DDA

Documentation **D**éveloppeurs **A**pple.

D.D.A. distribue en France les systèmes de développement Apple ainsi que les documentations techniques Apple, non disponibles au travers du réseau de revendeurs Apple.
D.D.A. est ouvert à tous, sans adhésion préalable et sans nécessité de contrat de développement Apple.
D.D.A. comporte trois branches se répartissant les activités suivantes :

D.D.A. Informations/Support

Documentation Développeurs Apple®
I.Development
"Technopole", rue M.Faraday
78180 Montigny le Bretonneux
(1) 30 45 26 62 : Macintosh
(1) 48 56 89 27 : Apple II

D.D.A. Commandes

Documentation Développeurs Apple®
Prim'Vert (Etablir les chèques à l'ordre de Prim'Vert)
36 rue des Etats-Généraux
78000 Versailles
(1) 39 02 33 44

En outre, pour tous renseignements techniques relatifs à la documentation, aux orientations et au choix des informations, classeurs ou disquettes à acquérir dans le domaine Apple IIGS, trois moyens de liaison sont à votre disposition :

Télétel 2
Code **APPLE**
ouvert à tous
Interlocuteur : DDA

Calvacom
par abonnement
Interlocuteur : DDA

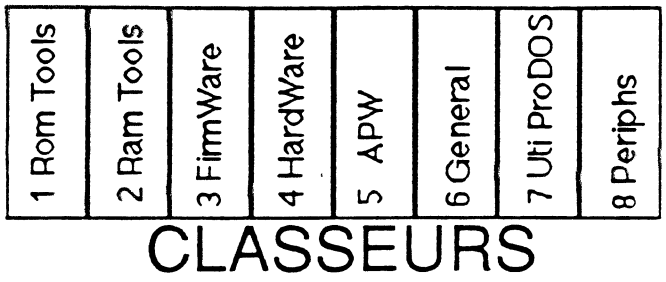
Téléphone
Hot-Line et Répondeur
Ouvert à tous
(1) 48 56 89 27

Documentation Développeurs Apple IIGS, au 17 mai 1988

Photocopies des documents originaux américains (écrits -donc en Anglais- par les constructeurs hardware et les auteurs du firmware eux-mêmes) classés par centre d'intérêt et rassemblés en plusieurs classeurs.

Date du précédent document : 9 avril1988.

Mises à jour signalées par *. Nouveaux éléments signalés par **.



Volume 1 - Les nouveaux outils

310 ~~200F~~

	Note Synthesizer	00:50	25.09.86	
*	Note Sequencer	~~1.0B~~ 1.1.P	~~02.02.88~~ 26.8 ~~65 F~~	75F
**	A.C.E	~~00.14~~ 01.12	~~04.01.88~~ 6.8 45 F	
**	MIDI Tool Set	~~1.7~~ 1.1.4	~~13.01.88~~ 13.06.88 ~~80 F~~	90F

Volume 5 - L'environnement APW, ApwA/C 624F

*	APW : Apple IIGS Programmer's Workshop	Apda Draft	27.08.87
*	Assembler Reference : APW 1.0	Apda Draft	6.08.87
*	C reference : APW 1.0	Apda Draft	25.08.87
	Workshop Pascal ERS. Compiler conventions	0.01	6.02.86
*	Debugger reference	Apda draft	20.08.87

Volume 6 - Interface Utilisateur, Bases et 65816 588F

	Human Interface Guidelines	Final Draft	1.12.86
	Technical Introduction to the Apple IIGS	Final Draft	20.08.86
*	Programmer's introduction to the Apple IIGS	Final Draft	09.09.87
	Programming the 816		1.05.85
	Description of 65816 control instructions		5.06.85
	65816 addressing mode & bank crossing		19.11.85
	Preserving stack when changing modes		24.01.86
	Numeric Magnitude comp on IIGS	rev 2	20.11.85

560 F

☌ Volume 7 - Utilitaires, TechNotes et Systèmes 396F

	Finder Demo explanations	version 1	21.01.87
•	Icon Editor		01.09.87
	Compiler conventions for Orca/M		14.01.86
	Paint : Load File Source	Final	2.03.87
	Graphics Image File Format STANDARD	Final	26.02.87
•	Apple IIGS Sampled Instrument Format	1.02	05.05.87
	FontMunger	2.0	29.08.86
	ReadBin User's Guide		29.08.86
	Passport User's Guide	Preliminary Draft	10.09.86
	Memory Mangler ; Loader Dumper	APDA Draft	27.10.86
	TechNote : Window Information Bar Use		3.09.86
	TechNote : Changing graphics mode		5.09.86
	TechNote : Window titles		17.09.86
	TechNote : Cloning strings		22.09.86
	TechNote : Pen Pattern Data Structure		15.12.86
	TechNote : Halt Mechanism in SANE		15.12.86
	TechNote : Elems Functions in SANE		
•	TechNote : InvalRgn twist		2.04.87
•	TechNote : Ensoniq doc swap-mode anomaly		2.04.87
•	TechNote : Toolset Interdependencies		1.08.87
•	TechNote : Modem Firmware bug		2.04.87
•	TechNote : SFile & Grafports		1.06.87
•	TechNote : InstallFont and big fonts		1.06.87
•	TechNote : BackGround printing		1.06.87
•	TechNote : User ID		1.06.87
•	TechNote : SCC Interrupts		1.06.87
•	TechNote : Multichanel output		1.06.87
•	TechNote : APW Languages Numbers		1.08.87
•	TechNote : DMA Compatibility for Exp. RAM		1.08.87
•	TechNote : 2.0 ROM Revision Summary		1.10.87
••	TechNote : Unload Dynamic Segments		1.10.87
••	TechNote : Note Synthesizer Timer's Use of DOC RAM		1.10.87
••	TechNote : Graphic Images File Formats (NEW)		1.10.87
••	TechNote : REDIRECTing Output from C		1.11.87
	System Disk Release 1.1	1.1	19.12.86
••	System Disk Release 2.0	2.0	10.06.87
••	System Disk Release 3.1	3.1	1.09.87
••	System Disk Release 3.2	3.2	06.88

100F

Sound Tools 65F

Defining your windows 60F

☌ Volume 8 : Périphériques courants 276F

AppleTalk Filing Protocol	1.0/2	12.11.85
SCSI Card Reference Manual	Alpha Draft	1.06.86
LaserWriter Reference	Beta	7.07.86

☌ Apple IIGS BASIC — Avec 1 disque — 444F

••	Apple IIGS BASIC	Apda Draft	11.09.87

☌ Firmware reference manual — Addison-Wesley — 280F

☌ Hardware reference manual — Addison-Wesley — 280F

☌ ProDOS 16 reference manual — Addison-Wesley — 305F

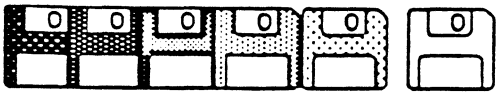
System Loader inclu. (**Avec** 1 disque : Exerciser).

