= FreeMem());
    };
  if (!ok)
    NotEnoughMemoryAlert();                // This is your own lack of memory alert
  return (ok);                             // Return function's value
  }
```

Pascal

```
function EnoughMemory (ramRequired: LONGINT): BOOLEAN;
  var
    ok:    BOOLEAN;
    xLong: LONGINT;
begin
  if ramRequired <= FreeMem then           {Enough memory without compacting heap?   }
    ok := true
  else
    begin
      xLong := MaxMem(xLong);              {Purge dynamic objects, compact memory    }
      ok := (ramRequired <= FreeMem)
    end;
  if not ok then
    NotEnoughMemoryAlert;                  {This is your own lack of memory alert    }
  EnoughMemory := ok;                      {Return function's value                  }
end;
```

Once you have the above routine, your application can use it to test if the next operation will fit into the available memory, and if not, a dialog is displayed with an appropriate message such as "Insufficient free memory to perform operation. Try closing other windows, or quit the application and give it more memory in the 'Get Info' dialog." An example of using this routine is as follows:

```
if EnoughMemory (ramNeededForWindow1 + workingMemoryNeeded) then
    OpenWindow1;
```

In the example above, *ramNeededForWindow1* represents the amount of memory that is needed to keep the window open (memAfterPurge - memBefore), while *workingMemoryNeeded* represents the worst case requirement of opening a window or refreshing it (memAfterPurge - memAfter).

Editing Fields

Each editing field can consume up to 32K of memory, depending on how much text is stored by the field. This amount is minimized by using length-limited editing fields which limits stored text to a specified number of characters (i.e., 30 character field instead of 32K). Additionally, the one field on each window that contains the cursor when the window is active, can consume another 32K of memory (temporarily edited text versus permanently stored text). This too is minimized when a length-limited field is used. Lastly, the one field that is currently being edited by the user stores a temporary copy of the field's text when the user starts typing to allow an "undo/redo" operation.

To summarize, the working memory required for all editing fields in your application is as follows:

1 Add the maximum editable text for all fields in all windows
2 Add the maximum editable text for the largest field in each window
2 Add the single, largest value for a single window you calculated in step 2

Keep in mind that Tools Plus performs advanced memory checks when editing fields. If memory is tight, it will warn the user that an undo operation will not be available, and will allow a field to be edited. If memory is still tighter, Tools Plus will periodically warn the user of insufficient memory as he/she is typing. In the worst case, Tools Plus will not allow a field to be activated if there is not enough memory.

Handle Blocks

Many Tools Plus objects contain handles to other objects, and because of this, they consume memory from *handle blocks*. Handle blocks are a pool of handles that are available to any process or object that requires them. When a process or object no longer requires a handle, the handle is released back into the available pool. Additional memory will not be required providing that the demand for handles never exceeds the available supply. As soon as the demand for handles exceeds the available supply, an additional block of 64 handles is automatically created, thereby consuming 512 bytes of memory. An entire block will be created even if only one more handle is needed.

It is always best to allocate the number of required Handle Blocks by specifying the count as a parameter in InitToolsPlus when your initialize your application. This prevents memory fragmentation. A product like ZoneRanger

(included with CodeWarrior) can be used to determine how many handles are used by your application, and how many are free. To determine the handle requirements for your application, run it as extensively as possible, excercising every opetion and window available, the use ZoneRanger to determine the number of handles in use and the number that are available. Take this sum (used + available), divide by 64, and this is the number of handle blocks required by your application. Make sure that your create at least that many when you initialize Tools Plus by using the InitToolsPlus routine.

The Style Table

Tools Plus uses a memory-efficient method of storing font information for objects that use styles (buttons, scroll bars, editing fields, list boxes, panels and pop-up menus). The Style Table keeps a record for each unique combination of font, font size, and font style. These settings are stored in an 8-byte record that is automatically referenced by the GUI elements that use those settings.

An 8-byte Style Table record is created for each unique combination of font, font size, and font style, regardless of the number of objects that use the same setting, while the objects that refer to those styles use only a 2-byte reference each.

Good memory habits

It's a very good idea to check the amount of contiguous memory that is available before allocating objects or calling a routine that may need to be loaded from disk (i.e., a 680x0 code segment that is not yet in memory). Do the same when you open a document too, because there are a number of applications out there that bomb just because a user created a document on a 8-meg Mac and tried to open it on another Mac with less memory. Users perceive this type of behavior as being indicative of an unreliable program ("it keeps bombing on me for *no reason*").

24 Font Heights

The following is a list of font heights (in pixels) for some of the Macintosh's most popular fonts. A font's height for any font can be determined by calling GetFontInfo and adding *Ascent* + *Descent* + *Leading*. Please note that these numbers are for reference purposes only.

Font Name Pt:	**9**	**10**	**12**	**14**	**16**	**18**	**20**	**24**	**36**
Athens						23			
Cairo						26			
Chicago			16						
Courier	11	12	12	15		18		23	
Geneva	12	13	16	19		23	24	29	
Helvetica	11	12	14			18		25	
London						23			
Los Angeles			16					29	
Monaco	11		16						
New York	12	12	16	19		23	24	29	40
San Francisco						23			
Times	11	12	12	15		19		24	
Venice				19					

☞ **Note**: Font heights can vary depending on whose fonts you use. With bit-mapped, TrueType and PostScript fonts being available, as well as the proliferation of suppliers, it is possible that one Macintosh's Helvetica 12pt may have a different height than another's.

It's a good habit to rely on the Macintosh's core set of fonts: Chicago 12pt, Geneva 9pt, and Monaco 9pt wherever possible. Always check that a font exists before using it!

25 Special Routines

Use these routines with caution, or don't use them!

The following routines should be used with caution, or not at all. The "comments or special instructions" column explains if the routine can be used in a limited way, or if a Tools Plus equivalent should be used instead. If the column is blank, the Macintosh toolbox routine should not be used at all.

If Inside Macintosh states that a specific routine should not be called because it is automatically called by the system when it starts up, you need not reference that routine in this section.

Some routines that are not listed here *can* be used, but they will require a pointer to a window. Your application can use Tools Plus's WindowPointer routine to determine a window's pointer.

Toolbox Routine	Comments or Special Instructions
AddResMenu	Use AppleMenu
AdvanceKeyboardFocus	
AEProcessAppleEvent	Called automatically by InitToolsPlus
AppendMenu	Use Menu
AutoEmbedControl	Use SetAutoEmbed
BackColor	Use SetBackRGB
BackPat	A window's background pattern should never be changed from white
BeginUpdate	This is done automatically when your application receives a doPreRefresh or doRefresh event. If you want to call BeginUpdate and EndUpdate yourself in response to a doPreRefresh or doRefresh event, add the wManualUpdate option to the window's spec parameter when you create the window.
BringToFront	
CheckItem	Use CheckMenu
ClearKeyboardFocus	
ClearMenuBar	
CloseDialog	Use WindowClose
CloseWindow	Use WindowClose
CouldDialog	
CountMItems	Use MenuItemCount or PopUpItemCount
CreateRootControl	
DeleteMenu	Use RemoveMenu or RemovePopUp
DelMCEntries	
DelMenuItem	Use RemoveMenu or RemovePopUp
DialogSelect	
DisableItem	Use EnableMenu
DispMCInfo	
DisposeControl	Use with custom controls only
DisposeDialog	Use WindowClose
DisposeMenu	
DisposeWindow	
DlgCopy	
DlgCut	
DlgDelete	
DlgPaste	
DragControl	Use with custom controls only
DragGrayRgn	Make sure you call ResetMouseClicks immediately after using DragGrayRgn because DragGrayRgn consumes a mouse-up event at the end of the drag.
DragWindow	
Draw1Control	Use with custom controls only
DrawDialog	

Toolbox Routine	Comments or Special Instructions
DrawGrowIcon	
DrawMenuBar	Use UpdateMenuBar
EmbedControl	Use the appropriate Tools Plus routine to embed the control. See the chapter regarding the control you want to embed, such as “Buttons” (EmbedButtonInButton), “Pop-Up Menus” (EmbedPopUpInButton), etc.
EnableItem	Use EnableMenu or EnablePopUp
EndUpdate	This is done automatically when your application finishes executing your event handler routine after receiving a doPreRefresh or doRefresh event. If you want to call BeginUpdate and EndUpdate yourself in response to a doPreRefresh or doRefresh event, add the wManualUpdate option to the window’s spec parameter when you create the window.
EventAvail	Under MultiFinder and System 7 or higher, this routine will switch tasks to other applications and desk accessories. Don’t use this routine unless you really know what you are doing and you thoroughly test your results.
FindControl	Done automatically for buttons and scroll bars
FindWindow	
ForeColor	Use SetFrontRGB
FlashMenuBar	
FlushEvents	InitToolsPlus flushes all events when Tools Plus is initialized. You may flush mouse-down, mouse-up, key-down, key-up and auto-key events, as well as application-defined events 1 through 4. When flushing mouse-up events, you may be disrupting a drag that is already in progress. If the StillDown routine returns *true*, a drag is in progress, and Tools Plus will continue processing the drag until a mouse-up event is encountered. As a rule, don’t use this routine unless you really know what you are doing and you thoroughly test your results.
FreeDialog	
FrontWindow	This routine is almost useless if your application has a tool bar or floating palettes. See ToolBarNumber, FirstPaletteNumber, FirstStdWindowNumber, WorkWindowNumber and GetWindowInOrder.
GetBackColor	You can use the safer GetBackRGB for Macs with and without Color QuickDraw
GetCTitle	Use with custom controls only
GetCtlMax	Use with custom controls only. For scroll bars, use GetScrollBarMax
GetCtlMin	Use with custom controls only. For scroll bars, use GetScrollBarMin
GetCtlValue	Use with custom controls only. For scroll bars, use GetScrollBarVal. For buttons (check boxes or radio buttons), use ButtonIsSelected
GetDItem	Use GetDialogItemRect to obtain the item’s display rectangle. All other information that is returned by GetDItem is maintained automatically by Tools Plus and is therefore inaccessible.
GetForeColor	You can use the safer GetFrontRGB for Macs with and without Color QuickDraw
GetItem	Use GetMenuString
GetItemCmd	Use GetMenuCmd
GetItemIcon	Use GetMenuIcon or GetPopUpIcon
GetItemMark	Use GetMenuMark or GetPopUpMark
GetIText	Use GetEditString or GetEditHandle to access a field’s edited text. Use GetFieldString or GetFieldHandle to access a field’s saved text.
GetMCEntry	Use GetMenuBarColors, GetMenuColors, or GetMenuItemColors
GetMHandle	Use GetMenuHandleFromMemory. You should never need to get a menu’s handle
GetMenu	Use LoadMenu, Menu or NewPopUp instead
GetMHandle	Use GetMenuHandleFromMemory. You should never need to get a menu’s handle.
GetNewControl	Use LoadButton or LoadScrollBar to create a control using a ‘CNTL’ resource template
GetNewCWindow	Use LoadWindow, WindowOpen, WindowOpenRect, or ToolBarOpen instead
GetNewDialog	Use LoadDialog
GetNewMBar	Use LoadMenuBar
GetNewWindow	Use LoadWindow, WindowOpen, WindowOpenRect, or ToolBarOpen instead
GetNextEvent	Your application will intercept events so that they never reach Tools Plus. Also note that under MultiFinder and System 7 or higher, this routine will switch tasks to other applications and desk accessories. Don’t use this routine unless you really know what you are doing and you thoroughly test your results.

Toolbox Routine	Comments or Special Instructions
GetOSEvent	Your application will intercept events so that they never reach Tools Plus. Also note that under MultiFinder and System 7 or higher, this routine will switch tasks to other applications and desk accessories. As a rule, don't use this routine unless you really know what you are doing and you thoroughly test your results.
GetPenState	You can use the safer GetColorPenState for Macs with and without Color QuickDraw
GrowWindow	
HandleControlClick	
HandleControlKey	
HideControl	Use ButtonDisplay or ScrollBarDisplay instead. Use this routine with custom controls only.
HideWindow	Use WindowDisplay
HiliteControl	Use with custom controls only. For scroll bars, use EnableScrollBar. For buttons, use EnableButton.
HiliteMenu	Use MenuHilite
HiliteWindow	
IdleControls	
InitCursor	Use CursorShape or ResetCursor routines instead
InitDialogs	
InitMenus	Not required in THINK Pascal
InitWindows	
InsertMenu	Use Menu or NewPopUp
InsertResMenu	
InsMenuItem	Use InsertMenuItm
IsControlActive	Use related Tools Plus routine (ButtonIsEnabled, etc.)
IsControlVisible	Use related Tools Plus routine (ButtonIsVisible, etc.)
IsDialogEvent	
KillControls	
List Manager	List Manager routines may be incorporated into your application to handle sophisticated lists. However, the list's scroll bars must be flagged as "custom controls," and your application must handle all events pertaining to the list.
MaxApplZone	Called automatically by InitToolsPlus
MenuEvent	
MenuKey	
MenuSelect	
ModalDialog	If the dialog's procID is dBoxProc, the dialog is automatically modal. To make any other dialog modal, open the modal window using WindowOpen, then attach the dialog list using LoadDialogList.
MoveControl	Use with custom controls only. For buttons, use MoveButton. For scroll bars, use MoveScrollBar.
MovePortTo	The co-ordinates may be changed temporarily, providing they are reset prior to calling any Tools Plus routine
MoveWindow	Use WindowMove
NewCDialog	Use LoadDialog, or WindowOpen combined with LoadDialogList
NewControl	Use with custom controls only. For scroll bars, use NewScrollBar. For buttons, use NewButton. If you create a custom control, see Tools Plus's DeleteControl routine.
NewDialog	Use LoadDialog, or WindowOpen combined with LoadDialogList
NewMenu	Use Menu or NewPopUp
NewWindow	Use WindowOpen, WindowOpenRect, or ToolBarOpen
NMInstall	Use SetNotification and PostNotification
NMRemove	Executed automatically (if required) when your application is activated
OpenDeskAcc	
PenNormal	You can use the safer PenColorNormal for Macs with and without Color QuickDraw
PopUpMenuSelect	
PostEvent	Do not post app4Evt (or osEvt in System 7 or higher) events. They are used by MultiFinder and System 7's (or higher) Finder.
RegisterAppearanceClient	Option in InitToolsPlus
ReverseKeyboardFocus	
RGBBackColor	You can use the safer SetBackRGB for Macs with and without Color QuickDraw

Toolbox Routine	Comments or Special Instructions
RGBForeColor	You can use the safer SetFrontRGB for Macs with and without Color QuickDraw
SelectWindow	Use ActivateWindow
SelIText	Use SetFieldSelection
SendBehind	
SetApplLimit	You may use the simpler Set68KStackSize or ChangeStackSize
SetCCursor	User CursorShape
SetClikLoop	Needed only when creating non-Tools Plus fields. If you must use SetClikLoop when writing a 680x0 application, call it just before using TEClick. Otherwise, a bug in the 680x0 ROMs will use the most recently installed click loop proc, which may be another proc or the proc used by Tools Plus's fields.
SetControlData	If possible, use the appropriate Tools Plus routine
SetControlFontStyle	Use the appropriate Tools Plus routine (SetButtonFontSettings, etc.)
SetControlVisibility	
SetCTitle	Use with custom controls only. For buttons, use ButtonTitle.
SetCtlMax	Use with custom controls only. For scroll bars, use SetScrollBarMax.
SetCtlMin	Use with custom controls only. For scroll bars, use SetScrollBarMin.
SetCtlValue	Use with custom controls only. For scroll bars, use SetScrollBarVal. For buttons (check boxes or radio buttons), use SelectButton.
SetCursor	Use CursorShape or ResetCursor routines instead
SetDAFont	Use SetDialogFontInfo
SetDialogFont	Use SetDialogFontInfo
SetDItem	
SetEventMask	Upon startup, key-up events are disabled. Do not disable the following events: mouseDown, MouseUp, keyDown, autoKey, updateEvt, activateEvt, and app4Evt
SetGDevice	First use GetGDevice to get the current graphic device, then remember to restore the original gDevice before using a Tools Plus routine
SetItem	Use RenameItem or RenamePopUp
SetItemCmd	Use MenuCmd
SetItemIcon	Use MenuIcon or PopUpIcon
SetItemMark	Use MenuMark or PopUpMark
SetItemStyle	Use MenuStyle or PopUpStyle
SetIText	Use PasteIntoField
SetKeyboardFocus	
SetMCEntries	Use SetMenuBarColors, SetMenuColors, or SetMenuItemColors
SetMCInfo	
SetMenuBar	
SetOrigin	The co-ordinates may be changed temporarily, providing they are reset to (0,0) prior to calling any Tools Plus routine
SetResLoad	Must be set to SetResLoad(true) before calling any Tools Plus routine
SetThemeWindowBackground	Use SetBackgroundTheme
SetWinColor	
SetWindowPic	Nothing other than the picture should be put in the window
SetWTitle	Use WindowTitle
SetZone	If your application has multiple heap zones, make sure you reset the current zone to be the application's default heap zone before using any Tools Plus routines.
ShowControl	Use ButtonDisplay or ScrollBarDisplay instead. Use this routine with custom controls only.
ShowHide	Use WindowDisplay
ShowWindow	Use WindowDisplay
SizeControl	Use with custom controls only. For buttons, use SizeButton. For scroll bars, use SizeScrollBar.
SizeWindow	Use WindowSize
SpaceExtra	SpaceExtra must be set to 0, the initial value, when calling any Tools Plus routine
SysBeep	Use Beep (for convenience)
SystemClick	
SystemEdit	
SystemEvent	
SystemMenu	

Toolbox Routine	Comments or Special Instructions
TEActive	Use ActivateField
TEAutoView	
TECalc	
TEClick	Use ClickToFocus
TECopy	
TECut	
TEDeactivate	Use DeactivateField
TEDelete	
TEDispose	Use DeleteField
TEFromScrap	
TEGetText	Use GetEditString
TEIdle	Done when your application finishes executing an event handler routine, or by calling Process1EventWhileBusy
TEInit	
TEInsert	
TEKey	
TENew	Use NewField
TEPaste	
TEPinScroll	
TEScroll	
TESelect	Use SetFieldSelection
TESelView	
TESetClickLoop	Needed only when creating non-Tools Plus fields. If you must use TESetClickLoop when writing a 680x0 application, call it just before using TEClick. Otherwise, a bug in the 680x0 ROMs will use the most recently installed click loop proc, which may be another proc or the proc used by Tools Plus's fields.
TESetJust	
TESetText	
TEToScrap	
TEUpdate	
TrackBox	
TrackControl	Done automatically for buttons and scroll bars
TrackGoAway	
UnloadSeg	Do not unload the segments containing Tools Plus's libraries
UnregisterAppearanceClient	Automatic in DeinitToolsPlus if application was optionally registered by InitToolsPlus
UpdtControl	
WaitNextEvent	Your application will intercept events so that they never reach Tools Plus. Also note that under MultiFinder and System 7 or higher, this routine will switch tasks to other applications and desk accessories. Don't use this routine unless you really know what you are doing and you thoroughly test your results.
ZoomWindow	

26 Completing Your Application

This section details the final steps that you must take to produce a stand-alone, double-clickable application. The requirements for an application using Tools Plus libraries are nearly identical to those of ordinary Macintosh applications, except that the SIZE resource has some very specific needs. It is assumed that by the time you address these items, you have already written and debugged an application within your development environment and that you are ready to make it a stand-alone application.

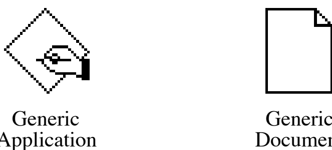
All the work encompassed in this section requires that you use a resource editor such as Apple's ResEdit to create and/or modify resources. The Symantec and Metrowerks development environments allow you to use a resource file as part of your project. That resource file is automatically merged into your compiled code when you build your project. Therefore we recommend you create and save these resources in a separate resource file and allow your development environment to merge them with your application (instead of manually adding them to your compiled application).

Completing your application involves making some decisions, then creating the appropriate resources. The table below summarizes the various tasks and their significance to your final application.

Task (section in this chapter)	**Significance to your application**
Application Icons File Types, Creators, and the Application Signature Bundle (BNDL resource)	Defines the icons that are displayed by the Finder for your application and the files it creates. The bundle resource also defines file types, and optionally folders, that can be opened or processed by your application at the request of an Apple Event.
Version (vers resource)	For displaying a version number in your application's "Get Info" box. An application description can also be displayed. (Optional)
mstr resource	Remapping of the File menu's "Quit" item and the File menu's "Open…" item to another menu and/or menu item. You only need to do this if your application will run on System 6 or older, or if you choose not to make it Apple Event aware. The "Quit" command is automatically invoked when your application is running and the user selects the Special menu's Restart or Shut Down commands. The "Open…" command is automatically invoked when your application is running and the user double-clicks (opens) a document created by your application, or drags a document onto your application to launch it. If your application does not have the "Open…" or "Quit" items in a "File" menu, and your application will [i] run on Macs with System 5 or 6, or [ii] not process high-level events, you need to add mstr resources. (Optional)
'SIZE' resource	Defines your application's behavior and memory requirements in a multitasking environment. Needed for MultiFinder and System 7 or higher.