☌ ToolBox reference manual — Addison-Wesley — 305F

Volume 1.

☌ ToolBox reference manual — Addison-Wesley — 305F

Volume 2.



DISQUES

♁ 1 Apple IIGS Programmer's WorKshop : 4 disques 296F

**	Apw	1.0	31.08.87
**	ApwA 65816	1.0	31.08.87
**	ApwC	1.0	31.11.87
**	ApwU utilities et debugger 1.1.1 du 14.09.87	1.0	31.08.87

Description :
APW est un environnement de programmation cohérent et extensible, comprenant un shell, un linker, des utilitaires, des librairies et un certain nombre de langages. Sont livrés l'Assembleur Macro 65816, le compilateur C, et les librairies associées. Le système de développement Apw s'impose naturellement comme le standard de programmation sur le II GS, on peut lui adjoindre d'autres langages comme par exemple le Compilateur TmlPascal (importé par PIngénierie), ou le Compilateur Orca Pascal (importé par Réseau Planétaire). La licence d'utilisation du Compilateur Megamax C, de 150F, est comprise dans le prix du package.

♁ 3 SAMPLES : 6 disques 216F

**	Apple II Demo Sampler (disk)	V1.2	03.04.87
**	Apple II Demo Sampler (data)	V1.2	03.04.87
**	Apple II Demo Sampler (sources asm)	V1.2	03.04.87
	Samples : DeskAccs, Simp...		19.12.86
**	Samples MovePic, DemoStuff, Brick...	V1.2	03.04.87
**	Hodge.Podge Asm/C/TmlPascal	1.0B1	20.08.87

Description :
- Une application complète en mode natif (Demo.sys16), en un disque programme, un disque d'images pour le slide.show et un disque comprenant les sources asm des diverses applications : Daleks, Color Dabbler, HodgePodge, Slide Show, QuickDraw Sampler, un Paint en mode 640, Brickout et Keyboard Sounds.
- Samples du 19.12 : Sources en assembleur et applications stand.alone ; Color, Daleks, HodgePodge, Simp (synthe), Seq (Sequencer), Clock.Nda, ScreenDump.Cda
- Samples du 03.04 : Sources en assembleur et applications ; Demo, Hp, Color Dabbler, Brick Out, Daleks, Move.Pic, Simp
- New HodgePodge : Application complète, écrite et réalisée par les auteurs mêmes de la Boîte à Outils, décrivant la gestion idéale des fenêtres, menus, icônes, pictures, fonts. Utilise plus de 80% des outils (y compris Printer Mgr et Font Mgr). Les souces sont livrés à la fois en C, TmlPascal et Assembleur, sur le même disque. Sa description complète se trouve dans le document Programmer's Introduction to the Apple IIGS, classeur n°6.

♁ 4 Systèmes & Tools : 2 disques 72F

*	Disque Système 3.1, Icon Editor	3.1 Fr	01.11.88
**	Disque Système 3.2	3.2	[illegible] 06.88

Description :
Ce sont les références en matière d'outils Ram Tools. Le système 3.1 est le système français à jour. Le système 3.1 permet d'utiliser le Finder GS, livré sur la même disquette. Une application complémentaire, Icon Editor, est destinée à la réalisation d'icônes spéciales pour les documents et les applications, ainsi qu'à donner le lien entre ceux-ci, permettant de lancer un programme en ouvrant l'un de ses fichiers.
Le système 3.2[illegible], outre le support des nouveaux outils 29 et 32, permet une utilisation performante du reseau AppleShare.

☝ 5 Utilitaires : 3disques 108F

- • Macintosh utilities (Passport 1.0A2, MacToGS, ReadBin, FontMunger...)
- •• GS Utilities, Pic Convert, DA, Convert, Exerciser... 1.06.87
- •• Public Domain et sources 1.06.87

Description :
- *Macintosh Utilities* fonctionne sur Mac 512/800, Mac+ ou Mac SE. Il réunit divers utilitaires permettant le transfert de documents entre Mac et GS et inversement, la conversion d'images dans les deux sens, la transformation de polices de caractères Mac en Fonts au format d'Apw, de QuickDraw II ou du Font Manager. Propose aussi deux Editeurs graphiques pour Macintosh.
- *GS utilities* rassemble plusieurs CDA, plusieurs NDA, des accessoires pour MouseDesk, quelques sources, des exemples SystemSetup, Filer, Convert, Exerciser P8 et P16, et une application Public Domain de conversion d'images HGR, DbHGR, Atari et Mac (ShrConvert)
- *Public Domain et sources* rassemble de très nombreux et très variés exemples réalisés par les développeurs français qui les proposent (Panel.setup, Cda, Nda, Patches,Astuces, Various tricks...)

☝ 6 BASIC : 1 disque Non vendu séparément

- •• Apple IIGS BASIC 1.0B4 14.09.87

Description :
Basic 65816 ressemblant à l'Applesoft, avec procédures, gestion mémoire étendue et appels toolbox. Accompagne le manuel associé.

Documentation Développeurs
Apple II, au 9 avril 1988

ProDOS

ProDOS Tech Reference Manual (+disque)	305Fr
Basic Programming with ProDOS (+disque)	350Fr

SANE Mathematics

Apple Numerics Manual	360Fr

Periphériques

ImageWriter II Technical Reference Manual	320Fr
PostScript Language Reference Manual	280Fr
PostScript Language Tutorial and Cookbook	210Fr
** Apple CD-ROM Developper Notes	100Fr

Apple IIe / Apple IIc

Apple II Instant Pascal Reference Manual	320Fr
Applesoft tutorial	385Fr
Applesoft Basic programmer's Reference Manual	315Fr
** Apple II Technical Notes 1985, 86, 87	144Fr
Apple IIc Technical Reference Manual	290Fr
Apple IIc Programmer's guide to the 3.5 ROM	252Fr
Revised Apple IIe Technical Reference Manual	280Fr

oOo

L'ensemble de la documentation, Apple II, Apple IIGS et Macintosh est disponible auprès de DDA.

2) J'ai trouvé - 52 l.
De: Support Dev. apple2 APPLE (SDA10) - 24 sep 87 11h27

Rebonjour.

J'ai testé pour vous la fonction GetCOrigin (GetContentOrigin, qui est censée donner les coordonnées relatives (par rapport à la DataArea) du point de coordonnées locales 0,0. Ceci permet d'optimiser la quantité de texte à tenter d'envoyer vers la fenetre.
Comme je n'y comprend RIEN en C, voici la solution en assembleur.

```
;........Trouver le C origin de la Data Area.....................

         PushWord #0                    Space for result
         PushWord #0                    idem
         PushLong MyTextWindow          Hdl donné par _NewWindow
         _GetContentOrigin

         Pla                            Dans A (integer), les Y
         Sta 0                          Low

         Pla                            Dans A (integer), les X
         Sta 2                          Hi

;........Faire 2 strings.....................................

         PushWord 0
         PushLong #StrY                 addr de la chaine objet
         PushWord #4                    char max
         PushWord #0                    sans signe
         _Int2Dec

         PushWord 2
         PushLong #StrX                 addr de la chaine
         PushWord #4                    char max
         PushWord #0                    sans signe
         _Int2Dec

;........Ecrire les strings .................................

         PushWord #10
         PushWord #10

         _MoveTo

         PushLong #StrX
         _DrawString
         PushLong #StrY
         _DrawString

         End

StrX     Ds 6                           un peu de place pour la chaine
StrY     Ds 6                           Idem
```

Apple IIGS System Disk 3.2 Release Notes

Contents

Operating System

BASIC.SYSTEM v1.2

(The ProDOS AppleSoft Command Interpreter)

Bug fixes:

There were dispcrepancies between TOTENT, the word specifying the number of entries in the ProDOS directory, and the actual number of entries. This has been fixed by making the following changes:

CATALOG in <CMDS2>, B076	changed from BEQ ENDCAT to NOP/NOP
B07C	changed from BCS CCATERR to BCC
B0A1	changed from BPL to BCC

Control S on a CATALOG prematurely terminated if a subsequent <space> was pressed. This has been fixed.

ProDOS 8 v1.6 (P8)

Bug fixes:

Previously, on occasion, the mliactiv flag was left on after finishing an mli call. To correct this problem, and 'asl' has been changed to an 'lsr'.

ProDOS 8 now returns "*POSNERR" if the 32 MB limit is exceeeded.

"ESC" cleared from the keyboard strobe when booting on a IIc.

Interrupts are turned off during a status call.

ProDOS 16 v1.6

A number of pieces of code have been added to ProDOS 16, making it possible to boot over an AppleTalk network.

SetUp and DeskAccs will not be loaded if bit 15 of their AUX_TYPE is set.

Get_Dir_Entry will refresh its directory buffer if it is precede by any other P16 call.

A call to FORMAT will now cause a switch to emulation mode before calling the SCSI driver. Previously a call to FORMAT when in native mode caused the system to hang.

In order to be more descriptive, Get_Dir_Entry now returns End-of-File errors ($4C) instead of End-ofDirectory ($61).

If CREATE is called with an invalid pathname, it now returns Path-Not-Found instead of File-Not-Found.

System Loader v1.6

Bug fixes:

There were several bugs which interfered with proper operation of the Loader. These include:

- A problem in the buffering scheme which made it impossible to load some files with large segments
- Improper alignment of a load segment with an align of exactly $10000
- Inability to load a segment of exactly one byte
- Incorrect returns from LoaderShutDown, LoaderVersion, and LoaderStatus (carry bit clear, accumulator non-zero)
- Subroutine Dispatcher wrote two bytes into random memory
- Loader's buffers moved during Restart without informing the Loader

Setup Files

AppleTalk INITs v1.0

There are eight AppleTalk INIT files with System Disk 3.2. These are:

ATINIT	Contains user configuration information
ATPATCH	Implements the RPM, PAP, EP, and ZIP AppleTalk protocols. Also contains the code needed to patch the bugs in ROM. A list of patches in this version follows.
SPLoad	Implements ASP, which allows up to eight open network sessions at once
PFILoad	Implements the ProDOS Filing Interface, which allows ProDOS calls to be made transparently over the network
ATStart	Starts AppleTalk
ATSetup	Determines if AppleTalk is active, and loads appropriate files
ATROM	Contains new code replacing ROM AppleTalk
ATResponder	Registers user-name on the AppleTalk network

Tool.Setup v2.3

The current version of Tool.Setup is actually three files, while on previous releases it was only one. The three files are TOOL.SETUP, TS1, and TS2. TOOL.SETUP examines the version of the ROM to determine which of the other setup files to load. It loads TS1 with the original ROM, and TS2 with the newer ROM. If it finds a ROM that is not one of the first two versions, it will load neither file. This change saves between 8 and 10 seconds of boot time.

TS1 and TS2 also contain patches to the SysBeep routine so that if the control panel volume is zero, a SysBeep will flash the screen border.

Toolbox

Audio Compression and Expansion Tool Set

The A.C.E. Tool Set provides a fast, reliable set of routines for compressing and decompressing digitized sounds. For detailed information on how to use the A.C.E. Tools, see the A.C.E. Tools ERS which accompanies these release notes.

Control Manager v2.5

Controls are now drawn in the port of the window in which they appear. This undoes a change introduced in version 2.1 of the control manager.

This rule has always been true, but needs to be clarified:

> Controls are drawn in the grafPort of the window that owns the controls. This means that the appeareance of the controls is determined by the state of the window's grafPort.For example, if the grafPort's font is changed to something other than the system font, then text in controls will be drawn in the window's font, not in the system font.
>
> This also means that the grafPort cannot be left in any old state, because the state of the grafPort will determine how controls are drawn. For instance, if the pen mask is changed, then controls will be drawn using the new mask. Thus, if you are drawing in a window which has controls, make sure the grafPort is left in the state in which you want your controls drawn
>
> Controls which are created by TaskMaster are drawn in the Window Manager's port, not in a window's. Therefore, the state of a window's grafPort will not affect controls drawn by TaskMaster.

The font ID is saved and restored when switching to and from the icon font.

Colors in control tables now use all four color bits in both modes; they formerly used only bits 0 and 1 in 640 mode. There shouldn't be any applications using color controls in 640 mode, but for any that do the effect will be that controls will be a different color.This change was made so that dithered colors can be used with controls.

The barArrowBack entry in the scroll bar table was never implemented as first intended, and is now no longer used. It was intended to allow the arrow to be drawn with three colors: the interior of the arrow, the arrow's outline, and the surrounding color. Since arrows are drawn as text characters, only two colors are possible.

The high word of the parameter passed to the draw command is now set to zero in some cases, to help some code that did not conform to documented requirements.

The Control Manager will preserve the current port across calls to the Control Manager, including those that are passed through other tools, such as the Dialog Manager.

The Control Manager now calls PenNormal before drawing controls in a window. This may affect the state of the window's grafPort.