Application's Icons

The Macintosh's Finder displays an application and its related files as icons. In order for the Finder to do this, it needs to know what each icon looks like. Without this information, your application will have the "generic application" icon and its related files will have the "generic document" icon. These Finder defaults are displayed below:

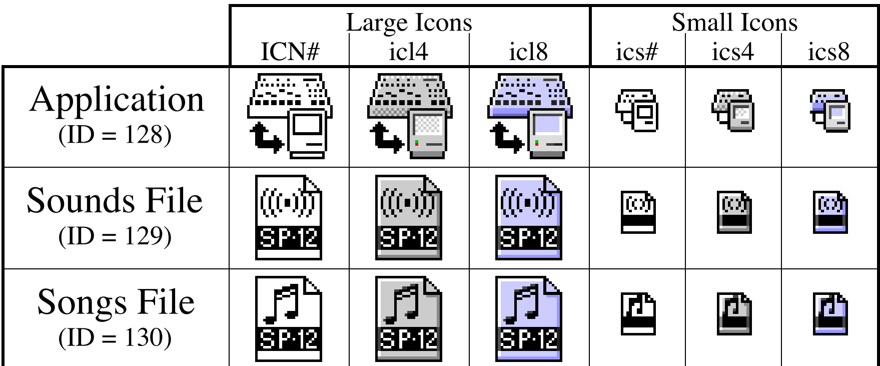


At the very least, you will need to create a large (32 x 32 bit) black and white icon (ICN#) for your application, and one for each type of document it creates. You should also consider creating equivalent small (16x16 bit) black and white icons (ics#), because the Finder displays these miniatures when a disk's or folder's view is set to "small icon," or when System 7 or higher displays lists with icons.

System 7 or higher can display your application's icons in color, so you can optionally include color icons as equivalents for the black and white ones. The icl8 and icl4 icon types are large icons using 8 bits (256 colors) and 4 bits (16 colors) respectively. Small color icons can also be created as ics8 and ics4 type icons. Later, the "bundle" describes how to integrate the icon resources into your finished application.

Icon Family

When a specific icon ID number is shared by two or more icon types, those related icons are called an "icon family." The icon family with an ID of 128 is usually the icon that depicts your application. Your application's document icons would likely be numbered 129 and up. When you create icons, remember to use ID numbers 128 or higher. The rest are reserved numbers. The following table depicts three font families for a completed application:



	Large Icons			Small Icons		
	ICN#	icl4	icl8	ics#	ics4	ics8
Application (ID = 128)						
Sounds File (ID = 129)						
Songs File (ID = 130)						

Note: When creating icl8 and ics8 icons with ResEdit, you have a choice of two color palettes: "Apple icon colors" or "Standard 256 colors." Use the Apple icon colors, since they will guarantee that your icon can be selected (darkened) properly.

File Types, Creators, and the Application Signature

On the Macintosh, all files (including application, desk accessories, documents, etc.) have a *file type* and a *creator* code. Both of these items are always four characters long, allowing any visible or invisible characters and spaces. The file type tells the Macintosh what the file contains, such as plain text ('TEXT' type) or a picture ('PICT' type). Your application will always have a file type of 'APPL.' You can define your own file types for the documents created by your application if they don't fit into any of the existing general types as follows:

APPL	Launchable application	adev	Network extension	pref	Preferences file
DFIL	File for sorting DAs	appe	Background-only application	qery	Query document (databases)
DRVR	Driver	cdev	Control panel	scri	System extension for scripting
FFIL	File for sorting fonts	edtp	Edition for sharing graphics	sfil	Sound
INIT	System extension	edts	Edition for sharing sound	tfil	TrueType font
PICT	QuickDraw picture	edtt	Edition for sharing text	ttro	TeachText read-only file
PRER	Printer driver	ffil	Font	zsys	A system file
RDEV	Chooser extension	ifil	Script system resources		
TEXT	Stream of ASCII characters	kfil	Keyboard layout		

Later, the "bundle" describes how to integrate file type(s) into your finished application.

Signature (the Creator code)

Each application must have a unique, four character *signature*. A signature is often called a *creator code* because it answers the question "who created me" for every document on a Macintosh (an application is considered to be its own creator). Your application's signature will be used as the creator code for all the documents it creates.

The Finder makes an association between each application and its related documents by using your application's signature. When a document is opened by the Finder, it knows which application created the file and automatically launches that application.

Notice that some file types can be opened by several applications. If you double-click a 'TEXT' file, it will be opened by the application that launched it (because the Finder recognizes the creator for the file is identical to the parent application's signature). However, just about any word processor can open a 'TEXT' document, as can the TeachText and SimpleText application that's included with System 7 or later. Your development environment allows you to set your project's "type" and "creator" codes.

Note: Apple reserves the use of all file types and signatures (creator codes) whose names contain only lowercase letters, and those that contain only non-alphabetic characters. Your file types and signatures must contain at least one uppercase letter. Since the system software never displays the file type or signature to users, these codes don't have to be meaningful to anyone but you.

All file types and signatures must be registered with Apple to guarantee uniqueness and prevent conflicts between applications.

Warning: When deciding upon a signature for your application, be careful to avoid signatures that are identical to any existing resource type, such as 'ICON' or 'STR '. An owner resource must be created with a type that matches your application's signature, and if it coincides with existing resources types, conflicts may arise. For a comprehensive list of resource codes, please refer to the ResEdit Reference manual, or contact Apple.

Bundle

The BNDL resource, which is simply called the "bundle," is used to "bundle up" several related resources that are needed to define the association between your application, its documents, and the icons displayed by the Finder. If you use ResEdit to create the BNDL resource, you will be presented with a window that looks like the one below.

Enter the four character signature you have selected for your application. In the example at right, the signature is 'SP12'.

Next, use the Resource menu's "Create New File Type" command to create a new line. Enter the file type. Note that the first line should have a type of 'APPL' (application). Double-click its "Finder Icons" section and you will be presented with a list of icons that you created earlier in the "Application Icons" section of this chapter. Select the icon you want to use to represent the specified file type. Repeat this for each file type that is supported by your application.

In the example at right, the application (type 'APPL') uses icon family 128. The sound file (type 'SPsd') uses icon family 129, and the song file (type 'SPsq') uses icon family 130.

The last thing you must do to complete the bundle is to set the "bundle bit." The bundle bit tells the Finder that your application has a bundle present. In THINK Pascal, you set the bundle bit by using the Project menu's "Set project Type" command, then building the application. The bundle bit is set automatically in THINK C/C++ and Metrowerks compilers.

When creating the BNDL resource, ResEdit automatically creates several other supporting resources. One FREF resource (Finder REFerence) is created for each file type. In the example above, a total of three FREFs would be created. An "Owner Resource" is also created (ID = 0). Its resource type matches your application's signature. In the above example, the type would be 'SP12'. The owner resource is essentially a string resource that lets you enter a comment that is displayed as version information in the Finder's "Get Info" box (providing you don't create 'vers' resources).

Note: If you change your application's icons, you will have to rebuild the desk top file to force the Finder to use your new icons. To rebuild the desk top, restart your Macintosh while holding the option and ⌘ keys down, and continue to hold them until you see a dialog that asks you if you want to rebuild your desk top. Click the OK button to start rebuilding.

A similar problem sometimes occurs when an application is copied to a disk using an installer or decompression program. These kinds of programs may not have the intelligence to update the desk top file like the Macintosh's Finder does.

Version

Your application can display some descriptive information in the Finder's "Get Info" box by having 'vers' resources. The two vers resources you can include in your applications are:

ID = 1 Application version number (i.e., 5, ©1998-99 Water's Edge Software)
ID = 2 Application info displayed beneath the file name (i.e., Tools Plus Library (1 of 7))

By using vers resources, any file can bear version information, including documents created by your application.

mstr Resources

This section describes how System 5 and 6's MultiFinder and System 7 or higher automatically interact with your application's menus if your application does *not* support Apple Events. You should set the appropriate bit in your application's SIZE resource to indicate if it supports Apple Events or not. You can skip this section if your Apple Event aware application runs only on System 7 or later.

Both MultiFinder (running under System 5 or 6) and System 7 or higher can automatically interact with your application through its menus. If your application is running while the user double-clicks (or select-opens) one of your application's documents from the Finder, the affected document is automatically opened by your application. Also, if the user selects the Special menu's Restart or Shut Down command while your application is running, it is instructed to quit.

In both these cases, the system simulates the selection of a menu item. Typically applications have a File menu with an item named "Open…" (including the ellipsis, the Option-; character), and the last item named "Quit". In the case of opening a document, Tools Plus reports a doMenu event to your application indicating that the File menu's "Open…" command was selected, in which case your application would do whatever is appropriate, like display an SFGetFile dialog to let the user choose which file to open. The system fools your application into thinking that the double-clicked file was selected from an SFGetFile dialog (which is not actually displayed). When the user selects Restart or Shut Down, Tools Plus reports a doMenu event to your application indicating that the File menu's "Quit" command was selected, in which case your application would do whatever is appropriate, such as asking if open documents should be saved before quitting.

If your application does not (1) open files by using the File menu's "Open…" command, or (2) quit by using the File menu's "Quit" command, these functions can be remapped to other menu items by including 'mstr' resources in your application. Each mstr resource is a Pascal string (byte-0 is the length byte) that tells the Macintosh which menu and menu item to use in place of the standard ones:

mstr ID = 100	Name of the menu containing the equivalent of the "Quit" command
mstr ID = 101	Name of the menu item that is the equivalent of the "Quit" command
mstr ID = 102	Name of the menu containing the equivalent of the "Open…" command
mstr ID = 103	Name of the menu item that is the equivalent of the "Open…" command

'SIZE' Resource

If your application is going to run under MultiFinder or System 7 or higher, it needs a SIZE resource. If you are using CodeWarrior, Symantec C/C++ or THINK C/C++, you can set these project settings from within your development environment. THINK Pascal users have to use a resource editor like ResEdit to create a SIZE resource in a resource file that is compiled into your application. The SIZE resource settings specify how your application behaves in a multitasking environment.

You will create a single resource with an ID of -1, but be aware that the Macintosh may clone (make a duplicate with a possible minor variation) your SIZE resource in your stand-alone application under certain conditions. The SIZE resource is comprised of sixteen bits that can be set to a value of 0 or 1, and two long integers. Tools Plus requires that some of these items be set to a specific value while others depend upon how you want your application to behave. The table below details the SIZE resource.

SIZE Resource's Field Name	Compulsory Setting	Recommended Setting	Comments
`Save screen (obsolete)`	0	0	(obsolete)
`Accept suspend events`	1	1	0 = doSuspend and doResume events are not reported to your application. 1 = Your application receives a doSuspend event prior to being suspended, and a doResume event immediately after being activated.
	0	0	(obsolete)
`Can background`		1	0 = When your application is suspended, it receives no processing time 1 = When your application is suspended, it receives processing time in the way of doNothing events. If "Accept suspend events" is set to 0, this item must be set to 1.
`Does activate on FG switch`	1	1	0 = The Macintosh's operating system takes care of activating and deactivating a window when your application is suspended or resumed. 1 = Your application takes care of activating or deactivating its windows in response to a doSuspend or doResume event (automatic in Tools Plus). This item must have the same setting as "Accept suspend events."
`Only background`	0	0	0 = Your application is a regular application. 1 = Your application runs only in the background. Usually, this is because it doesn't have a user interface and cannot run in the foreground.
`Get front clicks`		0	0 = Your application does not receive the mouse-down used to activate it after being suspended. Most applications work this way. 1 = The mouse-down that activates your application is applied to your application. This may mean that a button is selected, drawing begins, or an insertion point is set. The Finder works in this manner.
`Accept app died events (debuggers)`		0	0 = ApplicationDied events are not generated 1 = Your application is notified (via an ApplicationDied event) when an application or process launched by your application has terminated or has crashed. Read about System 7's Process Manager for more details.
`32 bit compatible`		1	0 = Your application is not 32-bit compatible 1 = Your application can be run with the 32-bit Memory Manager. Modern Mac compiler's generate 32-bit compatible code, so your app is 32-bit clean providing you do not modify your application's addressing.
`High level event aware`		1	0 = No high-level events are received or sent by your application 1 = Your application can receive and send high-level events (System 7+) If this item is set to 0, MultiFinder and System 7 or higher will automatically interact with your application's menus. See the Menus chapter for details.
`Local and remote high level events`		0	0 = Your application cannot be accessed by applications running on other computers on a network. 1 = Your application is accessible by applications running on other computers on a network, and it can receive high-level events across a network (System 7 or higher).
`Stationery aware`		0	0 = If the user opens a stationary document (System 7 or higher), the Finder makes a copy of the file and asks the user to name the copy. 1 = Your application knows how to work with stationery documents (System 7 or higher).

SIZE Template's Field Name	Compulsory Setting	Recommended Setting	Comments
`Use text edit services`		0	0 = Your application does not use inline TextEdit services 1 = Your application uses inline TextEdit services (System 7 or higher). Tools plus's editing fields don't use inline TextEdit services. Set this item to 1 only if you have designed your own editing fields that use these services.
`Reserved bit`	0	0	(reserved for future use)
`Reserved bit`	0	0	(reserved for future use)
`Reserved bit`	0	0	(reserved for future use)
`Size`			The memory size at which your application can run most efficiently. When your application is launched, the Operating System tries to secure this amount of memory.
`Min size`			The memory limit below which your application will not run.

Cloned SIZE resources

Although you need to create only one SIZE resource (ID =-1), the Macintosh may make clones of it when the user makes changes in the Finder's "Get Info" box. When the user makes changes in the "Get Info" box, the Macintosh clones (makes a copy of the SIZE resource with ID =-1) and lets the user makes changes to the "Size" and "Min size" fields of the clone. You only need to be concerned with this if you want to preset your application's memory requirements to something other than the recommended limits defined by your original SIZE resource.

The table below describes the contents of the "Size" and "Min size" fields for the original SIZE resource, as well as any clones that may be created by the Macintosh.

System File Version	SIZE ID	Field's Name	Title in "Get Info" box	Contents
5 to 7.0.x	-1	Size	Application memory size	Memory required for optimum performance
		Min size	Suggested memory size	Memory limit below which your application won't run
	0	Size	Application memory size	User's setting
		Min size	Suggested memory size	Memory limit below which your application won't run (same as ID =-1)
7.1 or higher	-1	Size	Suggested size	Memory required for optimum performance
		Min size	(not displayed)	Memory limit below which your application won't run
	0	Size	Preferred size	User's preferred memory allowance
		Min size	(not displayed)	Memory limit below which your application won't run (same as ID =-1)
	1	Size	(not displayed)	User's preferred memory allowance (same as ID = 0)
		Min size	Minimum size	User's minimum memory allowance below which your application won't run.

27 Technical Support

Unlimited best-in-class Technical Support is available only to *registered* Tools Plus users. We also provide limited support to developers who are evaluating Tools Plus.

What does Technical Support do?

Water's Edge Software Technical Support provides the following services:

- Referring to the user manual and indicating where the requested information can be found
- Clarifying the user manual where information is not clear or if something is misunderstood
- Explaining how to apply Tools Plus to achieve the desired results
- Communicate information about the latest version of Tools Plus (i.e., version number, new features, etc.)
- Logging your requests to add functionality or to make enhancements to Tools Plus, or its manual

What doesn't Technical Support do?

As the creator and publisher of Tools Plus, Water's Edge Software is prepared to support its *own* product. We cannot, however, support others' products even though you may be using them to develop or maintain your applications. Specifically, our Technical Support staff will not do your homework for you regarding:

- Computer hardware and peripherals
- Macintosh system software
- Your development environment (CodeWarrior, THINK C/C++, ResEdit, etc.) or other applications
- Your programming language (we won't teach you how to program, or how to use C/C++ or Pascal)
- The Macintosh toolbox (Inside Macintosh and THINK Reference are both excellent resources)
- Queries that are beyond the scope of Tools Plus, such as "how do you do animation?" or "how do I include QuickTime movies in my application?" These items are not specific to Tools Plus, and you can implement them in a Tools Plus application in a fashion that is similar, if not identical, to a non-Tools Plus application.

If your query is specific to Tools Plus, you can be sure that we'll be able to help. A special case exists when it comes to plug-ins. The variety of applications that support plug-ins is so vast, and the frequency with which these architectures change their requirements for plug-ins is so rapid, that Water's Edge Software does not have the expertise to tell you how to do something in a plug-in. Even so, we can offer valuable assistance in telling you how Tools Plus behaves, what it expects, and how it responds inside any plug-in environment. This information, when combined with your knowledge of your plug-in environment, will help you quickly create plug-in solutions. Tools Plus has been used to create plug-ins for Adobe Acrobat, Photoshop, and other applications.

Electronic Mail (Email) and Web Support

Why email? Our Technical Support department is best equipped to respond to urgent matters by being able to prioritize all incoming requests then assigning our people to the most urgent ones. Your urgent requests will typically be resolved within hours. Water's Edge Software can be reached at WaterEdg@interlog.com

Include the words "Technical Support" or "Tech Support" (without the quotations) in your email's subject to route your message directly to our technical support staff. Please do not send files without first obtaining authorization from a technical support representative.

Similarly, you can submit reports by web at the following URL:
http://www.interlog.com/~wateredg/_Comments/TechSupport/Form.html

Mail Support

Water's Edge Software also offers a mail-in Technical Support service. We will endeavor to respond within three business days of receiving your letter. Our mailing address is:

Water's Edge Software
Technical Support
2441 Lakeshore Road West, Box 70022
Oakville, Ontario
Canada, L6L 6M9

Fax Support

Water's Edge Software also offers Technical Support by fax. Our fax number is: 1-905-847-1638.

Telephone Support

Due to the technical nature of Tools Plus, we cannot offer unlimited telephone support. Users are encourages to use email or fax as a primary means of support (typically, you'll get an answer within hours). Water's Edge Software Technical Support can be reached at our office number 1-416-219-5628. A menu will direct you to our Technical Support staff.

Notification by Email

If you have an email account at which you can be readily reached, it is always to your advantage to receive notifications and news by email instead of conventional mail. You'll get the most recent news as quickly as possible, as well as being notified of upgrades before the press even hears about them. Just send us an email requesting this free service. Include your full name so we can confirm that you are a registered developer.

Updates and Upgrades by Email

You can receive Tools Plus updates and upgrades electronically as a file enclosure using conventional email. Your email service must be capable of receiving large enclosures as large as 1MB (or 1000K). Some email systems segment incoming mail into smaller chunks (usually less than 32K) and require that the segments be reassembled before the enclosure can be decoded and decompressed. If you are running a "pure" Internet account, it may be as simple as using a different application for your email to prevent segmentation, such as Eudora, a shareware email application that can send and receive very large enclosures. Your email account must also be able to receive volume as high as 20 MB (a set of twenty enclosures, each being 1 MB). Check with your email service provider to ensure that your email account can support this high volume because some email gateways have a limited capacity, and they simply reject all email when they are full.

Our updates are compressed by StuffIt Deluxe 5 (to save downloading time) and encoded using BinHex 4.0 format (to allow transmission on a text-only medium such as Internet), so you will need an application that can decode the BinHexed file and decompress it. An excellent application that does just this is Aladdin System's "StuffIt Expander" which is available as freeware from most electronic bulletin boards. You will need StuffIt Expander version 5 or newer.

To receive updates and upgrades by email, just send us an email requesting this free service. Include your full name so we can confirm that you are a registered developer.

Updates by the web

If you have Internet access and a web browser, you can obtain software updates and upgrades from our web site. Our updates page provides all the details about what you need to do to download and install the latest update. This is an excellent alternative to getting updates by email because this free service lets you obtain the update at your own leisure, and it circumvents problems that may occur if your email gateway has difficulty processing large enclosures.

Just send us an email requesting this free service. Include your full name so we can confirm that you are a registered developer. We'll notify you by email with instructions on how to access the secret updates area of our web site.

Mail updates

If you don't have Internet access for obtaining updates from our web site, or if updates by email are not possible or they are not practical for you, we also offer updates on CD by conventional mail.

Tools Plus Developer Forum

Water's Edge Software offers a free electronic forum where Tools Plus developers can meet, discuss issues, and exchange information. If you would like to talk with these developers or just lurk to see what's happening, send an email to TPDevLst@interlog.com and we'll send you more information about the forum and how to get on it. You don't need any special software other than your email application.

Known Bug List

You can get a copy of all confirmed bugs in the latest version of Tools Plus by sending an email to WaterEdg@interlog.comwith a *subject* of "send Tools Plus bug list" (without the quotes). This is a free, on-demand service for registered developers only, so please do not release this information publicly. Our bug list is updated as soon as new bugs are confirmed. This is typically a very short list.

Bug Alert Service

We can alert you by email as soon as new bugs are discovered in Tools Plus. To subscribe to this free service, send an email to WaterEdg@interlog.comwith a *subject* of "subscribe Tools Plus bugs" (without the quotes). As soon as a new bug is confirmed, you will receive a notification by email. History has shown us that Tools Plus is exceedingly stable and bug free, so you can expect negligible email traffic from this source. Developers with mission-critical or high-profile projects are encouraged to subscribe to this free service. Please do not make this information public as it is available to registered developers only.

Registered Developer Benefits Period

As part of your Tools Plus licensing fee, Water's Edge Software provides the following products and services to you for one full year starting from your initial purchase, at no additional cost:

- Prompt, world-class technical support with no limit to the number of emails/calls
- Software updates (bug fixes and minor revisions)
- Software upgrades (major releases containing considerable new functionality and/or improvements to existing services and features)
- Access to the electronic Tools Plus Developer Forum where you can meet other Tools Plus developers and leverage their expertise and experiences.
- Access to the online Tools Plus Known Bug List (a detailed list of all known bugs confirmed to date, their status, work arounds, and what we are doing about them)

- Subscription to the Tools Plus Bug Alert Service. This service sends you an email as soon as a new bug is discovered and confirmed in Tools Plus libraries + framework. The email details the impact of the bug, work arounds, and what we are doing about it. This service is highly recommended for all developers!
- Subscription to Water's Edge Software's press releases, as well as internal communiqués that are intended *only* for Tools Plus licensees. This service keeps you informed about what we are doing and the projects that are being planned.

Your benefit period starts with your initial Tools Plus purchase, and continues for one full year. Software updates and upgrades include delivery to you at no additional cost. Our goal is to have at least two substantial releases per year. We automatically send you a reminder by email and regular mail when it is time to renew your benefits period for an additional year. The reminder includes complete details about your renewal.

How to Submit Queries or Problem Reports

Should you require the assistance of our Technical Support services, please be prepared by being able to communicate to them what the issue is, and how you want to resolve it. Some of the things you may want to think about *before* you contact us are:

- Have you read the Tools Plus user manual and looked for the answer yourself?
- Does the Tools Plus demo application or one of our tutorials have a working example of what you are trying to do? If so, inspect it to see what you are doing differently.
- What is the nature of your query:
  - Are you trying to implement a feature and don't know how to start?
  - Have you written some code that isn't working the way you expect it to?
  - Are you in need of information that you can't find in the user manual?
  - Is the user manual unclear about something?
  - Are you requesting new features or additional services?
- If you suspect a bug, or Tools Plus isn't working the way you expect it to…
  - Submit only one item at a time and work through that item until it is resolved. We find that if you have several issues on the go, the process slows down considerably as we try to determine which of the issues we're talking about, and whether one issue is related to another.
  - Which Water's Edge Software product and version are you using (i.e., Tools Plus Pro 4.5 for CodeWarrior, C/C++ PowerPC libraries)
  - Which development environment are you using? (Software, version, Macintosh model, system version, amount of memory, etc.) In the case of CodeWarrior, tell us which CD you are using (i.e., CodeWarrior Pro 4) rather than the IDE version number.
  - What is the nature of the problem? What do you want it to do, and what is it actually doing?
  - Does the problem occur in the development environment, in a stand alone application, or both?
  - Does it happen all the time, or is it intermittent?

Remember that you are describing a problem to someone who does not know your history, cannot see your screen, and has not seen your code.

Reporting a Simple Bug

If you can describe a bug you have encountered in a sentence or two, then that implies it should be easy for our Technical Support staff to duplicate it. These kinds of issues are typically simple, easily reproduced, and they occur without other influencing conditions. An example of this is "when I create a certain kind of Bevel Button control, it crashes my Mac as soon as I click it." In a case like this, all you'd need to do is email us a copy of the line of code you used to create the button and the resource(s) that are needed to create the control.

If the bug is a little more complex, consider sending us a report that includes step-by-step instructions on how we can reproduce the problem. For example:

- Problem: Dynamic alert does not behave modally, and active window misbehaves after dynamic alert is closed
- Open a modeless window #1 [provide code used to open window]
- Open a dynamic alert [provide code used to open the alert]
- Click on window #1 (window #1 activates, but all controls are disabled)

BUG: Window #1 should not activate because the dynamic alert should be modal
- Click on the dynamic alert (dynamic alert disables)
- Dismiss the dynamic alert by clicking the OK button

BUG: Window #1 is active, but all controls are still disabled

I'm using Tools Plus Pro 4.5, 68K C/C++ libs for applications, Power Mac 8500, Mac OS 8.5.1, and CodeWarrior Pro 2.

As you can see in this fictitious example, it provides all the information we need to reproduce the problem, and to identify what you see as the problem.

Reporting a More Complex Bug

If you come across a bug that cannot be described in very simple terms, chances are that it will be difficult or time-consuming for us to reproduce. To help us reproduce your problem and track down the cause, please do the following:

1. Create a mini-application that does nothing other than show the problem. Make it *absolutely minimal* to eliminate the possibility of your code being the source of the problem. Remove all code other than the minimum that is needed to demonstrate the problem. Do not send a stripped down version of your application or of our framework because the volume of code means that we would have to validate it all to determine if it is contributing to the issue you are reporting.
2. Email the following to us at wateredg@interlog.com
   - Project file (with binaries removed)
   - Source files
   - Resource files
   - Any other support files required for the mini-app that are not part of the standard development environment.

   Do not send Tools Plus libraries.
3. Include a step-by-step account of *how* we can reproduce your problem. Detail what we will be seeing as a demonstration of the problem. Describe what we *should* see if the problem was to be fixed.
4. Tell us:
   - Which compiler are you using (i.e., CodeWarrior Pro 4, C/C++, 68K)
   - Tell us the computer you are encountering the problem on (i.e., Quadra 840AV)
   - System version (i.e., Mac OS 8.0)
   - Tools Plus versions (i.e., 5)

Why is it important to write a mini-app?

- A mini-app reduces or eliminates the possibility that your code is causing a problem, and it clearly points to Tools Plus. Minimal code leaves no place for bugs to hide. In most cases, when a developer writes a mini-app to demonstrate a problem, it disappears! As they work to reproduce the error in the mini-app, they discover that there was an error in their source code rather than in Tools Plus libraries. If your mini-app's source code is more than a few dozen lines, eliminate more code!
- We have found that we spend over 80% of our technical support time just trying to recreate a user's reported problem, and often we were not be able to do so. We want to make sure that we have exactly the same source code and resources that you do to help us duplicate your problem. If we can't duplicate it, we can't do anything about it.
- Providing a mini-app lets us be most effective in discovering and resolving your reported problems. As an example, we can easily spend half an hour setting up a test case in an attempt to reproduce your reported bug. With a mini-app, we can eliminate that half hour of non-productive time and get to your specific issues within seconds.
- If we can duplicate the problem, we will correct Tools Plus libraries and use the new libraries on the same mini-application to make sure that our change has resolved your problem.

Index

3D Buttons (see Picture Buttons)
4-byte integers (in CodeWarrior C/C++) 60

A

About box 353, 359
Action routine 211, 231, 233
ActivateButton routine 175
ActivateField routine 262
ActivateListBox routine 293
ActivateScrollBar routine 222
ActivateWindow routine 140
Active application 436
Active window 139, 140, 141
ActiveFieldNumber routine 270
ActiveWindowNumber routine 148
AddResMenu routine 547
Address changes 43
Address of Water's Edge Software 562
AdvanceKeyboardFocus routine 547
AEProcessAppleEvent Routine 547
ALERT BOXES 497
  Allow/disallow doNothing events 505
  Changing button titles 503
  Determine number of open alerts 505
  Dynamic alerts 501
  Getting preferences 503
  Macintosh Alerts 539
  Maximum number of open alerts 505
  Setting preferences 504
AlertBox routine 501
AlertBox3 routine 503
AlertBoxCount routine 505
AlertButtonName routine 503
alertPlainBackdrop constant 502
Alerts 109
alphaLock constant 417, 418
altDBoxProc constant 120, 123
altPaletteProc constant 120
AnimateCursor routine 384
Animated cursor (see Cursors)
Animation 522
Appearance Manager 78, 91
  Are Appearance Manager routines available? 531
  Auto-embedding controls 111
  Background theme 103, 112, 122, 136, 137
  Clicking to a new keyboard focus 166, 214, 248, 264, 285
  Compiling 680x0 apps 91
  Compiling PowerPC apps 91
  Containers 92
  Controls 91, 92, 102, 111, 159, 212, 248, 284
    Bevel button 128, 160, 310
    Chasing Arrows 128, 163
    Check box 129, 160
    Clock 128, 162
    Disclosure triangles 128, 162
    Edit Text 128, 249
    Group Box 128, 162
    Icon Control 129, 165
    Image Well 128, 164
    List Box 129, 285
    Little Arrows 128, 163, 213
    Picture Control 129, 164
    Placard 128, 163
    Pop-Up Arrows 128, 164
    Pop-Up menu 128, 312
    Progress indicator 128, 213
    Push button 129, 160
    Radio button 129, 160
    Scroll bar 129
    Slider 128, 213
    Standard scroll bar 213
    Static Text 129, 163, 249
    Tabs 128, 161
    Thermometer 128, 213
    User Pane 128, 165
    Visual Separator 128, 164
    Window Header 129, 165
  Embedding 92
  Embedding a button 172
  Embedding a field 257
  Embedding a list box 290
  Embedding a pop-up menu 316, 317
  Embedding a scroll bar 218
  Embedding controls 171
  Getting the keyboard focus 152
  Is Appearance Manager available? 530
  Is Appearance Manager running? 531
  Keyboard focus 140, 166, 175, 214, 222, 248, 262, 264, 265, 266, 285, 293
  Keyboard focus in toolbar or floating palette 238
  Keyboard focus window 150
  Remove keyboard focus 140
  Substituting a button ProcID 158, 186
  Substituting a scroll bar ProcID 211
  Substituting a window ProcID 118, 154
  Tabbing to a new keyboard focus 166, 214, 248, 266, 285
  Windows 91, 102, 111, 122
Appearance.h header file 91
Appearance.p interface file 91
AppearanceLib library 91, 159, 212, 248, 284
AppendDialogList routine 131
AppendMenu routine 547
APPLE EVENTS 354
  Awareness (making your app Apple Event aware) 426
  Making your application Apple Event aware 558
  Not using them (or pre System 7) 557
  Open Application 426, 464
  Open Documents 426, 465
  Overview 33, 426
  Print Documents 426, 469
  Quit Application 426, 470

Setting an Apple Event error 431
Simulated Apple Events 427, 430
Apple menu () 356, 359
AppleChar constant 326, 327, 373
AppleMenu routine 359
AppleTalk Manager 461
Application heap (consumption) 541
Application Zone limit 98
Application's signature 555
Applications (hiding other apps and Finder) 153
ApplicationSuspended routine 436
Architecture model for Tools Plus 31
arrowCursor constant 383, 385, 386
AttachMenu routine 364
AttachPopUpSubMenu routine 318
AutoEmbedControl routine 547
Automatic cursor changes 379
AutoMoveSize routine 153
AutoMoveSizeButton routine 182
AutoMoveSizeField routine 278
AutoMoveSizeListBox routine 304
AutoMoveSizePanel routine 348
AutoMoveSizePictButton routine 207
AutoMoveSizePopUp routine 331
AutoMoveSizeScrollBar routine 227

B

BackColor routine 547
Backdrop color (setting it) 112, 135, 136
BackdropColor routine 135
Background color (also see backdrop color)
Getting a window's background color 486
Setting a window's background color 486
Background processing 418, 434, 435, 438, 441
Background theme 112, 136, 137
BackPat routine 547
BackSpaceKey constant 448, 458, 460
BALLOON HELP 391
Help for menus 392
Help for objects in windows 392
Help for the Finder 392
Inheritance 392
Issues with THINK Pascal 398
Performance 397
bAutoMoveSize constant 167
bCDEFCheckBox constant 168
bCDEFPushButton constant 168
bCDEFRadioButton constant 168
bCmdKey constant 167
bColorButton constant 167
bDefault constant 167
Beep routine 527
Beeping the user 527
BeepSynch routine 527
BeginUpdate routine 547
BeginUpdateScreen routine 468, 480, 517
Bevel buttons (see Buttons)
bHidden constant 167
Bit depth of a monitor 483
Bit depth of a screen 481
BitMap2Region routine 525
bitmapBW constant 524
bitmapFromCurrentGDevice constant 524
bitmapFromGDevice constant 524
bitmapFromMaxGDevice constant 524
bitmapFromWindow constant 524
BitMaps 522
Converting a BitMap to a region 525
Create a BitMap 523
Destroy a BitMap 524
bold constant 144, 145, 183, 228, 260, 300, 328, 331, 349, 375
BringToFront routine 547
btnState constant 417, 418
Bugs (source of most) 44
Build Order for THINK Pascal projects 72
Bundle bit 556
bUseWFont constant 167
Busy mode 380
ButtonColors routine 173
ButtonDisplay routine 174
ButtonIsEnabled routine 177
ButtonIsSelected routine 177
ButtonIsVisible routine 175
BUTTONS 157
Activating a button 175
Appearance Manager 102, 111
Appearance/behavior specifications 167
Automatic moving/resizing 153, 167, 182
cctb resource 170, 171
Changing a button's co-ordinates 181
Changing a button's size 181
Changing a button's title 180
Changing button titles on dynamic alert boxes 503
Clicking in a button with keyboard focus 264
Clicking with the wrist watch cursor 390
CNTL resource 170, 171
Color 158, 167
Default colors for multiple buttons 173
Getting a button's colors 184
Setting a button's colors 184
Setting colors for multiple buttons 173
Color resources 170, 171
Command key in buttons 167
Creating a button 166, 169, 170
Creating using a 'CNTL' resource 170, 171
Current value
Getting the current value 179
Setting the current value 179
Custom buttons 157, 168
Default button 167
Removing default status 185
Setting the default button 185
Deleting a button 174
Deselecting
Deselecting a button (un-checking) 177
Is a button deselected (un-checked) 177
Determining if a button is visible 175
Disabling
Disabling a button 176
Is a button disabled 177
Enabling
Enabling a button 176
Is a button enabled 177
Flashing a button 180

Font
Getting the font, size and style 183
Setting the font, size and style 158, 183
Font (setting the font) 167
Getting a button's co-ordinates 176
Handle (getting it) 185
Hiding a button 167, 174, 175
Maximum value limit
Getting the maximum limit 178
Setting the maximum limit 179
Minimum value limit
Getting the minimum limit 178
Setting the minimum limit 178
Moving a button 180
New button (creating a) 166, 169, 170
New button (using 'CNTL' resource) 170, 171
Obscuring a button 175
Scrolling buttons 175
Selecting
Is a button selected (checked) 177
Selecting a button (checking) 177
Selecting with Command key 158, 167
Setting Help for a button 399, 400
Showing a button 174
Styles
Bevel button 128, 160
Chasing Arrows 128, 163
Check box 128, 129, 160, 167, 168
Clock 128, 162
Disclosure triangles 128, 162
Edit Text 128
Group Box 128, 162
Icon Control 129, 165
Image Well 128, 164
Little Arrows 163
Picture Control 129, 164
Placard 128, 163
Pop-Up Arrows 128, 164
Push button 128, 129, 160, 167, 168
Radio button 128, 129, 160, 167, 168
Static Text 129, 163, 249
Tabs 128, 161
User Pane 128, 165
Visual Separator 128, 164
Window Header 129, 165
Substituting a ProcID throughout your app 91, 102, 158, 186, 530, 531
Tabbing between buttons with keyboard focus 265, 266
Unused button number (getting the) 173
ButtonTitle routine 180

C

C2PStr routine in C 89
CanAlert constant 502
CanAltBut constant 502
CanOkAlert constant 502
Caps Lock key 409
Caret 236, 241
Cascading menus (see Menus)
cautionIcon constant 502
Change of address notice 43
ChangedCursorZone routine 388
ChangedHelp routine 407
ChangeStackSize routine 106
Character shifting (to upper/lower case) 240, 279
Chasing Arrows control (see Buttons)
Check boxes (see Buttons)
checkBoxProc constant 167
CheckChar constant 326, 327, 373
CheckForMonitorChanges routine 482
CheckItem routine 547
CheckMenu routine 372
CheckPopUp routine 326
Child menu (see Menus)
Class Libraries 36
ClearFocus routine 140
ClearKeyboardFocus routine 547
ClearKeyCode constant 448, 458, 460
ClearListBox routine 299
ClearMenuBar routine 547
ClearPopUp routine 322
ClickToFocus routine 264
Clock control (see Buttons)
CloseDialog routine 547
CloseWindow routine 547
cmdKey constant 417, 418
Co-ordinates
Global 116
Local 116
Screen 116
Code Generators 36
Code modules 93
CODE resources 39
CodeWarrior C/C++ 39
CodeWarrior Pascal 39
COLOR
Color pen state record 491
Color-dependent drawing 477, 480, 481, 482
Determining if Color QuickDraw is available 479
Does a monitor have color? 483
Does a screen have color? 482
Drawing in color 477
Get the pen state 491
Highlight color
Drawing text on the highlight color 489
Drawing with the highlight color 489
Highlighting a rectangle 490
Highlighting a region 491
Number of colors available on a monitor 483
Number of colors available on a screen 481
QuickDraw (using or ignoring Color QuickDraw) 99
Reset pen to initial state 489
Set the pen state 492
Color cursor (see Cursors)
Color QuickDraw (see Color)
COLORS
Settings for dialog items 116
Command key 309, 358, 360, 409
Compiler directives 97
COMPILERS
CodeWarrior C 42
CodeWarrior C++ 42
CodeWarrior Pascal 42
Symantec C++ 42
THINK C 42
THINK Pascal 41, 42

ContAltBut constant 502
Content color (see backdrop color)
Control key 409
Control Panels 39
controlKey constant 417, 418
CONTROLS (see Buttons and Scroll Bars)
    CDEF 159, 212, 248, 284
    Deleting a control that has Help 406
    Setting Help for a control 406
CopyBits bug 190, 515
CouldDialog routine 547
CountIndexString routine 520
CountMItems routine 547
CountNumberOfFiles routine 432
CreateBitMap routine 523
CreateRootControl routine 547
Creator code 555
crossCursor constant 383, 385, 386
Current window 139, 140, 141
CurrentFieldFilter routine 280
CurrentWindow routine 141
CurrentWindowNumber routine 148
CurrentWindowReset routine 141
Cursor Table (see Cursors)
CURSOR TABLES
    Setting Help for a Cursor Table 404
Cursor Zones (see Cursors)
    Setting Help for a Cursor Zone 405
CURSORS 379
    Animation 382
    Application startup cursor 104
    Automatic changes 379
    Changing the cursor 383
    Clicking with the wrist watch cursor 390
    Color cursor 379
    Cursor Tables
        Changing the cursor 387
        Creating a new cursor table 385
        Deleting a cursor table 385
        Get a zone's co-ordinates (rectangle) 388
        Get a zone's co-ordinates (region) 388
        Manually altered cursor zone regions 388
        New cursor table (creating a) 385
        Unused cursor table number (getting the) 385
        Window (attaching to or from a table) 389
    Cursor Zones
        Changing a cursor zone 386
        Changing the cursor 387
        Clicking in a zone 389
        Creating a new cursor zone 386
        Deleting a cursor zone 387
        New cursor zones (creating a) 386
        Unused cursor zone number (getting the) 387
    Force the cursor to animate 384
    Initialization 380
    Reset the cursor to automatic changes 384
    Table (for automatic cursor changes) 381
    Watch cursor ("system busy" indicator) 380, 412
    Wrist watch 104
CursorShape routine 383
CursorZone routine 386
CursorZoneRect routine 386
CursorZoneRgn routine 386
Custom controls 540

D

dBoxProc constant 120, 123
DeactivateField routine 263
DefaultIconLook routine 518
DeinitToolsPlus routine 105
DeleteButton routine 174
DeleteControl routine 406
DeleteCursorTable routine 385
DeleteCursorZone routine 387
DeleteField routine 258
DeleteFwdKey constant 448, 458, 460
DeleteIndexString routine 521
DeleteKey constant 448, 458
DeleteListBox routine 291
DeleteListBoxLine routine 299
DeleteMenu routine 547
DeletePanel routine 344
DeletePictButton routine 199
DeleteScrollBar routine 220
DeleteTimer routine 445
DelMCEntries routine 547
DelMenuItem routine 547
Desk Accessories 39, 88
    Apple menu (creating the) 359
    Menus
        Affected by DAs 357
        Requirements 356
    Using desk accessories 356
Desk top (rebuilding it) 556
DestroyBitMap routine 524
Determine if two structures are equal 532