The Control Manager now saves and restores text mode across calls to the Control Manager which affect standard controls. Custom controls may still change text modes without a savwe and restore.

From now on the Control Manager will not change the following fields in the port of a window that contains controls:

bkPat	background pattern
pnLoc	pen location
pnSize	pen size
pnMode	pen mode
pnPat	pn pattern
pnMask	pen mask
pnVis	pen visibility
fontHandle	handle of current font
fontID	ID of current font
fontFlags	font flags
txSize	text size
txFace	text face
txMode	text mode
spExtra	value of space extra
chExtra	value of char extra
fgColor	foreground color
bgColor	background color

When a the enclosing RECTs of radio button and check-boxes were larger than the icons, the controls' titles were not vertically centered. This has been fixed.

EraseControl expands the bounds RECT of certain controls before erasing. This is because a simple button has a surrounding border when it is drawn bold (when it is the default button), and may have a drop shadow. These characteristics are drawn outside the bounds RECT of the control, so EraseControl expands the bounds RECT to erase them correctly. Formerly, EraseControl was expanding the bounds RECT in cases where it was not necessary. This has been fixed.

The Control Manager will now use the state of the window port to compute the size of control bounds RECTs when creating a control. Previously some of the state information was saved from startup time.

The SpecialRect call was added to QD AQux to replace calls to FrameRect and FillRect. The Control Manager now calls SpecialRect if it is available, instead of making separate calls to FrameRect and FillRect.

TrackControl now pushes an extra word on the stack before calling the action procedure. This word is provided only because some action procedures return a value where none is expected, and should not be used by an application.

When drawing simple buttons and when drawing page regions in scroll bars, the Control Manager was not using the outline color from the controls' color tables. This has been fixed.

If a radio button was turned on, but was hidden, the Control Manager would erroneously redraw it if another button in the family was turned on. This has been fixed so that the hidden button remains hidden instead of being redrawn as it is turned off.

When the Control Manager called application code, such as a custom control defProc, the call would not work if the code was located at the first byte of a bank. This has been fixed.

Bug fixes:

TestControl will now return a zero if an invisible or inactive control is selected.

MoveControl no longer makes invisible controls visible when moving them.

DisposeControl returned an incorrect error code. This has been fixed.

SetCtlTitle did not redraw the titles of radio buttons and check-boxes. This has been fixed.

The icons for radio buttons, check-boxes, and arrows in scroll bars were not drawn correctly if the item was not as high as the icon to be drawn. This has been fixed.

CtlStatus, CtlVersion, and CtlBootInit now return zero in the A register.They formerly returned non-zero values with the carry bit clear.

HiliteControl formerly returned an incorrect error code if it was called with a control was hilight flag was already set to the given value. This has been fixed.

GetCtlRefCon and SetCtlRefCon formerly returned an incorrect error code if a null handle was passed. This has been fixed.

HideControl, EraseControl and ShowControl formerly returned an incorrect error code if the specified control was already visible. This has been fixed.

DrawControl formerly returned an incorrect error code if the window awas visible. This has been fixed.

The grow box control now has its own color table. Formerly this control shared the simple button default color table, which had the effect that a hilighted size box was drawn as a single black box. The hilighted grow box now appears as a white icon in a black box.

Previous grow box default color table:

$0000	Black outline for box
$00F0	Not hilighted: black outline in white interior
$0000	Hilighted: black outline in black interior

New grow box default color table:

$0000	Black outline for box
$00F0	Not hilighted: black outline in white interior
$000*F*	Hilighted: *white* outline in black interior

The color table for the size box control in the *Apple IIGSToolbox Reference* is incorrect. The correct table follows, with information that was omitted from the reference in boldface:

growOutline WORD Color of size box's outline:
Bits 8-15 = zero
Bits 4-7 = outline color
Bits 0-3 = zero

growNorBack WORD	Color of interior when not hilighted Bits 8-15 = zero Bits 4-7 = background color Bits 0-3 = icon color
growSelBack WORD	Color of interior when hilighted Bits 8-15 = zero Bits 4-7 = background color Bits 0-3 = icon color

Desk Manager v2.4

Bug fixes:

SystemTask used CLI instead of SEI to access certain data with interrupts off.

The GetNumNDAs now works correctly.

The SystemEvent call now allows CDAs to be activated while an NDA window is active on-screen.

Dialog Manager v2.2

Bug fixes:

GetItext formerly always stored at least 3 bytes of data into the resultPtr passed to it. This was a problem if the editline item was only one character, because there should only be two bytes stored: one for the lngth, and one for the character itself. This has been fixed.

GetNewModalDialog no longer crashes when passed a non-zero refcon value.

IsDialogEvent now correctly claims all window control events.

HideDITem, ShowDItem, GetDItemValue, EnableDItem and DisableDItem no longer crash with invalid item IDs.

There was a minor problem with paramtext characters (^0 through ^3). When they were used at the end of a line, garbage characters were appended. They will now work correctly.

Several Dialog Manager calls failed if given invalid item IDs. This has been fixed. The calls affected are:

SetDItemValue
GetDItemType
SetDItemType

DialogStatus formerly returned a value of 'active' after the Dialog Manager had been shut down. This has been fixed.

Certain Dialog Manager and LineEdit calls assumed that foreground and background colors in the applicable grafPort were correctly set. This is actually only true if other color controls have previously been drawn. This problem has been fixed. The affected calls are:

StatText
LongStatText
LineEdit drawing routines.

Font Manager v2.3

The current version of the Font Manager incorporates several changes. In previous verisons, FMStartUp opened each font file in the FONTS folder, and constructed lists of information for all avialable fonts. These lists contained font IDs, font names, and so forth for every font in every file in the FONTS folder.

The present version of the Font Manager does this same work the first time it starts up, but caches all the information it compiles in a file called FONT.LISTS in the FONTS folder. The next time the Font Manager starts up, it checks all the font files' creation and modification dates and times against the information in FONT.LISTS. Only if it finds new font files or other evidence of change will it compile new FONT.LISTS information. Otherwise, it simply starts up with the information stored in the LISTS file. In most cases, since it doesn't have to open every font file, the Font Manager can start up much more quickly.

New call:

$1C1B InstallWithStats

returns: none Input parameters:

desiredID: LONG

scaleWord:WORD

ResultPtr: LONG

Installs a font and returns information about that font. When an application requests the installation of a font, the Font Manager attempts to install the requested font, but it may not be available. In such cases, the Font Manager will install the closest match it can find to the requested font.

InstallWithStats installs a font just as if the application had called InstallFont, but it returns a FontStatRec in the buffer pointed to by ResultPtr. This record contains the ID of the installed font, which may be different from the font requested. It also contains the purge status the font had before it was installed. Since purge status can be changed by installation, this information can make it easier to restore a font's purge status. The font's purge staus may not be the same after installation. If you need to know an installed font's purge status, use FindFontStats.