DfltIconDimBlackLtPat constant 518
DfltIconDimWhiteLtPat constant 518
DfltIconDimWhitePat constant 518
DfltIconLeaveBorder constant 518
Dialog Manager 34, 92
Dialogs (see Windows), 109, 539
DialogSelect routine 547
DiamondChar constant 326, 327, 373
disabled constant 166, 176, 190, 201, 215, 223, 264, 311, 319, 321, 325, 359, 364, 372, 390, 514
DisabledFieldLook routine 271
DisableItem routine 547
Disabling a field 240
Disclosure triangles (see Buttons)
Disk space requirements 40
DispMCInfo routine 547
DisposeControl routine 547
DisposeDialog routine 547
DisposeMenu routine 547
DisposeWindow routine 547
Dividing line in menus 360
Dividing line in pop-up menus 320
DlgCopy routine 547
DlgCut routine 547
DlgDelete routine 547
DlgPaste routine 547
doActivate constant 423
doAutoKey constant 423
doButton constant 423
doChgInField constant 423
doChgWindow constant 423
doClick constant 423

doClickControl constant 423
doClickDesk constant 423
doClickToFocus constant 423
documentProc constant 120, 123
doDeactivate constant 423
doGoAway constant 423
doGrowWindow constant 423
doKeyDown constant 423
doKeyUp constant 423
doListBox constant 423
doManualEvent constant 423
doMenu constant 423
doMoveCursor constant 423
doMoveWindow constant 423
doNothing constant 423
doOpenApplication constant 423
doOpenDocuments constant 423
doPictButton constant 423
doPopUpMenu constant 423
doPreRefresh constant 423
doPrintDocuments constant 423
doQuitApplication constant 423
doRefresh constant 423
doResume constant 423
doScrollBar constant 423
doSuspend constant 423
DotChar constant 326, 327, 373
Double-clickable applications 553
Double-clicking a list box 180, 206
Double-clicking a radio button 180
DownArrowKey constant 448, 458, 460
doZoomWindow constant 423
DragControl routine 547
DragGrayRgn routine 547
DragWindow routine 547
Draw1Control routine 547
DrawDialog routine 547
DrawGrowIcon routine 548
DrawIcon routine 514
Drawing in windows 109
Drawing pictures (PICTs) 509
DrawListBox routine 302
DrawMenuBar routine 548
DrawPict routine 509
DrawPictRect routine 513
DrawShiftPict routine 513
DrawShiftPictRect routine 514
DrawSICN routine 517
DrawSICNmode routine 517
DrawThermometer routine 530
Drivers 39
Drop-down menus (see Pop-Up Menus)
Dynamic alerts (see Alert Boxes)
DynamicFieldHandles routine 271

E

Edit menu 243, 355
Edit Text control (see Editing Fields)
EditFldWindowNumber routine 150, 270
Editing Fields (see Fields)
Email account for Water's Edge Software 561
email updates 562
EmbedButtonInButton routine 172
EmbedButtonInScrollBar routine 172
EmbedControl routine 548
Embedding controls (see appearance Manager)
EmbedFieldInButton routine 257
EmbedFieldInScrollBar routine 257
EmbedListBoxInButton routine 290
EmbedListBoxInScrollBar routine 290
EmbedPopUpInButton routine 316
EmbedPopUpInScrollBar routine 317
EmbedScrollBarInButton routine 218
EmbedScrollBarInScrollBar routine 218
EnableButton routine 176
enabled constant 166, 176, 190, 201, 215, 223, 264, 311, 319, 321, 325, 359, 364, 372, 390, 514
EnableField routine 264
EnableItem routine 548
EnableMenu routine 372
EnablePictButton routine 201
EnablePopUp routine 325
EnableScrollBar routine 223
Enabling a field 240
EndKey constant 448, 458, 460
EndUpdate routine 548
EndUpdateScreen routine 468, 481, 517
EnterKey constant 448, 458, 460
Entity relationship diagram 31
EqualMem routine 532
Error with parameters 36
EscClearKey constant 448, 458, 460
EscKeyCode constant 448, 458, 460
Evaluation Kit 43
Event loop
EVENT MANAGEMENT 409
- Background processing 418, 438, 441
- Clearing the event queue 104
- Creating a UPP 429
- Definition of event dispatching 409
- Definition of event polling 409
- Event Codes 423
- Event filter routine 99, 421
- Event handler for a window 429
- Event handler routine 86, 99, 419
- Event loop (replacement for) 419
- Event modifiers 416
- Event Queue 411
- Event Record 412
- Event Record Fields 415
- Field To Event cross reference 475
- File info for a file that needs to be opened or printed 432, 433
- Filtering events 99, 421
- Getting an event before Tools Plus processes it 99, 421
- Interleave (faster processing) 434
- Key-Up events 85
- Macintosh Event Queue 84
- Macintosh Events 84, 411
- Macintosh toolbox events
  - activateEvt 411, 425
  - app1Evt 411, 426, 461
  - app2Evt 411, 426, 461
  - app3Evt 411, 426, 461
  - app4Evt 411, 426, 461
  - autoKey 411

diskEvt 411, 426, 461
driverEvt 411, 426, 461
keyDown 411, 425
keyUp 411, 425
kHighLevelEvent 411, 426, 461
mouse moved 426
mouseDown 411, 424
mouseUp 411, 424
networkEvt 411, 426, 461
null 418, 434, 441, 463
nullEvent 95, 411, 424
osEvt 411, 426, 461
updateEvt 95, 411, 425, 461
Modifiers 416
Multitasking during lengthy processes 430
Number of files to open or print 432
Overview 33, 409
Periodic tasks 419, 438
Process a single event 430
Process a single toolbox event 431
Process events continuously 430
Queuing 411
Quit process events 433, 434
Recursion 87
Repeating rate for picture buttons 197
Serial events 422
Setting an Apple Event error 431
Timer events 419, 438
Toolbox to Tools Plus event translation 424
Tools Plus Events 85
doActivate 425, 446
doAutoKey 425, 447
doButton 424, 425, 448
doChgInField 424, 425, 449
doChgMonitorSettings 450
doChgWindow 424, 450
doClick 424, 451
doClickControl 424, 454
doClickDesk 424, 454
doClickToFocus 424, 454
doDeactivate 425, 455
doGoAway 424, 456
doGrowWindow 424, 425, 426, 457
doKeyDown 425, 447, 457, 459
doKeyInControl 425, 459
doKeyUp 425, 459
doListBox 424, 425, 460
doManualEvent 95, 425, 426, 461
doMenu 424, 425, 462
doMoveCursor 463
doMoveWindow 424, 425, 426, 463
doNothing 95, 418, 424, 434, 441, 461, 463
doOpenApplication 426, 427, 430, 464
doOpenDocuments 427, 430, 465
doPictButton 424, 466
doPopUpMenu 424, 467
doPreRefresh 424, 425, 467
doPrintDocuments 427, 430, 469
doQuitApplication 427, 430, 470
doRefresh 424, 425, 471
doResume 426, 471
doScrollBar 424, 472
doSuspend 424, 426, 473
doTimer 424, 473
doZoomWindow 424, 473
UPP for event handler routine 429
Watch cursor 412
Window event handler routine 420
Event modifiers 416
EventAvail routine 548
Events
Extensions 39
External code modules 93

F

F1KeyCode constant 448, 458, 460
F2KeyCode constant 448, 458, 460
F3KeyCode constant 448, 458, 460
F4KeyCode constant 448, 458, 460
F5KeyCode constant 448, 458, 460
F6KeyCode constant 448, 458, 460
F7KeyCode constant 448, 458, 460
F8KeyCode constant 448, 458, 460
F9KeyCode constant 448, 458, 460
F10KeyCode constant 448, 458, 460
F11KeyCode constant 448, 458, 460
F12KeyCode constant 448, 458, 460
F13KeyCode constant 448, 458, 460
F14KeyCode constant 448, 458, 460
F15KeyCode constant 448, 458, 460
Fax number for Water's Edge Software 562
Features found in Tools Plus 47
FieldDisplay routine 259
FieldIsEmpty routine 269
FieldIsEnabled routine 264
FieldIsVisible routine 259
FieldLengthLimit routine 270
FIELDS 235
Activating a field 236, 262
Active field 236
Active field number (determining it) 270
Alignment (left, right, centre) 251
Allocating memory for a field's string 235, 250
Appearance/behavior specifications 251
Automatic moving/resizing 153, 252, 278
Box around field 251, 252
Buffering 244
C string (handle pointing to it) 252
Capacity 235
Changing a field's co-ordinates 276, 277
Changing a field's size 277
Clicking in a field 237, 264
Color 102, 239
Getting a field's colors 262
Setting a field's colors 253, 261
Creating a field 250, 255
Creating a field with horiz. scrolling 255, 256, 257
Deactivating a field 263
Dedicated TextEdit record 252
Default spec for edit text items 132
Default spec for editing fields created using a 'CNTL' 132
Default spec for static text fields created using a 'CNTL' 133
Default spec for static text items 132
Deleting a field 258
Determining if a field is visible 259

Dimming on an inactive window 251
Disabled appearance of fields 271, 273
Disabled look
  Beep when clicked 272, 274
  Default settings 272, 274
  Dim not dither box (pre-System 7) 272, 274
  Dim not dither text (pre-System 7) 272, 273
  Never dim color box 272, 273
  Never dim color text 272, 273
  Never dither B&W box 272, 273
  Never dither B&W text 272, 273
Disabling
  Disabling a field 264
  Is a field disabled 264
Disabling a field 240, 252
Dynamic String Handles 235
Dynamic Text Handles (turning on/off) 252, 271
Edit field window number (determining it) 270
Edit text control 251
Edited text 236
  Getting a handle to it 267
  Getting edited text 267
  Getting its length 267
  Saving edited text 269
Editing the text 241
Empty (determining if field is empty) 269
Enabling
  Enabling a field 264
  Is a field enabled 264
Enabling a field 240, 252
Filtering characters 240
Filters
  Applying a filter to a field 252, 280
  Applying a filter to multiple fields 280
  Creating a field filter 279
  New field filter 279
Font
  Getting the font, size and style 261
  Setting the font, size and style 239, 260
Getting a field's co-ordinates 260
Hiding a field 252, 259, 260
Large fields 244
Length limiting 237, 252, 270, 271
Live scrolling 245, 252
Menus
  Affected by editing fields 356
  Affected by fields 243
  Required menus to support fields 355
Minimum memory after "undo" setup 281
Minimum memory during typing 282
Minimum memory for edits 281
Moving a field 276
New field (creating a) 250, 255
New field with horiz. scrolling 255, 256, 257
Obscuring a field 260
Pascal string (handle pointing to it) 252
Pasting into a field
  Handle 275
  Pointer 275
  String 274
Read-only 235
Resetting scrolling to default 252, 279
Return key (allowed or disallowed) 251
Scroll bars 244
Scrolling fields 260
Scrolling several fields on a window 276
Selection Range
  Getting the selection range 263
  Setting the selection range 263
Setting Help for a field 402
Showing a field 259
Single line fields 240, 253
Static Text 129, 235, 251
Static text control 251
String
  Getting a handle to it 268
  Getting its length 269
  Getting the string 268
  Saving edited text 269
String handle 235, 250
Tab in a field 237, 265, 266
TextEdit handle 282
Unused field number (getting the) 258
Using a 'CNTL' resource 249
Using the window's font 251
Vertical scroll bar 252
Word wrap 240
Filter (see Fields)
FindControl routine 548
FindCursorZone routine 389
Finder
  Hiding the Finder and other apps 153
  Programming for Finder in System 5 and 6 88
Finder icons 554
FinderDisplay routine 153
FindWindow routine 548
FirstPaletteNumber routine 149
FirstStdWindowNumber routine 149
FirstWindowNumber routine 148
FKey constant 448, 458, 460
FlashButton routine 180
FlashMenuBar routine 548
FlashPictButton routine 206
Flickering screen (preventing strobing) 528
Floating Palettes (see Windows)
FlushEvents routine 548
FocusWindowNumber routine 150
Font Manager 109
FONTS
  Buttons
    Getting the font, size and style 183
    Setting the font, size and style 158, 183
  Buttons (setting the font) 167
  Fields
    Getting the font, size and style 261
    Setting the font, size and style 239, 260
  Height in pixels 545
  List boxes
    Getting the font, size and style 300
    Setting the font, size and style 284, 300
  Panels
    Getting the font, size and style 349
    Setting the font, size and style 335, 349
  Panels (setting the font) 337
  Pop-up menus
    Getting the font, size and style 332
    Setting the font, size and style 308, 331
  Pop-up menus (setting the font) 312

ROM resident 294, 320, 361
Scroll bars
Getting the font, size and style 228
Setting the font, size and style 210, 228
Settings for dialog items 116
ForeColor routine 548
Foreground color
Getting a window's foreground color 485
Setting a window's foreground color 486
FreeDialog routine 548
Freeze (application freezes) 411, 420
FrontWindow routine 548
Functions (see Tools Plus Routines)

G

Generic Application icon 554
Generic Document icon 554
GetAlertBoxPrefs routine 503
GetBackColor routine 548
GetBackRGB routine 486
GetButtonColors routine 184
GetButtonFontSettings routine 183
GetButtonHandle routine 185
GetButtonMax routine 178
GetButtonMin routine 178
GetButtonRect routine 176
GetButtonVal routine 179
GetColorPenState routine 491, 492
GetCTitle routine 548
GetCtlMax routine 548
GetCtlMin routine 548
GetCtlValue routine 548
GetCurrentCursorZone routine 389
GetCursorZone routine 388
GetCursorZoneRgn routine 388
GetCustomPanelColors routine 344
GetDialogFontInfo routine 144
GetDialogItemRect routine 144
GetDimColor routine 488
GetDItem routine 548
GetEditHandle routine 267
GetEditLength routine 267
GetEditString routine 267
GetFieldColors routine 262
GetFieldFontSettings routine 261
GetFieldHandle routine 268
GetFieldLength routine 269
GetFieldRect routine 260
GetFieldSelection routine 263
GetFieldString routine 268
GetFocusInfo routine 152
GetForeColor routine 548
GetFreeButtonNum routine 173
GetFreeCursorTableNum routine 385
GetFreeCursorZoneNum routine 387
GetFreeFieldNum routine 258
GetFreeHMenuNum routine 363
GetFreeListBoxNum routine 291
GetFreeMenuNum routine 363
GetFreePanelNum routine 341
GetFreePictButtonNum routine 198
GetFreePopUpNum routine 317
GetFreeScrollBarNum routine 219
GetFreeWindowNum routine 135
GetFrontRGB routine 485
GetIndexFile routine 432
GetIndexFileFSS routine 433
GetIndexString routine 520
GetItem routine 548
GetItemCmd routine 548
GetItemIcon routine 548
GetItemMark routine 548
GetIText routine 548
GetListBoxColors routine 301
GetListBoxFontSettings routine 300
GetListBoxHandle routine 305
GetListBoxLine routine 297
GetListBoxLines routine 298
GetListBoxRect routine 293
GetListBoxText routine 296
GetMCEntry routine 548
GetMenu routine 548
GetMenuBarColors routine 368
GetMenuCmd routine 374
GetMenuColors routine 369
GetMenuHandleFromMemory routine 377
GetMenuIcon routine 375
GetMenuItemColors routine 370
GetMenuMark routine 373
GetMenuString routine 371
GetMHandle routine 548
GetNewControl routine 548
GetNewCWindow routine 548
GetNewDialog routine 548
GetNewMBar routine 548
GetNewWindow routine 548
GetNextEvent function 83
GetNextEvent routine 548
GetOSEvent routine 549
GetPanelColors routine 350
GetPanelFontSettings routine 349
GetPanelRect routine 346
GetParentMenu routine 376
GetPenState routine 491, 549
GetPictButtonMax routine 203
GetPictButtonMin routine 202
GetPictButtonRect routine 200
GetPictButtonVal routine 203
GetPopUpColors routine 333
GetPopUpFontSettings routine 332
GetPopUpHandle routine 334
GetPopUpIcon routine 328
GetPopUpItemColors routine 334
GetPopUpMark routine 327
GetPopUpRect routine 323
GetPopUpSelection routine 329
GetPopUpString routine 324
GetScrollBarActionInfo routine 233
GetScrollBarColors routine 229
GetScrollBarFontSettings routine 228
GetScrollBarHandle routine 233
GetScrollBarMax routine 224
GetScrollBarMin routine 223
GetScrollBarRect routine 222
GetScrollBarVal routine 224
GetStandardPanelColors routine 342

GetSubMenu routine 377
GetTEHandle routine 282
GetToolsPlusVersion routine 526
GetTPSerialEvent routine 437
GetWindowInOrder routine 147
GetWindowZoom routine 143
Global co-ordinates 116
Global variables 83
GoAway constant 119
Graphic User Interface 43
Graphics Device handle to a monitor 484
Group Box control (see Buttons)
Group Boxes (see Panels), (see Buttons)
Grow box (in a window) 122
GrowWindow routine 549
GUI 43

H

Handle blocks 543
HandleControlClick routine 549
HandleControlKey routine 549
Hang (application hangs) 411, 420
HasAppearanceManager routine 530
HasAppearanceManagerRoutines routine 531
HasColorQuickDraw routine 479
HaveTabInFocus routine 265, 447, 457
hdlg resource 392, 393, 397
HEAP 98
    Default memory for application 558
    Size 98, 105, 106
Help (see Balloon Help), 561
    Methods for displaying balloons
        Regular window 399
        Save image behind balloon 399
        Save image behind balloon + refresh 399
    Searching Help resources
        All 400
        All resources, all states 400
        Dialog item list (‘hdlg’) 400
        Menu (‘hmnu’) 400
        Rectangles (‘hrct’) 400
    Searching Help resources’ states
        All resources, all states 400
        All states’ Help data 400
        Disabled item’s Help data 400
        Enabled and checked item’s Help data 400
        Enabled item’s Help data 400
        Enabled, other item’s Help data 400
    Setting help using a Help resource
        Setting Help for a button 400
        Setting Help for a control 406
        Setting Help for a Cursor Table 404
        Setting Help for a Cursor Zone 405
        Setting Help for a field 402
        Setting Help for a list box 403
        Setting Help for a panel 404
        Setting Help for a picture button 401
        Setting Help for a pop-up menu 403
        Setting Help for a scroll bar 402
    Setting help without using a Help resource
        Setting Help for a button 399
        Setting Help for a control 406
        Setting Help for a Cursor Table 404
        Setting Help for a Cursor Zone 405
        Setting Help for a field 402
        Setting Help for a list box 402
        Setting Help for a panel 404
        Setting Help for a picture button 401
        Setting Help for a pop-up menu 403
        Setting Help for a scroll bar 401
    User interface object state
        Disabled 399
        Enabled 399
        Enabled and checked 399
        Enabled, other 399
Help Manager (see Balloon Help)
helpAllStates constant 400
helpDisabledState constant 400
helpEnabledCheckedState constant 400
helpEnabledOtherState constant 400
helpEnabledState constant 400
HelpKey constant 448, 458, 460
helpUseAllHelp constant 400
helpUseAllRsrc constant 400
helpUseHdlgRsrc constant 400
helpUseHmnuRsrc constant 400
helpUseHrctRsrc constant 400
hfdr resource 392
HideControl routine 549
HideWindow routine 549
Hierarchical menus (see Menus)
Hierarchy triangles (see Buttons, disclosure triangles)
High-level events (see Apple Events), 354
HiliteControl routine 549
HiliteMenu routine 549
HiliteRect routine 490
HiliteRgn routine 491
HiliteWindow routine 549
hmnu resource 392, 397
HomeKey constant 448, 458, 460
hrct resource 392, 393, 397

I

{$I-} compiler directive 97
iBeamCursor constant 383, 385, 386
Icon Buttons (see Picture Buttons)
Icon control (see Buttons)
Icon family 190, 498, 515, 554
ICONS
    Creating your own 516
    Disabling (graying out) 514, 516, 518
    Displayed by the Finder 554
    Displaying icons 514, 517
    Displaying SICN elements 517
    Large (see icl8, icl4, cicn, ICON and ICN# under Resources)
    Menus displaying icons 360
    Not displaying correctly 556
    Pop-up menus displaying icons 320
    Resource IDs for picture buttons 190
    Selecting (darkening) 514, 516, 518
    Small (see ics8, ics4, SICN and ics# under Resources)
    Small icons (see ics#, ics4 and ics8 under Resources)
    Using icons as buttons (see Picture Buttons)

IdleControls routine 549
IgnoreFirstMouseClick routine 436
Image Well (see Buttons)
inButton constant 449
inCheckBox constant 449
inClick1 constant 453
inClick1Drag constant 453
inClick2 constant 453
inClick2Drag constant 453
inClick3 constant 453
inClick3Drag constant 453
include statement (C) 83
Indexed strings (see Strings)
inDownButton constant 233, 449, 472
Infinity Windoid 155
initAllWindowsHaveBackgroundTheme constant 103
initAppearanceManagerSavvy constant 102
initAutoFocusChanges 237