Bug fixes:

A handle to the parameter block was not locked down at the correct point during the call to FixFontMenu. As a result, memory compaction caused the parameter block to move, and invalidated the handle to the parameter block. This has been fixed.

Handles were not being locked down during scaling routines. This has been fixed.

Line Edit v2.1

Bug fixes:

LETextBox2 now locks its handles to prevent the allocated blocks from being moved during the call's execution.

List Manager v2.3

The IIGS Toolbox Reference incorrectly states that a disabled item of a list cannot be selected. In fact, a disabled item **can** be selected, it simply may not be hilighted. The List Manager provides the ability to selected disabled (un-hilighted) items so that it is possible, for instance, to allow a user to select a disabled menu choice as part of a help dialog. There may be other reasons to allow a user to select a disabled item, as well, so the List Manager will allow it.

Further clarification:

List manager definitions:

disabled:	Bit 6 of the list-item's memFalg field is set. Disabled items appear dimmed and cannot be hilighted.
enabled:	Bit 6 of the list-item's memFalg field is clear. Enabled items appear normal and can be hilighted.
selected	Bit 7 of the item's memFlag field is set. This bit is set when a user clicks on the list-item, or the item is within a range of selected items. A selected item will only appear hilighted if it is also enabled.
hilighted	A member of a list will only appear hilighted when it is both selected and enabled. This means that bit 7 of the memFlag field is 1 and bit 6 is 0. A hilighted member is drawn using the highlight colors.

Bug fixes:

The List Manager uses a custom control defProc. The Control Manager passes a handle to this defProc for the 'record size' operation. The List Manager was locking the passed handle, but not checking to ensure that it was valid before using it. As a result, changes were being written to unknown locations in memory. This has been fixed.

The List Manager now correctly handles member record arrays larger than 64K.

The List Manager was updating lists incorrectly when the thumb was dragged. The number of memebers scrolled times the height of the members was greater than 64K pixels. This has been fixed.

Member text is now drawn in 16 colors in both 320 and 640 mode.

MewList sometimes updated the list scroll bar incorrectly. This has been fixed.

Memory Manager v2.1

New Call:

$2F02 **RealFreeMem** returns: Long

No inputs

Returns the number of bytes in memory that are free, plus the number that could be made free by purging. FreeMem only returns the number of bytes which are actually free, ignoring memory which is occupied by unlocked purgeable blocks, so RealFreeMem provides a more accurate picture of available memory.

Bug fix:

During the call SetHandleSize, if memory was compacted while its handle was resized, and the original handle moved before the new memory was found, and if another handle was moved into the original handle's old location, then the data in the resized handle was incorrect. In order to correct this problem, the Memory Manager now dereferences the original handle after compaction.

Menu Manager v2.1

The NewMenuBar call will now automatically set bit 31 of the CtlOwner field in the menu bar record, if the designated menubar is a window menu bar, and the value passed for the window is not zero.

The menu manager's justification procedures now adjust for menu bars in windows. Menus will be moved to the left if they would otherwise appear to the right of the menu bar's right end.

The default menu bar has the following coordinates: top = 0; left = 0; height = 13; width = the width of the screen.

The Menu Manager now locks menu and menu bar handles during any call that accesses those records. This prevents the records from being moved during accesses.

Empty menus are now drawn with the QD Aux call SpecialRect, unless it is unavailable. In this case, the calls FillRect and FrameRect are used.

Some Menu Manager calls to an application did not work if the routine being called was located at the first byte of a bank. This has been fixed.

MenuShutDown no longer returns an error if the Menu Manager has already been shut down.

The call CalcMenuSize uses the parameters newWidth and newHeight to compute the menu's size. These parameters may contain the width and height of the menu, or may contain the values $0000 or $FFFF. A value of $0000 tells CalcMenuSize to calculate the parameter automatically. A value of $FFFF tells it to calculate the parameter only if the current setting is zero.

The effect of all three uses:

1) Pass the new value:	The value passed will become the menu's size. Use this method when a specific menu size is needed.
2) Pass $0000	The size value will be automatically computed. This is useful if menu items are added or deleted, rendering the menu's size incorrect. The Menu's height and width can be automatically adjusted by calling CalcMenuSize with newWidth and newHeight equal to $0000.
3) Pass FFFF	The width and height of a menu is zero when it is created. FixMenuBar calls CalcMenuSize with newWidth and newHeight equal to $FFFF to calculate the sizes of those menus with heights and widths of zero.

Using SetMenuBar to change the menu bar color now changes the dynamic default menu color table and leaves the static menu color table unchanged. If SetMenuBar is called to change the color of a menu which uses the default color table, then the colors of all menu bars which use the default color table will be changed. To change the color of a single menu bar without affecting other menu bars, set the menu bar's CtlColor field to a new color table.

Menu Caching

This version of the Menu Manager introduces new Menu Caching features. Menu caching is designed to provide faster display of menus under certain circumstances. When a menu is drawn on the screen, the area of the screen that it covers is copied into a buffer. Then, when the menu goes away, the contents of the buffer is simply copied back to the screen.

In the current version of the Menu Manager, when the saved screen image is copied back to the screen, the menu that goes away is copied into the buffer. In other words, the Menu Manager swaps the menu image with the screen image. That way, the next time that menu is pulled down, the Menu Manager can copy it from the buffer instead of drawing a new image.

Of course, if the menu image changes, for example, if an item is disabled, or the items on the menu change, then the cached image is inaccurate, and the Menu Manager must redraw the menu. Nevertheless, in those cases when a menu image does not change, the menubar will respond to the user more quickly.

Table 1

Calls that can change a menu image

CalcMenuSize	CheckMItem	DeleteMItem	DisableMItem
EnableMItem	FixMenuBar	InsertMItem	MenuNewRes
SetBarColors	SetMenuFlag	SetMItem	SetMItemFlag
SetMItemMark	SetMItemName	SetMItemStyle	

Menu caching should not increase memory requirements, since menu images are purgeable when not displayed on the screen.

This menu caching scheme should work properly with all existing standard menus. Custom menus, however, will have to be altered to work correctly with menu caching. Custom menus will still function normally, as long as they do not change the menu record directly, they just will not be able to take advantage of the menu caching scheme to speed up display.

Caching does not work with menus in windows, so the InsertMenu call automatically disables caching for such menus.

Caching with custom menus

Bit 7 of the MenuFlag field in a menu record indicates whether a menu's definition procedure knows about caching. A value of 1 indicates that the menu in question works correctly with caching. A custom menu which uses caching must define a menu record which sets this flag, and allocates an extra field, a handle to the cache in which the menu image will be stored. (See Table 2)

Table 2

Fields in a cacheable menu record

MenuID	WORD
MenuWidth	WORD

```
MenuHeight      WORD
MenuProc        LONG
MenuFlag        BYTE        ; Bit 7 = 1 to enable caching
MenuRes         BYTE
FirstItem       BYTE
NumOfItems      BYTE
TitleWidth      WORD
TitleName       LONG
MenuCachr       LONG        ; New field in cacheable menu records
                            ; Handle to cache
```

The FixMenuBar call will automatically allocate a cache for the defined menu if the caching flag is set.

Bug fixes:

CalcMenuSize has been modified so that it takes text styles into account when calculating the width of a menu.

Display of menu titles has been corrected so they will be arranged properly relative to the left side of the menu bar RECT.

The CtlOwner flag of a menu inside a window must be negative (bit 31 must be set). This has always been true, but has been undocumented. NewMenuBar now sets that flag if the value passed for the window is not zero, and the menu bar is a window menu bar.

Menus in windows can now display the Apple character (ASCII $14).

Menus now use their outline color for lines which separate menu items.

MIDI Tool Set

The MIDI Tool Set is a set of utilities for use with the Musical Instrument Digital Interface data transfer protocols. With a MIDI interface and the correct driver, the MIDI Tools enable a developer to write applications which can control MIDI-compatible synthesizers. For a detailed explanation of how to use the MIDI Tools, see the MIDI Tools ERS which accompanies these release notes.

Known bugs:

If you use the Note Sequencer with the CARD6850.MIDI driver, either the Note Sequencer or the driver will crash, because they step on each other's memory. This problem will be fixed with a patch to the driver which is available from Developer Tech Support.

Miscellaneous Tools v2.1

Bug fixes:

The SetHeartBeat call sometimes allowed heartbeat interrupts to accidentally clear the quarter second timer interrupt. SetHeartBeat has been patched to install a different heartbeat interrupt routine, which should prevent this problem.

Note Sequencer v1.1

The Note Sequencer Tool Set provides numerous routines for controlling MIDI -compatible sequencing in the Apple IIGS. For detailed information on how to use the Note Sequencer, see the Note Sequencer ERS which accompanies these release notes.

Known bugs:

If the Note Sequencer was not started up and Note Sequencer calls are made, no error is returned.

If the MIDI Tools were not started up and a program calls StartSeq specifying that it wants to use the MIDI Tools, no error is returned, and the carry bit is not set. StartSeq will quit, but no error is posted to explain why.

The same problem occurs if StartSeq is called specifying that MIDI is to be used, but no output buffer has been allocated for it. Once again, StartSeq quits, but does not return an error or set the carry bit.

An update which fixes these bugs is available from Developer Tech Support.

Note Synthesizer v1.2

The Note Synthesizer is a set of routines which control the generation of musical notes and sounds on the Apple IIGS.For detailed information on how to use the Note Synthesizer, see the Note Synthesizer ERS which accompanies these release notes.

Print Manager v2.1

The call PrChooser has been completely redesigned. It now has a new user interface, and supports printing over AppleTalk zones.

Printer.Setup now saves separate settings for direct and network connections to printers, and saves the User Name for use on a network. Old version of the Printer.Setup file are incompatible with these changes, so the Print Manager will delete such files and create a new one in the correct format. Unfortunately, the old settings and default settings are lost, and the default settings are used to create the new setup file.

If the System disk is locked, the Print Manager will not be able to save changes in Chooser settings, and will display a dialog to warn the user of this fact. It will, however, save any changes to the Chooser settings in RAM, so the new settings will remain in effect until the current application quits.

If the Print Manager attempts to load a driver and finds that it is missing, it will pass control to a routine that determines what call was being made to the driver, pops the parameters off the stack, and returns a Missing Driver error. It will also display an alert asking the user to make sure a printer and port driver are selected.

PMStartup no longer loads any drivers into memory. It does not require that the Drivers folder be present, and if it is present, does not require that there be any drivers in it.

PMStartup checks to see whether the List Manager has been loaded. Since PrChooser uses List Manager calls, PMStartup will load the List Manager if it has not already been loaded.

New calls:

$3413 **PMUnloadDriver** returns:none parameters:driver:WORD

Unloads the current port driver, printer driver, or both, depending on the input parameter.The current driver is determined from the settings in the Printer.Setup file. Legal values for the driver parameter:

0 = unload both drivers

1 = unload printer driver

2= unload port driver.

The a register is zero if the call was successful. If an illegal input parameter is passed then the error code BadParam ($1308) is returned. Any other value is an error returned by the Loader call UserShutDown.

$3513 **PMLoadDriver** returns: none parameters: driver:WORD

Loads the current printer driver, port driver, or both, depending on the input parameter. The current driver is determined by the settings saved in the Printer.Setup file. Legal values for the driver parameter:

0 = load both drivers

1 = load printer driver

2= load port driver.

The a register is zero if the call was successful.If an illegal input parameter is passed then the error code BadParam ($1308) is returned. Any other value is an error returned by the Loader call InitialLoad.

Bug fixes:

The Chooser will now correctly display more than twelve driver files.

Alert messages now appear properly in 320 mode.

QuickDraw Auxiliary v2.3

New Call

$0C12 SpecialRect returns: none parameters: rectPtr : LONG

FrameColor: WORD

FillColor: WORD

SpecialRect frames and fills a rectangle in a single call, making separate calls to FrameRect and FillRect unnecessary. The single call to SpecialRect is considerably faster than separate calls to FrameRect and FillRect.

Bug fixes:

DrawPicture was formerly initializing the pen size incorrectly when the specified picture was scaled. This has been fixed.

When a picture that contained text was recorded, the recording feature did not work correctly, and the Font Manager was not used to install the font. This has been fixed.

When stretching and shrinking images, CopyPixels formerly initialized an internal error term incorrectly, causing the left side of the image to be adjusted incorrectly, and the right side to the image to contain garbage. This has been fixed.

Formerly, if the source bounds RECT in PPToPort's LocInfo parameter was smaller than the SourceRect parameter, then too many bytes would be stored in the picture, and redrawing it could cause a crash. This has been fixed.

The QD Aux call DrawPicture assumed that the Font Manager was available, but the Finder calls DrawPicture without using the Font Manager. To make this work, the font manager calls must all look for dispatcher errors, and keep the stack clean. SetPurgeStat failed to do this required maintenance. This has been fixed.

QuickDraw II v2.3

Bug fix:

InsetRgn failed to create a valid region when joining two points into one. This has been fixed.