initAutoFocusChanges constant 102
initAutoSaveFieldString constant 102
InitCursor routine 383, 549
InitDialogs routine 97, 549
initDontUnloadDeskScrap constant 101
initFasterWinDrag constant 101
InitFonts routine 97
InitGraf routine 97
INITIALIZATION
  Application 97
  Automatic 97, 101
  Tools Plus 97
Initialize a record (set to zero) 532
Initializing a program (deinitializing) 105
Initializing an application or plug-in 99
initIgnoreColor constant 101
initIgnoreTEScrap constant 101
initInheritHelp constant 101, 392
initLiveWindowDrag constant 103
initLiveWindowDrag040 constant 103
initLiveWindowDragPPC constant 103
initMacToolbox constant 101
InitMenus routine 97, 549
initPureAppearanceManager constant 102
initReleaseResources constant 101
INITs 39
initTE32KBuffer constant 100
initTEStr255Buffer constant 100
InitToolsPlus function 97
InitToolsPlus routine 99, 515
initUnloadDeskScrap constant 101
initUseColor constant 100
initUseTEScrap constant 101
InitWindows routine 97, 549
inPageDown constant 233, 472
inPageUp constant 233, 472
InsertIndexString routine 521
Insertion point 236
InsertListBoxLine routine 298
InsertMenu routine 549
InsertMenuItm routine 364
InsertPopUpItem routine 321
InsertResMenu routine 549
InsMenuItem routine 549
Integers (4-byte vs 2-byte in CodeWarrior C/C++) 60
Internet 561
inThumb constant 233, 472
inUpButton constant 233, 449, 472
IsControlActive routine 549
IsControlVisible routine 549
IsDialogEvent routine 549
italic constant 144, 145, 183, 228, 260, 300, 328, 331, 349, 375

J

Jerky behavior 411, 420

K

kControlBehaviorMultiValueMenu constant 310
kControlBehaviorOffsetContents constant 161, 310
kControlBehaviorPushbutton constant 161
kControlBehaviorSticky constant 161
kControlBehaviorToggles constant 161
kControlBevelButtonLargeBevelProc constant 161, 310
kControlBevelButtonMenuOnRight constant 310
kControlBevelButtonNormalBevelProc constant 161, 310
kControlBevelButtonSmallBevelProc constant 161, 310
kControlChasingArrowsProc constant 163
kControlCheckBoxMixedValue constant 160
kControlCheckBoxProc constant 160
kControlClockDateProc constant 162
kControlClockIsDisplayOnly constant 162
kControlClockIsLive constant 162
kControlClockMonthYearProc constant 162
kControlClockNoFlags constant 162
kControlClockTimeProc constant 162
kControlClockTimeSecondsProc constant 162
kControlContentCIconRes constant 161, 164, 310
kControlContentIconRef constant 161, 164, 310
kControlContentIconSuiteRes constant 161, 164, 310
kControlContentPictRes constant 161, 164, 310
kControlContentTextOnly constant 161, 164, 310
kControlEditTextProc constant 249
kControlGroupBoxCheckBoxProc constant 162
kControlGroupBoxPopupButtonProc constant 162
kControlGroupBoxSecondaryCheckBoxProc constant 162
kControlGroupBoxSecondaryPopupButtonProc constant 162
kControlGroupBoxSecondaryTextTitleProc constant 162
kControlGroupBoxTextTitleProc constant 162
kControlIconNoTrackProc constant 165
kControlIconProc constant 165
kControlIconSuiteNoTrackProc 165
kControlIconSuiteProc constant 165
kControlImageWellProc constant 164
kControlListBoxProc constant 285
kControlLittleArrowsProc constant 163, 213
kControlPictureNoTrackProc constant 164
kControlPictureProc constant 164
kControlPlacardProc constant 163
kControlPopupArrowEastProc constant 164
kControlPopupArrowNorthProc constant 164
kControlPopupArrowSmallEastProc constant 164
kControlPopupArrowSmallNorthProc constant 164
kControlPopupArrowSmallSouthProc constant 164
kControlPopupArrowSmallWestProc constant 164
kControlPopupArrowSouthProc constant 164

kControlPopupArrowWestProc constant 164
kControlProgressBarProc constant 213
kControlPushButLeftIconProc constant 160
kControlPushButRightIconProc constant 160
kControlPushButtonProc constant 160
kControlRadioButtonProc constant 160
kControlScrollBarLiveProc constant 213
kControlScrollBarProc constant 213
kControlSeparatorLineProc constant 164
kControlSliderHasTickMarks constant 213
kControlSliderLiveFeedback constant 213
kControlSliderNonDirectional constant 213
kControlSliderProc constant 213
kControlSliderReverseDirection constant 213
kControlStaticTextProc constant 163, 249
kControlTabLargeProc constant 161
kControlTabSmallProc constant 161
kControlTriangleAutoToggleProc constant 162
kControlTriangleLeftFacingAutoToggleProc constant 162
kControlTriangleLeftFacingProc constant 162
kControlTriangleProc constant 162
kControlUserPaneProc constant 165
kControlWindowHeaderProc constant 165
kControlWindowListViewHeaderProc constant 165
Keyboard equivalents for menus (see Command key)
KEYS
  * 243
  + 243
  / 243
  = 243
  Backspace key (see Delete key)
  Caps Lock 416
  Clear 242, 244, 447, 457, 459
  Command 158, 167, 241, 309, 358, 360, 416, 498
  Control 416
  Del (delete forward) 447, 457, 459
  Delete 241
  Delete Forward 242
  Down arrow 243, 447, 457, 459
  End 242, 447, 457, 459
  Enter 241, 447, 457, 459
  Esc 447, 457, 459
  Escape 158, 167, 498
  F1 through F15 447, 457, 459
  Help 447, 457, 459
  Home 242, 447, 457, 459
  Left arrow 242, 447, 457, 459
  Option 416
  Page Down 242, 447, 457, 459
  Page Up 242, 447, 457, 459
  Return 241, 244, 447, 457, 459
  Right arrow 242, 447, 457, 459
  Shift 416
  Tab 237, 241, 265, 266, 447, 457, 459
  Up arrow 243, 447, 457, 459
kHMCheckedItem constant 399
kHMDisabledItem constant 399
kHMEnabledItem constant 399
kHMOtherItem constant 399
kHMRegularWindow constant 399
kHMSaveBitsNoWindow constant 399
kHMSaveBitsWindow constant 399
KillButton routine 174
KillControls routine 549
KillField routine 259
KillListBox routine 291
KillPanel routine 345
KillPictButton routine 199
KillPopUp routine 323
KillScrollBar routine 220
KillTPSerialEvent routine 437
kThemeActiveAlertBackgroundBrush constant 136
kThemeActiveDialogBackgroundBrush constant 136
kThemeActiveModelessDialogBackgroundBrush constant 136
kThemeActiveUtilityWindowBackgroundBrush constant 136
kThemeChasingArrowsBrush constant 136
kThemeDocumentWindowBackgroundBrush constant 136
kThemeDragHiliteBrush constant 136
kThemeFinderWindowBackgroundBrush constant 136
kThemeIconLabelBackgroundBrush constant 136
kThemeInactiveAlertBackgroundBrush constant 136
kThemeInactiveDialogBackgroundBrush constant 136
kThemeInactiveModelessDialogBackgroundBrush constant 136
kThemeInactiveUtilityWindowBackgroundBrush constant 136
kThemeListViewBackgroundBrush constant 136
kThemeListViewSeparatorBrush constant 136
kThemeListViewSortColumnBackgroundBrush constant 136
kWindowAlertProc constant 123
kWindowDocumentProc constant 123
kWindowFloatFullZoomGrowProc constant 123
kWindowFloatFullZoomProc constant 123
kWindowFloatGrowProc constant 123
kWindowFloatHorizZoomGrowProc constant 123
kWindowFloatHorizZoomProc constant 123
kWindowFloatProc constant 123
kWindowFloatSideFullZoomGrowProcID constant 123
kWindowFloatSideFullZoomProcID constant 123
kWindowFloatSideGrowProcID constant 123
kWindowFloatSideHorizZoomGrowProcID constant 123
kWindowFloatSideHorizZoomProcID constant 123
kWindowFloatSideProcID constant 123
kWindowFloatSideVertZoomGrowProcID constant 123
kWindowFloatSideVertZoomProcID constant 123
kWindowFloatVertZoomGrowProc constant 123
kWindowFloatVertZoomProc constant 123
kWindowFullZoomDocumentProc constant 123
kWindowFullZoomGrowDocumentProc constant 123
kWindowGrowDocumentProc constant 123
kWindowHorizZoomDocumentProc constant 123
kWindowHorizZoomGrowDocumentProc constant 123
kWindowModalDialogProc constant 123
kWindowMovableAlertProc constant 123
kWindowMovableModalDialogProc constant 123
kWindowPlainDialogProc constant 123
kWindowShadowDialogProc constant 123
kWindowVertZoomDocumentProc constant 123
kWindowVertZoomGrowDocumentProc constant 123

L

Languages (supporting others) 535
Layers (for windows) 115
LeftArrowKey constant 448, 458, 460

Length limiting of fields 237, 270, 271
Lengthy processes (displaying the watch cursor) 380, 412
lExtendDrag constant 287
Lines (drawing zoom lines) 528
Lisa 39
List Box control (see List Boxes), 285
LIST BOXES 283
  Activating a list box 293
  Appearance/behavior specifications 287
  Automatic moving/resizing 153, 288, 304
  Changing a list box's co-ordinates 303
  Changing a list box's size 303
  Changing co-ordinates 303
  Clicking in a list box 264
  Color
    Getting a list box's colors 301
    Setting a list box's colors 301
  Creating a list box 286, 289
  Custom List Boxes 540
  Default spec for list boxes created using a 'CNTL' 133
  Deleting a list box 291
  Determining if a list box is visible 292
  Dimming an inactive list 288
  Disabling
    Is a list box disabled 299
  Enabling
    Is a list box enabled 299
  Font
    Getting the font, size and style 300
    Setting the font, size and style 284, 300
  Getting a list box's co-ordinates 293
  Handle (getting it) 305
  Hiding a list box 288, 292
  Lines
    Adding a set of strings ('STR ')
    Deleting a line 299
    Deleting all lines 299
    Drawing (turn on/off) 302
    Find text 296
    Getting text 296
    Inserting a blank line 298
    Inserting resource names 295
    Selecting
      Is a line selected 297
      Searching for selected lines 298
      Selecting/deselecting a line 297
    Set text 294
  List box control 285, 288
  Moving a list box 302
  New list box (creating a) 286, 289
  No box around list 288
  Number of lines in a list box 301
  Obscuring a list box 292
  Resource names 295
  Scrolling list boxes 292
  Selection Methods
    1 line 287, 288
    Default 287, 288
    Extend drag 287, 288
    First line sensing 287, 288
    No disjoints 287, 288
    No extensions 287, 288
  Setting Help for a list box 402, 403
  Showing a list box 292
  Tabbing between list boxes 265, 266
  Text (see Lines)
  Unused List box number (getting the) 291
  Using a 'CNTL' resource 129, 285
  Using color 288
  Using the window's font 288
List Manager 540
List Manager replacement 46
listAutoMoveSize constant 288
ListBoxDisplay routine 292
ListBoxIsEnabled routine 299
ListBoxIsVisible routine 292
ListBoxLineCount routine 301
listColorList constant 284, 288
listDefault constant 288
listDimWhenInactive constant 288
listExtendDrag constant 288
listHidden constant 288
listNoDisjoint constant 288
listNoExtend constant 288
listNoFrame constant 288
listOnlyOne constant 288
listSystemBody constant 285, 288
listUseSense constant 288
listUseWFont constant 288
Little Arrows control (see Buttons, Scroll Bars)
lNoDisjoint constant 287
lNoExtend constant 287
LoadButton routine 170
LoadDialog routine 127
LoadDialogList routine 131
LoadDialogPopUp routine 316
LoadMenu routine 361
LoadMenuBar routine 362
LoadPopUp routine 315
LoadPopUpRect routine 316
LoadScrollBar routine 217
LoadSpecButton routine 171
LoadSpecDialog routine 130
LoadSpecDialogBehind routine 131
LoadSpecScrollBar routine 218
LoadSpecWindow routine 126
LoadSpecWindowBehind routine 127
LoadWindow routine 125
Local co-ordinates 116
Logical screens 478, 480, 481, 482
Long waits (displaying the watch cursor) 380, 412
lOnlyOne constant 287
Lower case (shifting typed letters to) 240, 279
lUseSense constant 287

M

µRuntime.Lib 73
Mac OS 8 91, 115, 354, 410, 557, 558
Mac OS 8 (programming for) 88
Macintosh 128K 39
Macintosh 512K 39
Macintosh 512KE 39
Macintosh XL 39
mail updates 563
MainMonitorNumber routine 484

mAppleMenu constant 361, 369, 370, 371, 372, 376, 377
mApplicationsMenu constant 372, 376, 377
Max routine 533
MaxAlertBoxes constant 505
MaxApplZone routine 97, 104, 549
Maximum value of two numbers 533
Memory (application's) 558
Memory consumption 541
Memory requirements 40
Menu routine 359
MenuBarDisplay routine 367
MenuCmd routine 374
MenuEvent routine 549
MenuHilite routine 377
MenuIcon routine 374
MenuItemCount routine 376
MenuKey routine 549
MenuMark routine 373
MENUS 353
  Affected by desk accessories 357
  Affected by editing fields 356
  Apple menu (creating the) 359
  Applications menu 357
  Attaching a hierarchical menu 364
  Check mark (displaying, hiding) 360
  Color 102, 354
  Color resources 361
  Command key
    Getting an item's Command-key equivalent 374
    Setting an item's Command-key equivalent 374
  Command key equivalents 358, 359
  Creating a menu item 359
  Creating a pull-down or hierarchical menu 359
  Deleting a menu or item 366
  Deleting all menus 366
  Detaching a hierarchical menu 364
  Disabling menus/items 359, 372
  Displaying icons in menus 359
  Edit menu 355
  Edit menu (Select All) 356, 363
  Enabling menus/items 372
  Fields (effect on menus) 243
  Getting a menu item's colors 370
  Getting a menu's colors 369
  Getting an item's text 371
  Getting default menu colors 368
  Handle (getting it) 377
  Help menu 357
  Hiding the menu bar 367
  Hierarchical 309, 353, 359
  Highlighting a menu in the menu bar 377
  Icon
    Getting a menu item's icon 375
    Setting a menu item's icon 374
  Icons in menus 360
  Insert a menu item 364
  Insert resource names 365
  Mac OS 8 354, 557
  Mark
    Check mark (displaying, hiding) 372
    Display/clear a menu item's "mark" 373
    Getting a menu item's "mark" 373
  Marking menu items (i.e., check mark) 359
  MBAR resource 353, 362
  mctb resource 361
  MENU resource 353, 361, 362
  MultiFinder 354, 557
  Number of items in a menu 376
  Open… 354, 557
  Parent Menu
    Attaching/detaching a hierarchical menu 364
    Getting a menu's parent menu 376
  Pop-down menus (see Pop-Up Menus)
  Pop-up menus (see Pop-Up Menus)
  Quit 354, 557
  Selection
    Deselecting an item 372, 373
    Is an item selected 373
    Selecting an item 372, 373
  Setting a menu item's colors 370
  Setting a menu's colors 369
  Setting default menu colors 368
  Showing the menu bar 367
  Special characters (displaying, hiding) 360
  Styles (setting a menu item's style) 359, 360, 375
  SubMenu
    Attaching/detaching a hierarchical menu 364
    Getting a menu item's submenu 377
  System 7 354, 557
  Title changes 360, 371
  Unused hierarchical menu number (getting the) 363
  Unused menu number (getting the) 363
  Updating the menu bar 367
MenuSelect routine 549
MenuStyle routine 375
Metacharacters 319, 359
mHelpMenu constant 357, 359, 364, 365, 366, 369, 370, 371, 372, 373, 374, 375, 376, 377
Min routine 533
Minimum value of two numbers 533
Modal constant 119
Modal dialogs 92
ModalDialog routine 549
Modeless dialogs 92
Modifier flags 416
Monitor
  Does a monitor have color? 482, 483
  Drawing across multiple monitors 477
  Main monitor number 484
  Number of colors or grays 481, 483
  Number of differently set monitors 480
  Number of monitors 482
  Recalculating monitor and screen settings 482
MonitorDepth routine 483
MonitorGDevice routine 484
MonitorHasColors routine 483
Monitors 483, 484
MoreMasters routine 97, 99
MOUSE
  Button 416
  Clicks
    Detection 451
    Discontinuing multiple clicks 435
    Fields (click in) 264
    Ignoring first of a multiple click sequence 436
    Responding to 451
    Waiting for next mouse-down/up 435
    Zone (click in cursor zone) 381, 389

Dragging
Detection 451
Discontinuing a drag 435
Ignoring first click of the drag 436
Responding to 451
Waiting for mouse-up 435
movableBoxProc constant 120, 123
MoveButton routine 180
MoveControl routine 549
MoveField routine 276
MoveListBox routine 302
MovePanel routine 346
MovePictButton routine 206
MovePopUp routine 329
MovePortTo routine 549
MoveScrollBar routine 225
MoveSizeButton routine 181
MoveSizeButtonRect routine 182
MoveSizeField routine 277
MoveSizeFieldRect routine 278
MoveSizeListBox routine 303
MoveSizeListBoxRect routine 304
MoveSizePanel routine 347
MoveSizePanelRect routine 348
MoveSizePopUp routine 330
MoveSizePopUpRect routine 330
MoveSizeScrollBar routine 226
MoveSizeScrollBarRect routine 227
MoveWindow routine 549
MultiFinder 88, 115, 354, 410, 557, 558
Programming for MultiFinder 88
Multiple monitors 477
Multitasking 88, 410, 558

N

Networks 461
NewButton routine 166
NewButtonControl routine 169
NewButtonControlRect routine 169
NewButtonRect routine 169
NewCDialog routine 549
NewControl routine 549
NewCursorTable routine 385
NewDialog routine 549
NewDialogButton routine 170
NewDialogButtonControl routine 170
NewDialogField routine 255
NewDialogListBox routine 289
NewDialogPanel routine 341
NewDialogPictButton routine 198
NewDialogScrollBar routine 217
NewDialogWideField routine 257
NewEventHandlerProc routine 429
NewField routine 250
NewFieldFilter routine 279
NewFieldRect routine 255
NewIndexStringHandle routine 519
NewListBox routine 286
NewListBoxRect routine 289
NewMenu routine 549
NewPanel routine 336
NewPanelRect routine 341
NewPictButton routine 189
NewPopUp routine 311
NewPopUpRect routine 315
NewScrollBar routine 214
NewScrollBarActionProc routine 232
NewScrollBarRect routine 217
NewStrHandle routine 250
NewTimer routine 442
NewWideField routine 255
NewWideFieldRect routine 256
NewWindow routine 549
NMInstall routine 549
NMRemove routine 549
NoAltBut constant 502
NoBackdropColor routine 135
NoButtonAlert constant 502
NoButtonColors routine 173
NoChar constant 326, 327, 373
NoDefaultButton routine 185
NoGoAway constant 119
noGrowDocProc constant 120, 123
NoIcon constant 502
NoPopUpColors routine 319
NoScrollBarColors routine 220
noteIcon constant 502
Notification Manager 493, 494
NotModal constant 119
notSelected constant 166, 177, 190, 201, 204, 514
NoYesAlert constant 502
NoYesCanAlert constant 502
NumberOfMonitors routine 482
NumberOfScreens routine 468, 480
Numeric pad 243

O

ObscureButton routine 175
ObscureField routine 260
ObscureListBox routine 292
ObscurePanel routine 346
ObscurePictButton routine 200
ObscurePopUp routine 324
ObscureScrollBar routine 221
off constant 270, 271, 297, 326, 373, 390
OffsetButton routine 181
OffsetField routine 276
OffsetListBox routine 303
OffsetPanel routine 347
OffsetPictButton routine 206
OffsetPopUp routine 329
OffsetScrollBar routine 225
Offspring menu (see Menus)
OkAlert constant 502
OkAltBut constant 502
OkCanAlert constant 502
Old keyboard 243
on constant 270, 271, 297, 326, 373, 390
OpenDeskAcc routine 549
Option key 409
ordPaletteProc constant 120
outline constant 144, 145, 183, 228, 260, 300, 328, 331, 349, 375

P

P2CStr routine in C 89
PageDownKey constant 448, 458, 460
PageUpKey constant 448, 458, 460
paletteProc constant 120
Palettes (see Windows)
pan3DGroupBox constant 339
panAutoDeselect constant 338
panAutoMoveSize constant 338
panBlackBorder constant 337
panBWGrayBorder constant 337
panCenterTitle constant 338
panColorBack constant 338
panColorBorder constant 338
panColorText constant 338
panCustomColors constant 337
PanelDisplay routine 345
PanelIsVisible routine 345
PANELS 335
  3D Title
    Inset with heavy shadows 339
    Inset with soft shadows 339
    Plain style (no 3D) 339
    Raised with heavy shadows 339
    Raised with soft shadows 339
  Appearance/behavior specifications 337
  Automatic button deselection 338
  Automatic moving/resizing 338, 348
  Background filling 337
  Centered title 338
  Changing a panel's co-ordinates 347
  Changing a panel's size 347
  Changing co-ordinates 347
  Colors
    Custom color table 336, 337, 343, 344
    Getting a panel's colors 350
    Getting colors for multiple panels 342
    Getting custom color table 344
    Replacement color for background 338
    Replacement color for border 338