Scrap Manager v1.3

The Scrap Manager has been updated, and will now run from ROM.

Sound Tools v2.3

The current version of the Sound Tools is required to support the MIDI Tools, Note Synthesizer, and Note Sequencer.

The Sound Tools now return the same version number and behave identically whether running with old or new versions of the Apple IIGS ROM.

Further information on the Sound Tools is provided in the Sound Tools ERS which accompanies these release notes. This ERS adds information which was not included in the IIGS Toolbox Reference.

A patch to FFStopSound, which was in its own file, has been incorporated in TS2,the patch file for ROM 2.

The Sound Tools' BootInit call to initialize the "MidiInitPoll" vector ($E11DD89) has been changed to an RTL.

Bug fix:

The Sound Tools set up the data bank register incorrectly, which caused it to write garbage into bank $00 memory. The offending code has been removed.

The call StopSound was not switching off the sound generators. This has been fixed.

Standard File v2.2

Standard File now uses the GetDirEntry call instead of reading directories itself.

Bug fixes:

A common exit routine was calling DisposeHandle with a random value, which can occasionally crash the system. This has been fixed.

Tool Locator v2.2

The Tool Locator uses a new algorithm to load tools from disk. It will only load tools from disk if a tool in ROM does not have a high enough version number. The Tool Locator makes no assumptions about what tools are in ROM and what are on the System disk.

For every tool that is to be loaded, the Locator makes a version call. If the version call returns an error either because the tool is not present, or because the version number is too low, then the tool is loaded form the System disk.

The Tool Locator no longer unloads all RAM-based tools every time TLShutDown is called. Instead, it returns the system to a default state, set by a new call in the Tool Locator, in which tools from ROM and from the System disk may be loaded.

New call:

$1601 SetDefaultTPT

Sets the default Tool Pointer Table to the current TPT. Used to permanently install a tool patch.An application should not make this call.

Window Manager v2.2

There are numerous significant changes to the Window Manager, notably in the definition of a Window Record. For a complete explanation of the Window Manager's new support for custom windows, see the **Designing Your Own Windows** documentation which accompanies these release notes.

There is one known bug in Window Manager v2.2:

CloseWindow will leave the top visible window unhilighted if an invisible window is in front of two or more visible windows in the window list. When CloseWindow closes the top visible window it leaves the next visible window unhilighted because it thinks that the invisible window is the next visible window. The problem apparently only occurs if you use invisible windows and SendBehind.

This bug will be fixed in the future. In the meantime,here is a work-around that will also work after the bug is fixed:

After CloseWindow, execute the following code:

```
pea     1               Pass TRUE to hilight
pha                     Space for result from FrontWindow
pha
_FrontWindow            Pass front visible window to HiliteWindow
_HiliteWindow           Make sure fornt window is hilighted.
```

Because HiliteWindow does nothing if a window is already hilighted, this will work after this bug has been fixed.

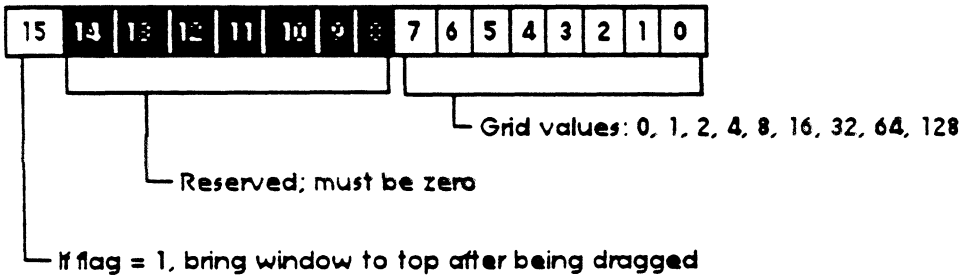
If you discover other situations in which the front visible window remains un-hilighted, this work-around should wotk with them as well.

TaskMaster now brings a window to the front **after** dragging is complete. TaskMaster previously brought windows to the front before dragging.

Standard windows will now draw their titles in sixteen colors regardless of mode.

The Grid parameter has been renamed to DragFlag. Its new definition is detailed in Fig.1.

Figure 1



It is no longer possible to specify Grid values of 256 or 512.

Alert Windows

The new call AlertWindow (see below) can be used to create Alerts for presenting the user with important messages. The call does all the work of creating and displaying the window and contents for the Alert, and returns the ID of the button that the user chooses.

AlertWindow accepts a pointer to a string which contains its message, and a pointer to an array of substitution strings. The substitution strings can be any of seven standard strings (such as "OK", "Continue", and so on) or can be specified by the application and stored in the buffer to which the substitution-string pointer refers.

Format of Alert String

Size Character

Character 1 is the size of the alert window. The character can be 0-9 where:

Character	Approximate max number of characters.
0	(Character followed by 4 integers that are size and position.)
1	30
2	60
3	110
4	175
5	110
6	150
7	200
8	250
9	300

Since AlertWindow provides a limited number of standard sizes, it is possible to create alerts which will display properly whether the Apple IIGS is in 320 or 640 mode. It is necessary, however, to design the text and buttons carefully in order to make this work.

The following table shows the dimensions of the standard alert windows. This is to give an idea of the size of each window. Application code should not rely on the exact widths, heights, or position of standard windows.

Character	Height 320	Width 320	Height 640	Width 640
1	46	152	46	200
2	62	176	54	228
3	62	252	62	300
4	90	252	72	352
5	54	252	46	400
6	62	300	54	452
7	80	300	62	500
8	108	300	72	552
9	134	300	80	600

Icon Number

Next character is the icon number. The icon number can be 0-9 where:

0 No icon.
1 custom icon, followed by:
LONG Pointer to image data.
WORD Number of bytes image data is wide.
WORD Number of scan lines image data is high.
2 Stop icon.
3 Note icon.
4 Caution icon.
5 Disk icon.
6 Disk swap icon.
7-9 are reserved - DON'T USE THEM.

Separator Character

The next character is a separator character. The separator can be any character you would like and cannot appear in the message text or button strings. The separator is used to separate the message from the first button string and each button string from each other. For purposes of standardization the / character might be a good choice.

Message Text

Following the separator character comes the message text. Any characters allowed by LETextBox2 are allowed in the message text. See "Special Characters" for additonal functions of message text. The total size of message text, after substitution of strings is limited to 1000 characters.

Button Strings

The first character after the message termination character is the beginning of the first button's title. The title can then be followed with either another message termination character and button title, or a string termination character (zero) to end the alert string. A total of three button titles may be included at the end of the alert string. These buttons will be evenly spaced and centered at the bottom of the alert window. The width of each button will be the same size and be set according to the widest button title. The total size of button text, after substitution of strings is limited to 80 characters.