    Replacement color for title 338
    Setting a panel's colors 350
    Setting colors for multiple panels 342
    Setting custom color table 343
    Standard color table 336, 337, 342
  Creating a panel 336, 341
  Deleting a panel 344
  Determining if a panel is visible 345
  Font
    Getting the font, size and style 349
    Setting the font, size and style 335, 349
  Font (setting the font) 337
  Getting a panel's co-ordinates 346
  Group box
    3D look 339
    Standard look 339
  Hiding a panel 338, 345, 346
  Left-aligned title 338
  Moving a panel 346
  New panel (creating a) 336, 341
  Obscuring a panel 346
  Outline 337
  Outline on B&W monitor 337
  Plain style (no 3D) 339
  Right-aligned title 338
  Round corners 339
  Scrolling panels 346
  Setting Help for a panel 404
  Shadows
    Inset 339
    None 339
    Raised 339
  Showing a panel 345
  Unused panel number (getting the) 341
panFillBack constant 337
panGroupBox constant 339
panHidden constant 338
panInsetShadow constant 339
panInsetTitle constant 339
panInsetTitleDark constant 339
panLeftTitle constant 338
panNoBackdrop constant 338
panNoShadow constant 339
panOutline4bit constant 337
panOutlined constant 337
panPlainTitle constant 339
panRaiseShadow constant 339
panRaiseTitle constant 339
panRaiseTitleDark constant 339
panRightTitle constant 338
panRoundCorner1 through 31 constant 339
panUseWFont constant 337
Parameter range error 36
Parent menu (see Menus)
Pascal.h header file in C 89
PasteHIntoField routine 275
PasteIntoField routine 274
PastePIntoField routine 275
Pausing for a period of time 527
PenColorNormal routine 489
PenNormal routine 489, 549
Periodic tasks (see Event Management)
picbutAutoMoveSize constant 193
picbutAutoValueChg constant 194
picbutBigSICN3D constant 193
picbutDimAltImage constant 195
picbutDimLeaveBorder constant 195
picbutDimNoChange constant 195
picbutDimUsingBlackLite constant 195
picbutDimUsingWhite constant 195
picbutDimUsingWhiteLite constant 195
picbutFastAccel constant 205
picbutGray4use8 constant 193
picbutHidden constant 193
picbutInstantEvent constant 193
picbutLeftRightSplit constant 194
picbutLinear constant 205
picbutLockSelected constant 193
picbutMedAccel constant 205
picbutMultiStage constant 192
picbutRepeatEvents constant 193
picbutScaleFastAccel constant 194
picbutScaleLinear constant 194
picbutScaleMedAccel constant 194
picbutScaleSlowAccel constant 194
picbutSelectAltImage constant 194
picbutSelectDarken constant 194

picbutSelectDarkenSICN3D constant 194
picbutSelectLightenSICN3D constant 194
picbutSelectPushedSICN3D constant 194
picbutSlowAccel constant 205
picbutSwitchSelected constant 193
picbutTopBottomSplit constant 194
picbutTrackWithHilite constant 193
picbutUsePICTS constant 193
picbutValueWrap constant 194
Pick lists (see List Boxes or Pop-Up Menus)
PictButtonDisplay routine 199
PictButtonIsEnabled routine 201
PictButtonIsSelected routine 202
PictButtonIsVisible routine 200
pictBWplus constant 511
pictClipToRect constant 510
pictColor4plus constant 511
pictColor8plus constant 511
pictGray4plus constant 511
pictGray8plus constant 511
pictMultiPICT constant 511
pictOnBackdrop constant 511
pictOnColor constant 511
pictOnWhite constant 511
pictScale1PICT constant 510
PICTURE BUTTONS 187
  3D buttons 191, 194
  Appearance/behavior specifications 192
  Automatic moving 153, 193, 207
  Automatic value changes 194
  Changing co-ordinates 206
  Creating a picture button 189, 198
  Current value
    Getting the current value 203
    Setting the current value 204
  Deleting a picture button 199
  Deselecting
    Deselecting a picture button 201
    Is a picture button deselected 202
  Deselecting a picture button 204
  Determining if a picture button is visible 200
  Disabling
    Disabling a picture button 201
    Is a picture button disabled 201
  Disabling effects
    Alternate image 195
    Light (25%) overlay using black 195
    Light (25%) overlay using white 195
    Medium (50%) overlay using white 195
    No effect 195
    Preserving image's border 195
  Enabling
    Enabling a picture button 201
    Is a picture button enabled 201
  Enhanced gray scale pictures 193
  Flashing a picture button 206
  Getting a picture button's co-ordinates 200
  Hiding a button 193
  Hiding a picture button 199, 200
  Instant Events 193
  Large 3D buttons 193
  Linear scaling 194
  Locking in "selected" state 193
  Maximum value limit
    Getting the maximum value 203
    Setting the maximum value 203
  Minimum value limit
    Getting the minimum value 202
    Setting the minimum value 202
  Moderately accelerated scaling 194
  Moving a picture button 206
  Multiple stages 192
  New picture button (creating a) 189, 198
  Obscuring a picture button 200
  PICTs instead of icons 193
  Polarized
    Left/Right split 194, 466
    Top/bottom split 194, 466
  Rapidly accelerated scaling 194
  Repeating event rate 197
  Repeating events 193
  Resource IDs 190
  Scaling rate of value changes
    Fast acceleration 194
    Linear 194
    Medium acceleration 194
    Slow acceleration 194
  Scrolling picture buttons 200
  Selecting
    Is a picture button selected 202
    Selecting a picture button 201
  Selecting a picture button 204
  Selection effects
    Alternate image 194
    Darkening 194
    Darkening an SICN 3D button 194
    Lightening an SICN 3D button 194
    Pushing in an SICN 3D button 194
  Setting Help for a picture button 401
  Setting the value change rate 205
  Setting the value change speed 205
  Showing a picture button 199
  Slowly accelerated scaling 194
  Switching between selected/deselected state 193
  Tracking with highlighting 193
  Unused picture button number (getting the) 198
  Value wrapping 194
Picture control (see Buttons)
PICTURES
  Clipping 510
  Drawing across multiple monitors 511
  Drawing as per monitor settings 511
  Drawing pictures 509
  Offsetting in frame 513, 514
  Retaining original proportions 510
  Scaling 510
  Shifting in frame 513, 514
  Using pictures as buttons (see Picture Buttons)
pictUsePictRect constant 510
PixMaps 522
  Converting a PixMap to a region 525
  Create a PixMap 523
  Destroy a PixMap 524
Placard (see Buttons)
plainDBox constant 120, 123
Plug-ins 93, 121, 122, 354
plusCursor constant 383, 385, 386
Pointer (getting a window's pointer) 153

Polling (see Event Management)
Polling for events (see Event Management)
Pop-down menus (see Pop-Up Menus)
Pop-Up Arrows (see Buttons)
POP-UP MENUS 307
  Attaching a hierarchical menu 318
  Automatic moving/resizing 153, 313, 331
  Changing a pop-up menu's co-ordinates 330
  Changing a pop-up menu's size 330
  Changing co-ordinates 329
  Check mark (displaying, hiding) 320
  Color 102, 308, 312
    Default colors for multiple pop-up menus 319
    Getting a pop-up menu item's colors 334
    Getting a pop-up menu's colors 333
    Setting a pop-up menu item's colors 333
    Setting a pop-up menu's colors 332
    Setting colors for multiple pop-up menus 318
  Color resources 315
  Command key equivalents 309
  Creating a menu item 319
  Creating a pop-up menu 311, 315
  Creating using a 'MENU' resource 315, 316
  Default appearance and behavior 313
  Default spec for pop-ups created using a 'CNTL' 133
  Deleting a menu or item 322
  Deleting all items 322
  Detaching a hierarchical menu 318
  Determining if a pop-up menu is visible 324
  Dimming selected text 312
  Dimming the control's body 312
  Dimming the title 313
  Disabling
    Disabling menus/items 325
    Is a pop-up menu disabled 326
  Disabling menus/items 319
  Displaying icons in menus 319
  Dividing lines 320
  Down-arrow suppression 313
  Drawing on a background 312
  Enabling
    Enabling menus/items 325
    Is a pop-up menu enabled 326
  Fixed title in control body 313
  Font
    Getting the font, size and style 332
    Setting the font, size and style 308, 331
  Font (setting the font) 312
  Getting a pop-up menu's co-ordinates 323
  Getting an item's text 324
  Handle (getting it) 334
  Hiding a pop-up menu 313, 323, 324
  Hierarchical 309
  Icon
    Getting a menu item's icon 328
    Setting a menu item's icon 327
  Icon in control's body 313
  Icons in pop-up menus 320
  Insert a pop-up menu item 321
  Insert resource names 321
  Mark
    Check mark (displaying, hiding) 326
    Display/clear a menu item's "mark" 326
    Getting a menu item's "mark" 327
  Marking menu items (i.e., check mark) 319
  mctb resource 315
  MENU resource 308, 315
  Moving a pop-up menu 329
  New pop-up menu (creating a) 311, 315
  New pop-up menu (using 'MENU' resource) 315, 316
  Number of items in a menu 328
  Obscuring a pop-up menu 324
  Pop-down menu 313
  Scrolling pop-up menus 324
  Selecting multiple items 313
  Selecting single item only 313
  Selection
    Deselecting an item 326
    First item selected 329
    Is an item selected 327
    Selecting an item 326
  Setting Help for a pop-up menu 403
  Showing a pop-up menu 323
  Special characters (displaying, hiding) 320
  Styles
    3D body 312
    System's CDEF 312
    Tools Plus style 312
  Styles (setting a menu item's style) 319, 320, 328
  Title changes 325
  Unused pop-up menu number (getting the) 317
  Using a 'CNTL' resource 128, 129, 309
popup3DBody constant 312
popupAutoMoveSize constant 313
popupColorPopUp constant 312
PopUpColors routine 318
popupDefaultType constant 313
PopUpDisplay routine 323
popupDropDown constant 313
popupHasBackground constant 312
popupHidden constant 313
PopUpIcon routine 327
popupIconTitle constant 313
PopUpIsEnabled routine 326
PopUpIsVisible routine 324
PopUpItemCount routine 328
PopUpMark routine 326
PopUpMenu routine 319
PopUpMenuSelect routine 549
popupMultiSelect constant 313
popupNeverDimOutline constant 312
popupNeverDimSelection constant 312
popupNeverDimTitle constant 313
popupNoArrow constant 313
PopUpStyle routine 328
popupSystemBody constant 312
popupUseWFont constant 312
PostEvent routine 549
PostNotification routine 495
Power Macintosh performance 92
PPC .lib Converter 71
Preloading Tools Plus segments 64, 69, 73
Print Manager 141
Print Monitor 141
Printing text in windows 109
Procedures (see Tools Plus Routines)
Process1EventWhileBusy routine 430
ProcessEvents routine 430

ProcessToolboxEvent routine 431
Progress indicator (see Scroll Bars)
Pull-Down menus (see Menus)
Purging Tools Plus segments 64, 69, 73
Push buttons (see Buttons)
pushButProc constant 167

Q

QC™ 44
QDGlobals.a.o library 70
QDGlobals.c library 71
Quality control 44
Queuing (see Event Management)
QuickDraw 109
QuitAltBut constant 502
QuitToolsPlus routine 433

R

Radio buttons (see Buttons)
radioButProc constant 167
Range checking 36
rDocProc constant 120, 123
Rectangle being visiblein current window or grafPort 484
RectIsVisible routine 484
RedrawRect routine 487
RedrawRgn routine 487
RefreshDrawingInWindow routine 147
RefreshToolsPlusInWindow routine 146
Region being visiblein current window or grafPort 485
RegisterAppearanceClient routine 549
RemoveAllMenus routine 367
RemoveMenu routine 366
RemoveMenus routine 366
RemovePopUp routine 322
RenameItem routine 371
RenamePopUp routine 325
Repeating event rate for picture buttons 197
ReplaceControlProcID routine 186
ReplaceWindowProcID routine 154
ResEdit 34, 38, 40, 42, 45, 553
ResetCursor routine 384
ResetFieldScrolling routine 279
ResetMouseClicks routine 435
ResNamesToListBox routine 295
ResNamesToMenu routine 365
ResNamesToPopUp routine 321
Resorcerer 27, 40, 45, 391
Resource Editor 34, 38, 40, 42, 45
Resource file 553
RESOURCES
acur 382
BNDL 515, 553, 556
cctb 170, 171, 217, 218
CDEF 91, 102, 157, 158, 168, 186, 211, 216, 530, 531, 540
CDEF (irregular variant codes) 167
cicn 320, 360, 515
cicn (bug when displaying) 190, 515
cicn icon in pop-up menus 320
CNTL 128, 132, 133, 163, 170, 171, 213, 217, 218, 249, 285, 309
crsr 379, 382
CURS 382
dctb 127
dftb 116
DITL 110, 127, 131
DLOG 110, 127, 130, 131, 539
File containing them 553
FREF 556
hdlg 392, 393, 397
hfdr 392
hmnu 392, 397
hrct 392, 393, 397
icl4 515, 554
icl8 515, 554
ICN# 515, 554
ICON 320, 360, 501, 515
Icons (drawing them) 514, 517
Icons in buttons (see Picture Buttons)
ics# 515, 554
ics4 515, 554
ics8 515, 554
IDs for picture buttons 191
LDEF 287, 288, 540
ldes 284
List box (resource names added to) 295
MBAR 353, 362
mctb 315, 316, 361
MENU 308, 315, 316, 353, 361, 362
Menu (resource names added to) 365
mstr 553, 557
Owner resource 556
PICT (also see Pictures)
PICT (drawing them) 509
PICT ID numbers (for drawing) 510
PICT in buttons (see Picture Buttons), 191, 193
Pop-up menu (resource names added to) 321
Releasing resources 101
Reserved resource IDs 535
Resource Editors 34, 38, 40, 42, 45
SICN 191, 318, 320, 358, 360, 364, 494, 515, 517
SIZE 88, 426, 436, 553, 558
snd 494
STR 397
STR# 535
tab# 161
vers 553, 556
wctb 125
WDEF 91, 102, 113, 118, 120, 122, 154, 530, 531
WIND 110, 125, 126, 127
Resumed application 436
ReturnKey constant 448, 458, 460
ReverseKeyboardFocus routine 549
RGBBackColor routine 549
RGBForeColor routine 550
RgnIsVisible routine 485
RightArrowKey constant 448, 458, 460
ROM requirements 104
ROMs 39
Routines (see Tools Plus Routines)

S

SaveFieldString routine 269
Scheduled processing 435
screenBits.bounds global variable 116
ScreenDepth routine 480, 481
ScreenHasColors routine 480, 482
Screens (more than 2) 477
scrlAutoMoveSize constant 216
scrlBusyThermometerMinLimit constant 213
scrlColorScrollBar constant 215
scrlHidden constant 216
scrlLiveScroll constant 215
scrlNoObscure constant 216
scrlStandard constant 215
scrlValueLimit constant 215
SCROLL BARS 209
- Action routine 211, 231, 233
- Activating a scroll bar 222
- Appearance Manager 102, 111
- Appearance/behavior specifications 215
- Automatic moving/resizing 153, 216, 227
- cctb resource 217, 218
- Changing a scroll bar's co-ordinates 226
- Changing a scroll bar's size 226
- Changing co-ordinates 225
- Clicking in a scroll bar with keyboard focus 264
- CNTL resource 217, 218
- Color 210, 215
  - Default colors for multiple scroll bars 220
  - Getting a scroll bar's colors 229
  - Setting a scroll bar's colors 229
  - Setting colors for multiple scroll bars 219
- Color resources 217, 218
- Creating a scroll bar 214, 217
- Creating using a 'CNTL' resource 217, 218
- Current value
  - Getting the current value 224
  - Setting the current value 225
- Custom scroll bars 216
- Deleting a scroll bar 220
- Determining if a scroll bar is visible 221
- Disabling
  - Disabling a scroll bar 223
  - Is a scroll bar disabled 223
- Enabling
  - Enabling a scroll bar 223
  - Is a scroll bar enabled 223
- Fields (scrolling them) 244
- Font
  - Getting the font, size and style 228
  - Setting the font, size and style 210, 228
- Getting a scroll bar's co-ordinates 222
- Handle (getting it) 233
- Hiding a scroll bar 216, 221
- Limit value to min/max 215, 216
- Live scrolling 211, 215
- Maximum value limit
  - Getting the maximum limit 224
  - Setting the maximum limit 224
- Minimum value limit
  - Getting the minimum limit 223
  - Setting the minimum limit 224
- Moving a scroll bar 225
- New scroll bar (creating a) 214, 217
- New scroll bar (using 'CNTL' resource) 217, 218
- Obscuring a scroll bar 221
- Scrolling scroll bars 221
- Setting Help for a scroll bar 401, 402
- Showing a scroll bar 221
- Speed 210
  - Line scrolling for new scroll bars 230, 231
  - Page scrolling for new scroll bars 230, 231
- Styles
  - Custom CDEFs 215, 216
  - Little Arrows 128, 213
  - Progress indicator 128, 213
  - Slider 128, 213
  - Standard scroll bar 129, 213, 215
  - Thermometer 128, 213
- Substituting a ProcID throughout your app 91, 211, 530, 531
- Tabbing between scroll bars with keyboard focus 265, 266
- Text 210
- Throttling 210
- Unused scroll bar number (getting the) 219

Scroll Boxes (see List Boxes)
ScrollBarColors routine 219
ScrollBarDisplay routine 221
ScrollBarIsEnabled routine 223
ScrollBarIsVisible routine 221
ScrollBarLineTime routine 230
ScrollBarPageTime routine 230
scrollBarProc constant 215
Scrolling fields or "cells" 276
SDK 27
SearchListBox routine 296
Select All in Edit menu 356, 363
SelectButton routine 177
selected constant 166, 177, 190, 201, 204, 514
SelectPictButton routine 201
SelectWindow routine 550
SelIText routine 550
SendBehind routine 550
Set all bytes in a record to zero 532
Set68KStackSize routine 105
SetAlertBoxNullEvents routine 505
SetAlertBoxPrefs routine 504
SetApplLimit routine 97, 550
SetAutoEmbed routine 171
SetBackdropColor routine 136
SetBackgroundTheme routine 136
SetBackRGB routine 486
SetButtonColors routine 184
SetButtonFontSettings routine 183
SetButtonHelp routine 399
SetButtonHelpRes routine 400
SetButtonMax routine 179
SetButtonMin routine 178
SetButtonVal routine 179
SetCCursor routine 550
SetClikLoop routine 550
SetColorPenState routine 491, 492
SetControlData routine 550
SetControlFontStyle routine 550
SetControlHelp routine 406
SetControlHelpRes routine 406

SetControlVisibility routine 550
SetCTitle routine 550
SetCtlMax routine 550
SetCtlMin routine 550
SetCtlValue routine 550
SetCursor routine 550
SetCursorAnimation routine 384
SetCursorTableHelp routine 404
SetCursorTableHelpRes routine 404
SetCursorZoneCurs routine 387
SetCursorZoneHelp routine 405
SetCursorZoneHelpRes routine 405
SetCustomPanelColors routine 343
SetDAFont routine 550
SetDefaultButton routine 185
SetDialogCNTLEditTextSpec routine 132
SetDialogCNTLListBoxSpec routine 133
SetDialogCNTLPopUpSpec routine 133
SetDialogCNTLStaticTextSpec routine 133
SetDialogEditTextSpec routine 132
SetDialogFont routine 550
SetDialogFontInfo routine 144
SetDialogItemRect routine 143
SetDialogStaticTextSpec routine 132
SetDisabledFieldLook routine 273
SetDItem routine 550
SetEventError routine 431
SetEventMask routine 85, 550
SetFieldColors routine 261
SetFieldFilter routine 280
SetFieldFontSettings routine 260
SetFieldHelp routine 402
SetFieldHelpRes routine 402
SetFieldLengthLimit routine 271
SetFieldSelection routine 263
SetFrontRGB routine 486
SetGDevice routine 550
SetIndexString routine 520
SetItem routine 550
SetItemCmd routine 550
SetItemIcon routine 550
SetItemMark routine 550
SetItemStyle routine 550
SetIText routine 550
SetKeyboardFocus routine 550
SetListBoxColors routine 301
SetListBoxFontSettings routine 300
SetListBoxHelp routine 402
SetListBoxHelpRes routine 403
SetListBoxLine routine 297
SetListBoxText routine 294
SetLiveWindowDragging routine 154
SetMCEntries routine 550
SetMCInfo routine 550
SetMenuBar routine 550
SetMenuBarColors routine 368
SetMenuColors routine 369
SetMenuItemColors routine 370
SetNextWindowBackgroundTheme routine 137
SetNotification routine 494
SetNullTime routine 434
SetOrigin routine 550
SetPanelColors routine 350
SetPanelFontSettings routine 349
SetPanelHelp routine 404
SetPanelHelpRes routine 404
SetParamRangeErrProc routine 106
SetPenState routine 492
SetPictButtonAccel routine 205
SetPictButtonHelp routine 401
SetPictButtonHelpRes routine 401
SetPictButtonMax routine 203
SetPictButtonMin routine 202
SetPictButtonSpeed routine 205
SetPictButtonVal routine 204
SetPictButtonValSelect routine 204
SetPopUpColors routine 332
SetPopUpFontSettings routine 331
SetPopUpHelp routine 403
SetPopUpHelpRes routine 403
SetPopUpItemColors routine 333
SetResLoad routine 550
SetRGB routine 487
SetScrollBarAction routine 231
SetScrollBarColors routine 229
SetScrollBarFontSettings routine 228
SetScrollBarHelp routine 401
SetScrollBarHelpRes routine 402
SetScrollBarLineTime routine 231
SetScrollBarMax routine 224
SetScrollBarMin routine 224
SetScrollBarPageTime routine 231
SetScrollBarVal routine 225
SetSelectAllItem routine 363
SetStandardPanelColors routine 342
SetTELowMemThresh routine 282
SetTENoEditThresh routine 281
SetTENoUndoThresh routine 281
SetThemeWindowBackground routine 550
SetToZero routine 532
SetWinColor routine 550
SetWindowEventHandler routine 429
SetWindowPic routine 550
SetWindowSizeLimits routine 142
SetWindowZoom routine 142
SetWTitle routine 550
SetZone routine 550
shadow constant 144, 145, 183, 228, 260, 300, 328, 331, 349, 375
Shareware 43
Shift key 243, 409
shiftKey constant 417, 418
ShowControl routine 550
ShowCursor routine 383
ShowHide routine 550
ShowWindow routine 550
Signature 555, 556
SizeButton routine 181
SizeControl routine 550
SizeField routine 277
SizeListBox routine 303
SizePanel routine 347
SizePopUp routine 330
SizeScrollBar routine 226
SizeWindow routine 550
SkipAltBut constant 502
Slider (see Scroll Bars)
Software Development Kit (SDK) 27

Software updates 43
SOUND
  Playing the System Error sound 527
SpaceExtra routine 550
Special routines 547
Spool file name 141
Spotlight 45
STACK 98
  Consumption 541
  Size 98
Stand-alone applications 553
Static Text control (see Buttons), (see Editing Fields)
Static text in dialog (editing it) 110, 129
StoneTable 46
stopIcon constant 502
STR resource 397
STR#' resource;.i.RESOURCES:'STR# 397, 519
Stress testing 44
StrInBox routine 507
StrInBoxRect routine 508
STRINGS
  C string parameters in Tools Plus routines 89
  Drawing 507
  Efficient storage for fields 235, 252, 271
  Indexed
    Appending a string 521
    Counting strings in a record 520
    Creating a new record 519
    Definition 519
    Deleting a string 521
    Getting a string 520
    Inserting a string 521
    Setting a string 520
  Memory-efficient arrays 519
  Pascal string parameters in Tools Plus routines 89
  Pascal String versus C String 89
  Str255 in C 89
Strobing (preventing it) 528
StrToListBox routine 295
Style Table 544
Submenus (see Menus)
Support for programmers 561
Suspended application 436
Symantec C/C++ 39
Synchronize to vertical retrace 528
SynchToVideo routine 528
SysBeep routine 550
System 6 (programming for) 88
System 7 115, 354, 410, 557, 558
System 7 and higher (programming for) 88
System Extensions 39
System file version 525
System Requirements 39, 40
SystemClick routine 550
SystemEdit routine 550
SystemEvent routine 550
SystemMenu routine 550
SystemVersion routine 525
_SYSV routine 525

T

Tab controls (see Buttons)
Tabbing in fields 237
TabKey constant 448, 458, 460
Tables 46
TabToFocus routine 266, 447, 457
Task switching 88, 410, 558
tbOffsetNewWindows constant 119, 134, 142, 143
tbShiftWindows constant 134
TEActive routine 551
teAllowCR constant 251
teAutoMoveSize constant 252
TEAutoView routine 551
teBackdrop constant 253, 507, 508
teBlackOnBackdrop constant 508
teBlackOnClear constant 508
teBlackOnColor constant 508
teBlackOnWhite constant 508
teBlackText constant 253, 508
teBottomEdge constant 251
teBox constant 251
teBuffered constant 252
TECalc routine 551
Technical support 561
TEClick routine 551
teClickBeep constant 272, 274
teColorBack constant 253, 508
teColorOnBackdrop constant 508
teColorOnClear constant 508
teColorOnColor constant 508
teColorOnWhite constant 508
teColorText constant 253, 508
teColSys6Box constant 272, 274
teColSys6Text constant 272, 273
TECopy routine 551
teCstring constant 252
TECut routine 551
TEDeactivate routine 551
TEDelete routine 551
teDfltDisabledLook constant 272, 274
teDimWhenInactive constant 251
teDisabled constant 252
TEDispose routine 551
teFilter constant 252
TEFromScrap routine 551
TEGetText routine 551
teHidden constant 252
teHScrollRight15 constant 256
teHScrollRight30 constant 256
teHScrollRight45 constant 256
teHScrollRight60 constant 256
teHScrollRight75 constant 256
teHScrollRight90 constant 256
teHScrollRight105 constant 256
teHScrollRight120 constant 256
teHScrollRight135 constant 256
teHScrollRight150 constant 256
teHScrollRight165 constant 256
teHScrollRight180 constant 256
teHScrollRight195 constant 256
teHScrollRight210 constant 256
teHScrollRight225 constant 256
TEIdle routine 551

teIgnoreCase constant 279
teIgnoreDiac constant 279
TEInit routine 97, 551
teInsert constant 274, 275
TEInsert routine 551
teJustCenter constant 251, 508
teJustLeft constant 251, 508
teJustRight constant 251, 508
TEKey routine 551
teLeftEdge constant 251
teLengthLimit constant 252
Telephone number for Water's Edge Software 562
teLiveScroll constant 252
teNeverDimBWBox constant 272, 273
teNeverDimBWText constant 272, 273
teNeverDimColorBox constant 272, 273
teNeverDimColorText constant 272, 273
TENew routine 551
teNoBack constant 508
teNoBox constant 252
teNoCR constant 251
teNoResetOnDeactivate constant 252
TEPaste routine 551
TEPinScroll routine 551
teReadOnly constant 251
teReplace constant 274, 275
teResizeHdl constant 252
teRightEdge constant 251
TEScroll routine 551
TESelect routine 551
teSelectAll constant 262
teSelectEnd constant 262
teSelectStart constant 262
TESelView routine 551
TESetClickLoop routine 551
TESetJust routine 551
TESetText routine 551
teShiftCaseDown constant 279
teShiftCaseUp constant 279
teStaticText constant 251
Testing your application 44
teSystemBody constant 251
teTabSelectAll constant 253
teTabSelectEnd constant 253
teTabSelectStart constant 253
teTopEdge constant 251
TEToScrap routine 551
TEUpdate routine 551
teUseWFont constant 251
teVScroll constant 252
teVScrollDown30 constant 253
teVScrollDown45 constant 253
teVScrollDown60 constant 253
teVScrollDown75 constant 253
teVScrollDown90 constant 253
teVScrollDown15 constant 253
teVScrollDown105 constant 253
teVScrollDown120 constant 253
teVScrollDown135 constant 253
teVScrollDown150 constant 253
teVScrollDown165 constant 253
teVScrollDown180 constant 253
teVScrollDown195 constant 253
teVScrollDown210 constant 253
teVScrollDown225 constant 253
teWhiteBack constant 253, 507, 508
TEXT
    Alignment (left, right, centre) 508
    Drawing 509
Text Editing (see Fields)
TextEdit records 244
TextInBox routine 509
TextInBoxRect routine 509
Thermometer (see Scroll Bars)
Thermometer (drawing one) 530
THINK C/C++ 39
THINK Pascal 39
THINK Pascal (issues with Balloon Help) 398
THINK Reference 43
timerDaysBetweenEvents constant 444
timerDeleteForHiddenWindow constant 443, 444
timerEventsPerDay constant 444
timerEventsPerHour constant 444
timerEventsPerMinute constant 444
timerEventsPerSecond constant 443, 444
timerEventsPerTick constant 443, 444
timerHoursBetweenEvents constant 444
timerInstantEvent constant 444
timerLockTimerToCount constant 443, 444
timerMinutesBetweenEvents constant 444
timerOneShot constant 443, 444
TIMERS
    Always report physical event for each logical event 443
    Behavior specifications 443
    Creating a new Timer 442
    Delete automatically 443
    Deleting a timer 445
    Frequency
        Events per day 444
        Events per hour 444
        Events per minute 444
        Events per second 443
        Events per tick (1/60 second) 443
    One shot 443
    Overview 438
    Period
        Days between events 444
        Hours between events 444
        Minutes between events 444
        Seconds between events 444
        Ticks between events 444
    Synchronize to another Timer 443
timerSecondsBetweenEvents constant 444
timerStandardInitDelay constant 444
timerSyncToTimer constant 443, 444
timerTicksBetweenEvents constant 444
TITLES
    Button title changes 180
    Getting a menu item's text 371
    Getting a pop-up menu item's text 324
    List box lines 294, 296
    Menu title changes 360, 371
    Pop-up menu title changes 325
    Window title changes 141
Tool Bar (see Windows)
ToolBarNumber routine 149
ToolBarOpen routine 134
Toolbox events (see Event Management)

Toolbox routines to be used with caution or avoided 547
Tools Plus events (see Event Management)
TOOLS PLUS ROUTINES (All, sorted alphabetically)
ActivateButton 175
ActivateField 262
ActivateListBox 293
ActivateScrollBar 222
ActivateWindow 140
ActiveFieldNumber 270
ActiveWindowNumber 148
AlertBox 422, 501
AlertBox3 422, 503
AlertBoxCount 505
AlertButtonName 503
AnimateCursor 384
AppendDialogList 131
AppleMenu 359
ApplicationSuspended 436
AttachMenu 364
AttachPopUpSubMenu 318
AutoMoveSize 153
AutoMoveSizeButton 182
AutoMoveSizeField 278
AutoMoveSizeListBox 304
AutoMoveSizePanel 348
AutoMoveSizePictButton 207
AutoMoveSizePopUp 331
AutoMoveSizeScrollBar 227
BackdropColor 135
Beep 527
BeepSynch 527
BeginUpdateScreen 480
BitMap2Region 525
ButtonColors 173
ButtonDisplay 174
ButtonIsEnabled 177
ButtonIsSelected 177
ButtonIsVisible 175
ButtonTitle 180
ChangedCursorZone 388
ChangedHelp 407
ChangeStackSize 106
CheckForMonitorChanges 482
CheckMenu 372
CheckPopUp 326
ClearFocus 140
ClearListBox 299
ClearPopUp 322
ClickToFocus 264
CountIndexString 520
CountNumberOfFiles 432
CreateBitMap 523
CurrentFieldFilter 280
CurrentWindow 141
CurrentWindowNumber 148
CurrentWindowReset 141
CursorShape 383
CursorZone 386
CursorZoneRect 386
CursorZoneRgn 386
DeactivateField 263
DefaultIconLook 518
DeinitToolsPlus 105
DeleteButton 174
DeleteControl 406
DeleteCursorTable 385
DeleteCursorZone 387
DeleteField 258
DeleteIndexString 521
DeleteListBox 291
DeleteListBoxLine 299
DeletePanel 344
DeletePictButton 199
DeleteScrollBar 220
DeleteTimer 445
DestroyBitMap 524
DisabledFieldLook 271
DrawIcon 514
DrawListBox 302
DrawPict 509
DrawPictRect 513
DrawShiftPict 513
DrawShiftPictRect 514
DrawSICN 517
DrawSICNmode 517
DrawThermometer 530
DynamicFieldHandles 271
EditFldWindowNumber 150, 270
EmbedButtonInButton 172
EmbedButtonInScrollBar 172
EmbedFieldInButton 257
EmbedFieldInScrollBar 257
EmbedListBoxInButton 290
EmbedListBoxInScrollBar 290
EmbedPopUpInButton 316
EmbedPopUpInScrollBar 317
EmbedScrollBarInButton 218
EmbedScrollBarInScrollBar 218
EnableButton 176
EnableField 264
EnableMenu 372
EnablePictButton 201
EnablePopUp 325
EnableScrollBar 223
EndUpdateScreen 481
EqualMem 532
FieldDisplay 259
FieldIsEmpty 269
FieldIsEnabled 264
FieldIsVisible 259
FieldLengthLimit 270
FindCursorZone 389
FinderDisplay 153
FirstPaletteNumber 149
FirstStdWindowNumber 149
FirstWindowNumber 148
FlashButton 180
FlashPictButton 206
FocusWindowNumber 150
GetAlertBoxPrefs 503
GetBackRGB 486
GetButtonColors 184
GetButtonFontSettings 183
GetButtonHandle 185
GetButtonMax 178
GetButtonMin 178
GetButtonRect 176
GetButtonVal 179

GetColorPenState 491
GetCurrentCursorZone 389
GetCursorZone 388
GetCursorZoneRgn 388
GetCustomPanelColors 344
GetDialogFontInfo 144
GetDialogItemRect 144
GetDimColor 488
GetEditHandle 267
GetEditLength 267
GetEditString 267
GetFieldColors 262
GetFieldFontSettings 261
GetFieldHandle 268
GetFieldLength 269
GetFieldRect 260
GetFieldSelection 263
GetFieldString 268
GetFocusInfo 152
GetFreeButtonNum 173
GetFreeCursorTableNum 385
GetFreeCursorZoneNum 387
GetFreeFieldNum 258
GetFreeHMenuNum 363
GetFreeListBoxNum 291
GetFreeMenuNum 363
GetFreePanelNum 341
GetFreePictButtonNum 198
GetFreePopUpNum 317
GetFreeScrollBarNum 219
GetFreeWindowNum 135
GetFrontRGB 485
GetIndexFile 432
GetIndexFileFSS 433
GetIndexString 520
GetListBoxColors 301
GetListBoxFontSettings 300
GetListBoxHandle 305
GetListBoxLine 297
GetListBoxLines 298
GetListBoxRect 293
GetListBoxText 296
GetMenuBarColors 368
GetMenuCmd 374
GetMenuColors 369
GetMenuHandleFromMemory 377
GetMenuIcon 375
GetMenuItemColors 370
GetMenuMark 373
GetMenuString 371
GetPanelColors 350
GetPanelFontSettings 349
GetPanelRect 346
GetParentMenu 376
GetPictButtonMax 203
GetPictButtonMin 202
GetPictButtonRect 200
GetPictButtonVal 203
GetPopUpColors 333
GetPopUpFontSettings 332
GetPopUpHandle 334
GetPopUpIcon 328
GetPopUpItemColors 334
GetPopUpMark 327
GetPopUpRect 323
GetPopUpSelection 329
GetPopUpString 324
GetScrollBarActionInfo 233
GetScrollBarColors 229
GetScrollBarFontSettings 228
GetScrollBarHandle 233
GetScrollBarMax 224
GetScrollBarMin 223
GetScrollBarRect 222
GetScrollBarVal 224
GetStandardPanelColors 342
GetSubMenu 377
GetTEHandle 282
GetToolsPlusVersion 526
GetTPSerialEvent 437
GetWindowInOrder 147
GetWindowZoom 143
HasAppearanceManager 530
HasAppearanceManagerRoutines 531
HasColorQuickDraw 479
HaveTabInFocus 265, 447, 457
HiliteRect 490
HiliteRgn 491
IgnoreFirstMouseClick 436
InitToolsPlus 97, 99
InsertIndexString 521
InsertListBoxLine 298
InsertMenuItm 364
InsertPopUpItem 321
KillButton 174
KillField 259
KillListBox 291
KillPanel 345
KillPictButton 199
KillPopUp 323
KillScrollBar 220
KillTPSerialEvent 437
ListBoxDisplay 292
ListBoxIsEnabled 299
ListBoxIsVisible 292
ListBoxLineCount 301
LoadButton 170
LoadDialog 127
LoadDialogList 131
LoadDialogPopUp 316
LoadMenu 361
LoadMenuBar 362
LoadPopUp 315
LoadPopUpRect 316
LoadScrollBar 217
LoadSpecButton 171
LoadSpecDialog 130
LoadSpecDialogBehind 131
LoadSpecScrollBar 218
LoadSpecWindow 126
LoadSpecWindowBehind 127
LoadWindow 125
MainMonitorNumber 484
Max 533
Menu 359
MenuBarDisplay 367
MenuCmd 374
MenuHilite 377

MenuIcon 374
MenuItemCount 376
MenuMark 373
MenuStyle 375
Min 533
MonitorDepth 483
MonitorGDevice 484
MonitorHasColors 483
MoveButton 180
MoveField 276
MoveListBox 302
MovePanel 346
MovePictButton 206
MovePopUp 329
MoveScrollBar 225
MoveSizeButton 181
MoveSizeButtonRect 182
MoveSizeField 277
MoveSizeFieldRect 278
MoveSizeListBox 303
MoveSizeListBoxRect 304
MoveSizePanel 347
MoveSizePanelRect 348
MoveSizePopUp 330
MoveSizePopUpRect 330
MoveSizeScrollBar 226
MoveSizeScrollBarRect 227
NewButton 166
NewButtonControl 169
NewButtonControlRect 169
NewButtonRect 169
NewCursorTable 385
NewDialogButton 170
NewDialogButtonControl 170
NewDialogField 255
NewDialogListBox 289
NewDialogPanel 341
NewDialogPictButton 198
NewDialogScrollBar 217
NewDialogWideField 257
NewEventHandlerProc 429
NewField 250
NewFieldFilter 279
NewFieldRect 255
NewIndexStringHandle 519
NewListBox 286
NewListBoxRect 289
NewPanel 336
NewPanelRect 341
NewPictButton 189
NewPopUp 311
NewPopUpRect 315
NewScrollBar 214
NewScrollBarActionProc 232
NewScrollBarRect 217
NewStrHandle 250
NewTimer 442
NewWideField 255
NewWideFieldRect 256
NoBackdropColor 135
NoButtonColors 173
NoDefaultButton 185
NoPopUpColors 319
NoScrollBarColors 220
NumberOfMonitors 482
NumberOfScreens 480
ObscureButton 175
ObscureField 260
ObscureListBox 292
ObscurePanel 346
ObscurePictButton 200
ObscurePopUp 324
ObscureScrollBar 221
OffsetButton 181
OffsetField 276
OffsetListBox 303
OffsetPanel 347
OffsetPictButton 206
OffsetPopUp 329
OffsetScrollBar 225
PanelDisplay 345
PanelIsVisible 345
PasteHIntoField 275
PasteIntoField 274
PastePIntoField 275
PenColorNormal 489
PictButtonDisplay 199
PictButtonIsEnabled 201
PictButtonIsSelected 202
PictButtonIsVisible 200
PopUpColors 318
PopUpDisplay 323
PopUpIcon 327
PopUpIsEnabled 326
PopUpIsVisible 324
PopUpItemCount 328
PopUpMark 326
PopUpMenu 319
PopUpStyle 328
PostNotification 495
Process1EventWhileBusy 422, 430
ProcessEvents 430
ProcessToolboxEvent 422, 431
QuitToolsPlus 433
RectIsVisible 484
RedrawRect 487
RedrawRgn 487
RefreshDrawingInWindow 147
RefreshToolsPlusInWindow 146
RemoveAllMenus 367
RemoveMenu 366
RemoveMenus 366
RemovePopUp 322
RenameItem 371
RenamePopUp 325
ReplaceControlProcID 186
ReplaceWindowProcID 154
ResetCursor 384
ResetFieldScrolling 279
ResetMouseClicks 435
ResNamesToListBox 295
ResNamesToMenu 365
ResNamesToPopUp 321
RgnIsVisible 485
SaveFieldString 269
ScreenDepth 481
ScreenHasColors 482
ScrollBarColors 219

ScrollBarDisplay 221
ScrollBarIsEnabled 223
ScrollBarIsVisible 221
ScrollBarLineTime 230
ScrollBarPageTime 230
SearchListBox 296
SelectButton 177
SelectPictButton 201
Set68KStackSize 105
SetAlertBoxNullEvents 505
SetAlertBoxPrefs 504
SetAutoEmbed 171
SetBackdropColor 136
SetBackgroundTheme 136
SetBackRGB 486
SetButtonColors 184
SetButtonFontSettings 183
SetButtonHelp 399
SetButtonHelpRes 400
SetButtonMax 179
SetButtonMin 178
SetButtonVal 179
SetColorPenState 492
SetControlHelp 406
SetControlHelpRes 406
SetCursorAnimation 384
SetCursorTableHelp 404
SetCursorTableHelpRes 404
SetCursorZoneCurs 387
SetCursorZoneHelp 405
SetCursorZoneHelpRes 405
SetCustomPanelColors 343
SetDefaultButton 185
SetDialogCNTLEditTextSpec 132
SetDialogCNTLListBoxSpec 133
SetDialogCNTLPopUpSpec 133
SetDialogCNTLStaticTextSpec 133
SetDialogEditTextSpec 132
SetDialogFontInfo 144
SetDialogItemRect 143
SetDialogStaticTextSpec 132
SetDisabledFieldLook 273
SetEventError 431
SetFieldColors 261
SetFieldFilter 280
SetFieldFontSettings 260
SetFieldHelp 402
SetFieldHelpRes 402
SetFieldLengthLimit 271
SetFieldSelection 263
SetFrontRGB 486
SetIndexString 520
SetListBoxColors 301
SetListBoxFontSettings 300
SetListBoxHelp 402
SetListBoxHelpRes 403
SetListBoxLine 297
SetListBoxText 294
SetLiveWindowDragging 154
SetMenuBarColors 368
SetMenuColors 369
SetMenuItemColors 370
SetNextWindowBackgroundTheme 137
SetNotification 494
SetNullTime 434
SetPanelColors 350
SetPanelFontSettings 349
SetPanelHelp 404
SetPanelHelpRes 404
SetParamRangeErrProc 106
SetPictButtonAccel 205
SetPictButtonHelp 401
SetPictButtonHelpRes 401
SetPictButtonMax 203
SetPictButtonMin 202
SetPictButtonSpeed 205
SetPictButtonVal 204
SetPictButtonValSelect 204
SetPopUpColors 332
SetPopUpFontSettings 331
SetPopUpHelp 403
SetPopUpHelpRes 403
SetPopUpItemColors 333
SetRGB 487
SetScrollBarAction 231
SetScrollBarColors 229
SetScrollBarFontSettings 228
SetScrollBarHelp 401
SetScrollBarHelpRes 402
SetScrollBarLineTime 231
SetScrollBarMax 224
SetScrollBarMin 224
SetScrollBarPageTime 231
SetScrollBarVal 225
SetSelectAllItem 363
SetStandardPanelColors 342
SetTELowMemThresh 282
SetTENoEditThresh 281
SetTENoUndoThresh 281
SetToZero 532
SetWindowEventHandler 429
SetWindowSizeLimits 142
SetWindowZoom 142
SizeButton 181
SizeField 277
SizeListBox 303
SizePanel 347
SizePopUp 330
SizeScrollBar 226
StrInBox 507
StrInBoxRect 508
StrToListBox 295
SynchToVideo 528
SystemVersion 525
TabToFocus 266, 447, 457
TextInBox 509
TextInBoxRect 509
ToolBarNumber 149
ToolBarOpen 134
ToolsPlusIsQuitting 434
ToolsPlusLanguage 537
UpdateMenuBar 367
UseCursorTable 389
UseHiliteColor 489
UseHiliteText 489
UsingAppearanceManager 531
Wait 527
WaitAvail 435

WaitForMultiClicks 435
WatchCursorButtons 390
WindowClose 137
WindowDisplay 139
WindowDisplayBehind 140
WindowIsActive 151
WindowIsOpen 151
WindowIsVisible 151
WindowKind 152
WindowMove 138
WindowOpen 119
WindowOpenRect 124
WindowOpenRectBehind 125
WindowPointer 153
WindowSize 138
WindowStatus 145
WindowTitle 141
WorkWindowNumber 150
ZoomLines 528
TOOLS PLUS ROUTINES (Buttons only)
ActivateButton 175
AutoMoveSizeButton 182
ButtonColors 173
ButtonDisplay 174
ButtonIsEnabled 177
ButtonIsSelected 177
ButtonIsVisible 175
ButtonTitle 180
DeleteButton 174
EmbedButtonInButton 172
EmbedButtonInScrollBar 172
EnableButton 176
FlashButton 180
GetButtonColors 184
GetButtonFontSettings 183
GetButtonHandle 185
GetButtonMax 178
GetButtonMin 178
GetButtonRect 176
GetButtonVal 179
GetFreeButtonNum 173
KillButton 174
LoadButton 170
LoadSpecButton 171
MoveButton 180
MoveSizeButton 181
MoveSizeButtonRect 182
NewButton 166
NewButtonControl 169
NewButtonControlRect 169
NewButtonRect 169
NewDialogButton 170
NewDialogButtonControl 170
NoButtonColors 173
NoDefaultButton 185
ObscureButton 175
OffsetButton 181
ReplaceControlProcID 186
SelectButton 177
SetAutoEmbed 171
SetButtonColors 184
SetButtonFontSettings 183
SetButtonMax 179
SetButtonMin 178
SetButtonVal 179
SetDefaultButton 185
SizeButton 181
TOOLS PLUS ROUTINES (Color & Multiple Monitors only)
BeginUpdateScreen 480
CheckForMonitorChanges 482
EndUpdateScreen 481
GetBackRGB 486
GetColorPenState 491
GetDimColor 488
GetFrontRGB 485
HasColorQuickDraw 479
HiliteRect 490
HiliteRgn 491
MainMonitorNumber 484
MonitorDepth 483
MonitorGDevice 484
MonitorHasColors 483
NumberOfMonitors 482
NumberOfScreens 480
PenColorNormal 489
RectIsVisible 484
RedrawRect 487
RedrawRgn 487
RgnIsVisible 485
ScreenDepth 481
ScreenHasColors 482
SetBackRGB 486
SetColorPenState 492
SetFrontRGB 486
SetRGB 487
UseHiliteColor 489
TOOLS PLUS ROUTINES (Cursors only)
AnimateCursor 384
ChangedCursorZone 388
CursorShape 383
CursorZone 386
CursorZoneRect 386
CursorZoneRgn 386
DeleteCursorTable 385
DeleteCursorZone 387
FindCursorZone 389
GetCurrentCursorZone 389
GetCursorZone 388
GetCursorZoneRgn 388
GetFreeCursorTableNum 385
GetFreeCursorZoneNum 387
NewCursorTable 385
ResetCursor 384
SetCursorAnimation 384
SetCursorZoneCurs 387
UseCursorTable 389
WatchCursorButtons 390
TOOLS PLUS ROUTINES (Dynamic Alerts only)
AlertBox 422, 501
AlertBox3 422, 503
AlertBoxCount 505
AlertButtonName 503
GetAlertBoxPrefs 503
SetAlertBoxNullEvents 505
SetAlertBoxPrefs 504
TOOLS PLUS ROUTINES (Editing Fields only)
ActivateField 262
ActiveFieldNumber 270

AutoMoveSizeField 278
ClickToFocus 264
CurrentFieldFilter 280
DeactivateField 263
DeleteField 258
DisabledFieldLook 271
DynamicFieldHandles 271
EditFldWindowNumber 270
EmbedFieldInButton 257
EmbedFieldInScrollBar 257
EnableField 264
FieldDisplay 259
FieldIsEmpty 269
FieldIsEnabled 264
FieldIsVisible 259
FieldLengthLimit 270
GetEditHandle 267
GetEditLength 267
GetEditString 267
GetFieldColors 262
GetFieldFontSettings 261
GetFieldHandle 268
GetFieldLength 269
GetFieldRect 260
GetFieldSelection 263
GetFieldString 268
GetFreeFieldNum 258
GetTEHandle 282
HaveTabInFocus 265
KillField 259
MoveField 276
MoveSizeField 277
MoveSizeFieldRect 278
NewDialogField 255
NewDialogWideField 257
NewField 250
NewFieldFilter 279
NewFieldRect 255
NewStrHandle 250
NewWideField 255
NewWideFieldRect 256
ObscureField 260
OffsetField 276
PasteHIntoField 275
PasteIntoField 274
PastePIntoField 275
ResetFieldScrolling 279
SaveFieldString 269
SetDisabledFieldLook 273
SetFieldColors 261
SetFieldFilter 280
SetFieldFontSettings 260
SetFieldLengthLimit 271
SetFieldSelection 263
SetTELowMemThresh 282
SetTENoEditThresh 281
SetTENoUndoThresh 281
SizeField 277
TabToFocus 266

TOOLS PLUS ROUTINES (Event Management only)
ApplicationSuspended 436
CountNumberOfFiles 432
DeleteTimer 445
GetIndexFile 432
GetIndexFileFSS 433
GetTPSerialEvent 437
HaveTabInFocus 447, 457
IgnoreFirstMouseClick 436
KillTPSerialEvent 437
NewEventHandlerProc 429
NewTimer 442
Process1EventWhileBusy 422, 430
ProcessEvents 430
ProcessToolboxEvent 422, 431
QuitToolsPlus 433
ResetMouseClicks 435
SetEventError 431
SetNullTime 434
SetWindowEventHandler 429
TabToFocus 447, 457
ToolsPlusIsQuitting 434
WaitAvail 435
WaitForMultiClicks 435

TOOLS PLUS ROUTINES (Help only)
ChangedHelp 407
DeleteControl 406
SetButtonHelp 399
SetButtonHelpRes 400
SetControlHelp 406
SetControlHelpRes 406
SetCursorTableHelp 404
SetCursorTableHelpRes 404
SetCursorZoneHelp 405
SetCursorZoneHelpRes 405
SetFieldHelp 402
SetFieldHelpRes 402
SetListBoxHelp 402
SetListBoxHelpRes 403
SetPanelHelp 404
SetPanelHelpRes 404
SetPictButtonHelp 401
SetPictButtonHelpRes 401
SetPopUpHelp 403
SetPopUpHelpRes 403
SetScrollBarHelp 401
SetScrollBarHelpRes 402

TOOLS PLUS ROUTINES (Initialization only)
ChangeStackSize 106
DeinitToolsPlus 105
InitToolsPlus 97, 99
Set68KStackSize 105
SetParamRangeErrProc 106

TOOLS PLUS ROUTINES (List Boxes only)
ActivateListBox 293
AutoMoveSizeListBox 304
ClearListBox 299
DeleteListBox 291
DeleteListBoxLine 299
DrawListBox 302
EmbedListBoxInButton 290
EmbedListBoxInScrollBar 290
GetFreeListBoxNum 291
GetListBoxColors 301
GetListBoxFontSettings 300
GetListBoxHandle 305
GetListBoxLine 297
GetListBoxLines 298
GetListBoxRect 293

- GetListBoxText 296
- InsertListBoxLine 298
- KillListBox 291
- ListBoxDisplay 292
- ListBoxIsEnabled 299
- ListBoxIsVisible 292
- ListBoxLineCount 301
- MoveListBox 302
- MoveSizeListBox 303
- MoveSizeListBoxRect 304
- NewDialogListBox 289
- NewListBox 286
- NewListBoxRect 289
- ObscureListBox 292
- OffsetListBox 303
- ResNamesToListBox 295
- SearchListBox 296
- SetListBoxColors 301
- SetListBoxFontSettings 300
- SetListBoxLine 297
- SetListBoxText 294
- SizeListBox 303
- StrToListBox 295

TOOLS PLUS ROUTINES (Menus only)
- AppleMenu 359
- AttachMenu 364
- CheckMenu 372
- EnableMenu 372
- GetFreeHMenuNum 363
- GetFreeMenuNum 363
- GetMenuBarColors 368
- GetMenuCmd 374
- GetMenuColors 369
- GetMenuHandleFromMemory 377
- GetMenuIcon 375
- GetMenuItemColors 370
- GetMenuMark 373
- GetMenuString 371
- GetParentMenu 376
- GetSubMenu 377
- InsertMenuItm 364
- LoadMenu 361
- LoadMenuBar 362
- Menu 359
- MenuBarDisplay 367
- MenuCmd 374
- MenuHilite 377
- MenuIcon 374
- MenuItemCount 376
- MenuMark 373
- MenuStyle 375
- RemoveAllMenus 367
- RemoveMenu 366
- RemoveMenus 366
- RenameItem 371
- ResNamesToMenu 365
- SetMenuBarColors 368
- SetMenuColors 369
- SetMenuItemColors 370
- SetSelectAllItem 363
- UpdateMenuBar 367

TOOLS PLUS ROUTINES (Miscellaneous Routines only)
- Beep 527
- BeepSynch 527
- BitMap2Region 525
- CountIndexString 520
- CreateBitMap 523
- DefaultIconLook 518
- DeleteIndexString 521
- DestroyBitMap 524
- DrawIcon 514
- DrawPict 509
- DrawPictRect 513
- DrawShiftPict 513
- DrawShiftPictRect 514
- DrawSICN 517
- DrawSICNmode 517
- DrawThermometer 530
- EqualMem 532
- GetIndexString 520
- GetToolsPlusVersion 526
- HasAppearanceManager 530
- HasAppearanceManagerRoutines 531
- InsertIndexString 521
- Max 533
- Min 533
- NewIndexStringHandle 519
- SetIndexString 520
- SetToZero 532
- StrInBox 507
- StrInBoxRect 508
- SynchToVideo 528
- SystemVersion 525
- TextInBox 509
- TextInBoxRect 509
- UsingAppearanceManager 531
- Wait 527
- ZoomLines 528

TOOLS PLUS ROUTINES (Multiple Languages only)
- ToolsPlusLanguage 537

TOOLS PLUS ROUTINES (Panels only)
- AutoMoveSizePanel 348
- DeletePanel 344
- GetCustomPanelColors 344
- GetFreePanelNum 341
- GetPanelColors 350
- GetPanelFontSettings 349
- GetPanelRect 346
- GetStandardPanelColors 342
- KillPanel 345
- MovePanel 346
- MoveSizePanel 347
- MoveSizePanelRect 348
- NewDialogPanel 341
- NewPanel 336
- NewPanelRect 341
- ObscurePanel 346
- OffsetPanel 347
- PanelDisplay 345
- PanelIsVisible 345
- SetCustomPanelColors 343
- SetPanelColors 350
- SetPanelFontSettings 349
- SetStandardPanelColors 342
- SizePanel 347

TOOLS PLUS ROUTINES (Picture Buttons only)
- AutoMoveSizePictButton 207
- DeletePictButton 199

EnablePictButton 201
FlashPictButton 206
GetFreePictButtonNum 198
GetPictButtonMax 203
GetPictButtonMin 202
GetPictButtonRect 200
GetPictButtonVal 203
KillPictButton 199
MovePictButton 206
NewDialogPictButton 198
NewPictButton 189
ObscurePictButton 200
OffsetPictButton 206
PictButtonDisplay 199
PictButtonIsEnabled 201
PictButtonIsSelected 202
PictButtonIsVisible 200
SelectPictButton 201
SetPictButtonAccel 205
SetPictButtonMax 203
SetPictButtonMin 202
SetPictButtonSpeed 205
SetPictButtonVal 204
SetPictButtonValSelect 204
TOOLS PLUS ROUTINES (Pop-Up Menus only)
AttachPopUpSubMenu 318
AutoMoveSizePopUp 331
CheckPopUp 326
ClearPopUp 322
EmbedPopUpInButton 316
EmbedPopUpInScrollBar 317
EnablePopUp 325
GetFreePopUpNum 317
GetPopUpColors 333
GetPopUpFontSettings 332
GetPopUpHandle 334
GetPopUpIcon 328
GetPopUpItemColors 334
GetPopUpMark 327
GetPopUpRect 323
GetPopUpSelection 329
GetPopUpString 324
InsertPopUpItem 321
KillPopUp 323
LoadDialogPopUp 316
LoadPopUp 315
LoadPopUpRect 316
MovePopUp 329
MoveSizePopUp 330
MoveSizePopUpRect 330
NewPopUp 311
NewPopUpRect 315
NoPopUpColors 319
ObscurePopUp 324
OffsetPopUp 329
PopUpColors 318
PopUpDisplay 323
PopUpIcon 327
PopUpIsEnabled 326
PopUpIsVisible 324
PopUpItemCount 328
PopUpMark 326
PopUpMenu 319
PopUpStyle 328
RemovePopUp 322
RenamePopUp 325
ResNamesToPopUp 321
SetPopUpColors 332
SetPopUpFontSettings 331
SetPopUpItemColors 333
SizePopUp 330
TOOLS PLUS ROUTINES (Scroll Bars only)
ActivateScrollBar 222
AutoMoveSizeScrollBar 227
DeleteScrollBar 220
EmbedScrollBarInButton 218
EmbedScrollBarInScrollBar 218
EnableScrollBar 223
GetFreeScrollBarNum 219
GetScrollBarActionInfo 233
GetScrollBarColors 229
GetScrollBarFontSettings 228
GetScrollBarHandle 233
GetScrollBarMax 224
GetScrollBarMin 223
GetScrollBarRect 222
GetScrollBarVal 224
KillScrollBar 220
LoadScrollBar 217
LoadSpecScrollBar 218
MoveScrollBar 225
MoveSizeScrollBar 226
MoveSizeScrollBarRect 227
NewDialogScrollBar 217
NewScrollBar 214
NewScrollBarActionProc 232
NewScrollBarRect 217
NoScrollBarColors 220
ObscureScrollBar 221
OffsetScrollBar 225
ScrollBarColors 219
ScrollBarDisplay 221
ScrollBarIsEnabled 223
ScrollBarIsVisible 221
ScrollBarLineTime 230
ScrollBarPageTime 230
SetScrollBarAction 231
SetScrollBarColors 229
SetScrollBarFontSettings 228
SetScrollBarLineTime 231
SetScrollBarMax 224
SetScrollBarMin 224
SetScrollBarPageTime 231
SetScrollBarVal 225
SizeScrollBar 226
TOOLS PLUS ROUTINES (User Notification only)
PostNotification 495
SetNotification 494
TOOLS PLUS ROUTINES (Windows only)
ActivateWindow 140
ActiveWindowNumber 148
AppendDialogList 131
AutoMoveSize 153
BackdropColor 135
ClearFocus 140
CurrentWindow 141
CurrentWindowNumber 148
CurrentWindowReset 141

EditFldWindowNumber 150
FinderDisplay 153
FirstPaletteNumber 149
FirstStdWindowNumber 149
FirstWindowNumber 148
FocusWindowNumber 150
GetDialogFontInfo 144
GetDialogItemRect 144
GetFocusInfo 152
GetFreeWindowNum 135
GetWindowInOrder 147
GetWindowZoom 143
LoadDialog 127
LoadDialogList 131
LoadSpecDialog 130
LoadSpecDialogBehind 131
LoadSpecWindow 126
LoadSpecWindowBehind 127
LoadWindow 125
NoBackdropColor 135
RefreshDrawingInWindow 147
RefreshToolsPlusInWindow 146
ReplaceWindowProcID 154
SetBackdropColor 136
SetBackgroundTheme 136
SetDialogCNTLEditTextSpec 132
SetDialogCNTLListBoxSpec 133
SetDialogCNTLPopUpSpec 133
SetDialogCNTLStaticTextSpec 133
SetDialogEditTextSpec 132
SetDialogFontInfo 144
SetDialogItemRect 143
SetDialogStaticTextSpec 132
SetLiveWindowDragging 154
SetNextWindowBackgroundTheme 137
SetWindowSizeLimits 142
SetWindowZoom 142
ToolBarNumber 149
ToolBarOpen 134
WindowClose 137
WindowDisplay 139
WindowDisplayBehind 140
WindowIsActive 151
WindowIsOpen 151
WindowIsVisible 151
WindowKind 152
WindowMove 138
WindowOpen 119
WindowOpenRect 124
WindowOpenRectBehind 125
WindowPointer 153
WindowSize 138
WindowStatus 145
WindowTitle 141
WorkWindowNumber 150
Tools Plus version 526
ToolsPlus Plug-In.Lib library 59, 62, 65, 67
ToolsPlus.c source code file 59, 65, 69, 70, 71
ToolsPlus.CW6&7.68K.A4.Lib library 59, 62
ToolsPlus.CW6&7.68K.Lib library 59, 62
ToolsPlus.CW6&7.PPC.Lib library 65, 67
ToolsPlus.h header file 59, 65, 69, 70, 71, 89
ToolsPlus.Lib library 59, 62, 65, 67, 71
ToolsPlus.Lib1 library 59, 62, 69, 72
ToolsPlus.Lib1.o library 70
ToolsPlus.Lib2 library 59, 62, 69, 72
ToolsPlus.Lib2.o library 70
ToolsPlus.Lib3 library 59, 62, 69, 72
ToolsPlus.Lib3.o library 70
ToolsPlus.Lib4 library 59, 62, 69, 72
ToolsPlus.Lib4.o library 70
ToolsPlus.Lib5 library 59, 62, 69, 72
ToolsPlus.Lib5.o library 70
ToolsPlus.Lib6 library 59, 62, 69, 72
ToolsPlus.Lib6.o library 70
ToolsPlus.Lib7 library 59, 62, 69, 72
ToolsPlus.Lib7.o library 70
ToolsPlus.p interface file 62, 67, 72
ToolsPlusIsQuitting routine 434
ToolsPlusLanguage routine 537
TOOLSPLUS_ALLOWS_CSTRINGS 89
TOOLSPLUS_USES_ONLY_CSTRINGS 89
TrackBox routine 551
TrackControl routine 551
TrackGoAway routine 551

U

underline constant 144, 145, 183, 228, 260, 300, 328, 331, 349, 375
Universal Procedure Pointer 429
UnloadSeg routine 551
UnregisterAppearanceClient routine 551
UpArrowKey constant 448, 458, 460
UpdateMenuBar routine 367
Updating software 43
UpdtControl routine 551
Upgrade Notification 562
UPP 429
Upper case (shifting typed letters to) 240, 279
UseCursorTable routine 389
UseHiliteColor routine 489
UseHiliteText routine 489
User notification 493
User Pane control (see Buttons)
uses statement (Pascal) 83
UsingAppearanceManager routine 531

V

VERSION
Application 556
Minimum required for compiler 39
System 525
Tools Plus 526
Visual Separator (see Buttons)

W

Wait routine 527
WaitAvail routine 435
WaitForMultiClicks routine 435
Waiting for a period of time 527
WaitNextEvent 435
WaitNextEvent function 83
WaitNextEvent routine 551
wAllOnScreen constant 121

wAllowEditMenu constant 121
wAllowMenus constant 121
wAnimateMove constant 139
watchCursor constant 383
WatchCursorButtons routine 390
wBackgroundTheme constant 122
wCenter constant 121
wDimEditMenu constant 122
web updates 563
wFloatingKind constant 145, 152
wHidden constant 122
Window Header control (see Buttons)
WindowClose routine 137
WindowDisplay routine 139
WindowDisplayBehind routine 140
WindowIsActive routine 151
WindowIsOpen routine 151
WindowIsVisible routine 151
WindowKind routine 152
WindowMove routine 138
WindowOpen routine 119
WindowOpenRect routine 124
WindowOpenRectBehind routine 125
WindowPointer routine 153
WINDOWS 109
    Activating a window 140
    Active (determining if a window is) 151
    Active field 236
    Active Window
        Defined 113
        Making active window current 141
        Number (getting the) 148
    Appearance Manager 102, 122
    Appearance/behavior specifications 120
    Attaching a ‘DITL’ resource 131
    Automatic positioning 121
    Background theme 103, 122, 136, 137
    Centering 121
    Clicking in a zone 389
    Close box 111, 119
    Closing a window 137
    Collapsed 146
    Color
        Backdrop color (defined) 112
        Reset backdrop color for new windows 135
        Setting backdrop color for new windows 112, 135, 136
    Color resources 125, 127
    Current Window
        Defined 114
        Making a window current 141
        Number (getting the) 148
    Cursor table (using one) 389
    dctb resource 127
    Default spec for edit text items 132
    Default spec for editing fields created using a ‘CNTL’ 132
    Default spec for list boxes created using a ‘CNTL’ 133
    Default spec for pop-ups created using a ‘CNTL’ 133
    Default spec for static text fields created using a ‘CNTL’ 133
    Default spec for static text items 132
    Dialog item’s display rectangle 143, 144
    Dialogs in Tools Plus 127
    DITL resource 127, 131
    DLOG resource 127, 130, 131
    Don’t protect GUI elements during doRefresh event 122
    Drag/resize windows in real time 103
    Drawing 109
    Edit menu
        Access from modal window 121
        Disabling the items 122
    Editing Field window
        Defined 114
        Number (getting the) 150
    Event handler for a window 429
    Floating Palettes 113, 120
        Adding to your project 122
        Closing a floating palette 137
        Hiding palettes 122, 139, 140
        Infinity Windoid 155
        Moving a palette 138
        Opening a floating palette 119
        Position in window list 115
        Showing palettes 139, 140
        Size (changing the) 138
        Window number (getting the) 149
    Font settings in dialogs 144
    Frontmost standard window number (getting the) 149
    Frontmost window number (getting the) 148
    Height 145
    Hidden (determining if a window is) 151
    Hiding a window 122, 139, 140
    Inactive (determining if a window is) 151
    Inactive window (usage of) 141
    Keyboard focus window 150
    Kind (determining a window’s kind) 152
    Layers 115
    Live dragging/resizing (turning it on or off) 154
    Location 145
    Manual BeginUpdate/EndUpdate use 122
    Manually refreshing application-drawn elements 147
    Manually refreshing GUI elements 146
    Maximum number of windows 99
    Menu (Edit menu affected by windows) 355
    Menu access from modal window 121
    Modal/Modeless window 115, 119
    Moving a window 138
    Moving onto screen 121
    New window (opening a window) 119, 124, 125
    Open (determining if a window is) 151
    Opening a window 119, 124, 125
    Opening using a ‘DLOG’ resource 127, 130, 131
    Opening using a ‘WIND’ resource 125, 126, 127
    Order from front to back 147
    Pointer (getting a window’s pointer) 153
    Refresh when opened 122
    Reset window operations to the “active” window 141
    Showing a window 139, 140
    Size (changing the) 138
    Size box 112
    Sizing limitations 142
    Standard windows 113
    Status of a window 145
    Styles
        Desk accessory 120
        Drop-shadow dialog 120
        Fixed size document 120
        Floating palette 120

Growable document 120
Movable modal window 120
Plain dialog 120
Standard dialog 120
Substituting a ProcID throughout your app 91, 102, 118, 154, 530, 531
Suppressing zoom lines 122
Text 109
Tiling 121
Title changes 141
Tool Bar 113
Closing a tool bar 137
Height (changing the) 138
Hiding the tool bar 122, 139, 140
Opening a tool bar 134
Position in window list 115
Showing the tool bar 139, 140
Toolbar inside a window 134
Window number (getting the) 149
Type (determining a window's type) 152
Unused window number (getting the) 135
User item's rectangle in a dialog 143, 144
Visible (determining if a window is) 151
wctb resource 125
Width 145
WIND resource 125, 126, 127
Work Window
Defined 114
Getting the work window number 150
Zoom box 111
Zooming
Including a zoom box 122
Standard co-ordinates
Getting standard co-ordinates 143
Setting standard co-ordinates 142
User co-ordinates
Getting user co-ordinates 143
Setting user co-ordinates 142
WindowSize routine 138
WindowStatus routine 145
WindowTitle routine 141
wManualUpdate constant 122
wNoKind constant 145, 152
wNoOffScreen constant 121
wNoZoomLines constant 122
wOffsetForToolBar constant 139
Word wrap 240
WorkWindowNumber routine 150
wPalette constant 120, 122
wRefresh constant 122
Writing in windows 109
wStandardKind constant 145, 152
wTile constant 121
wToolBarKind constant 145, 152
wUnprotectedRefresh constant 122

Y

YesAltBut constant 502
YesNoAlert constant 502
YesNoCanAlert constant 502

Z

Zones (see Cursors)
Zoom lines (drawing them) 528
ZoomAcross constant 529
ZoomBox constant 122
ZoomIn constant 529
ZoomLines routines 528
ZoomOut constant 529
ZoomWindow routine 551

Please send your comments regarding Tools Plus
and other Water's Edge Software products to:

Water's Edge Software
2441 Lakeshore Road West, Box 70022
Oakville, Ontario, Canada L6L 6M9

Phone: 1-416-219-5628
Fax: 1-905-847-1638

Email: WaterEdg@interlog.com

visit our web site at:
http://www.interlog.com/~wateredg

Water's Edge Software offers a **free electronic forum** where Tools Plus developers can meet, discuss issues, and exchange information. If you would like to talk with these developers or just lurk to see what's happening, send an email to TPDevLst@interlog.com and we'll send you more information about the forum and how to get on it.