Termination of Alert String

After the last button title is a zero (0) to end the alert string.

Special Characters

The following special characters can be embedded in the message text and button strings of an alert. In order to have a special character appear in the text of a button or message, enter it twice in the string. For example, if you want '^' to appear in an Alert message, you must enter it in the message string as '^^'.

^ If ^ is the first character in a button string the button will be considered the default button. The default button is the the button selected if the user presses the return key on the keyboard. This button will also appear bold on the screen. Only one button can be the default button. After the ^ character the button title must follow as in any other button. Other special characters may also appear after the '^'. A single ^ character in message text has no affect and is deleted from the message.

Substitute standard string. The # character must be followed by an ASCII decimal number. Numbers 0-6 can be used. 7-9 are reserved and should not be used. The standard substitution strings are:

#0	OK
#1	Cancel
#2	Yes
#3	No
#4	Try Again
#5	Quit
#6	Continue

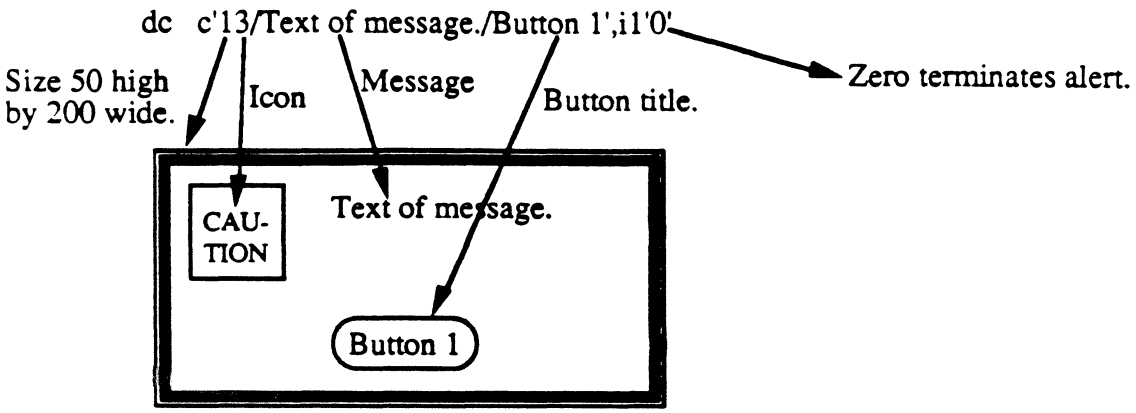
• Substitute given string. The • character followed by an ASCII decimal number from 0 through 9 denotes a substitution string to be inserted at that point. The • character and the following number will be replaced by the corresponding string in the specified substitution array. A pointer to the substitution array is passed to **AlertWindow**. The substitution array is defined as an array of LONG pointers where:

LONG[0]	Pointer to string that will substitute for •0.
LONG[1]	Pointer to string that will substitute for •1.
LONG[2]	Pointer to string that will substitute for •2.
LONG[3]	Pointer to string that will substitute for •3.
LONG[4]	Pointer to string that will substitute for •4.
LONG[5]	Pointer to string that will substitute for •5.
LONG[6]	Pointer to string that will substitute for •6.
LONG[7]	Pointer to string that will substitute for •7.
LONG[8]	Pointer to string that will substitute for •8.
LONG[9]	Pointer to string that will substitute for •9.

Substitution strings can be a C type, Pascal type, or RETURN terminated. C type and RETURN terminated strings are selected by passing 0 to **AlertWindow** as the string flag. Pascal strings are selected by passing 1.

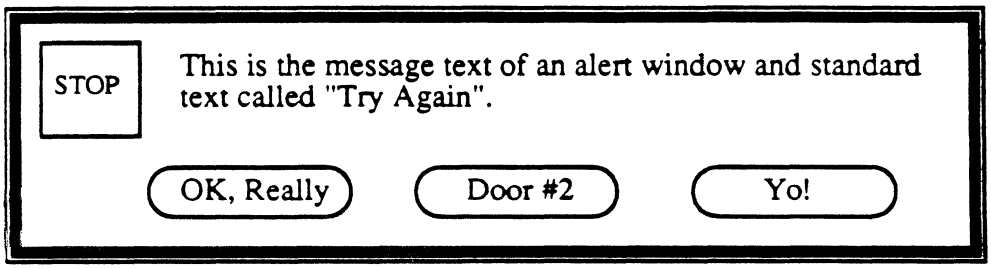
Elements do not need to be defined if they are not referenced in the alert.

Here are some examples of alert strings that can be passed to **AlertWindow** in APW 65816 assembler syntax.



A more complex alert string:

```
dc   c'51/This is the *0  of *3 alert *2*1 and standard text called "#4"./'
dc   c'^#0, Really/*4/Yo!',i1'13'
```



Where substitution array =

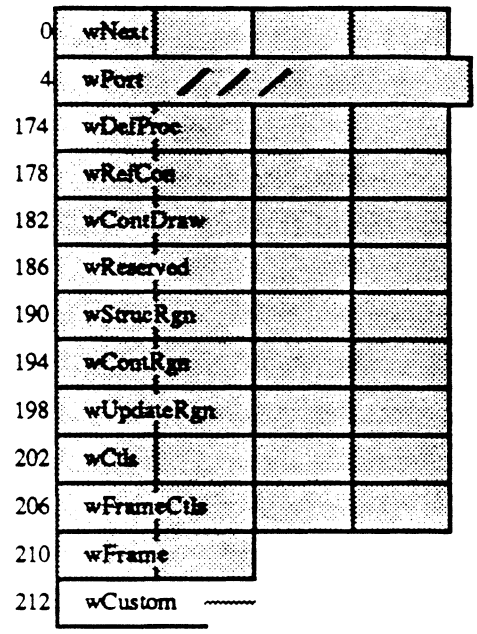
```
                dc   i4'sub0,sub1,sub2,sub3,sub4'
sub0    dc   c'message text',i1'0'
sub1    dc   c'dow',i1'0'
sub2    dc   c'win',i1'13'
sub3    dc   c'an',i1'0'
sub4    dc   c'Door #2',i1'0'
```

Window Records

The Window Record data structure has been redefined. The new definition is illustrated in Fig. 2.

Figure 2

Offset Field **Field description**



LONG - Pointer to next window record, zero is end of list.

BYTE[170] - Window's grafPort.

LONG - Address of window's definition procedure.

LONG - Reserved for application's use.

LONG - Address of routine that will draw window's content.

LONG - Reserved by Window Manager, do not use.

LONG - Handle of window's structure region.

LONG - Handle of window's content region.

LONG - Handle of window's update region.

LONG - Handle of first control in window's content.

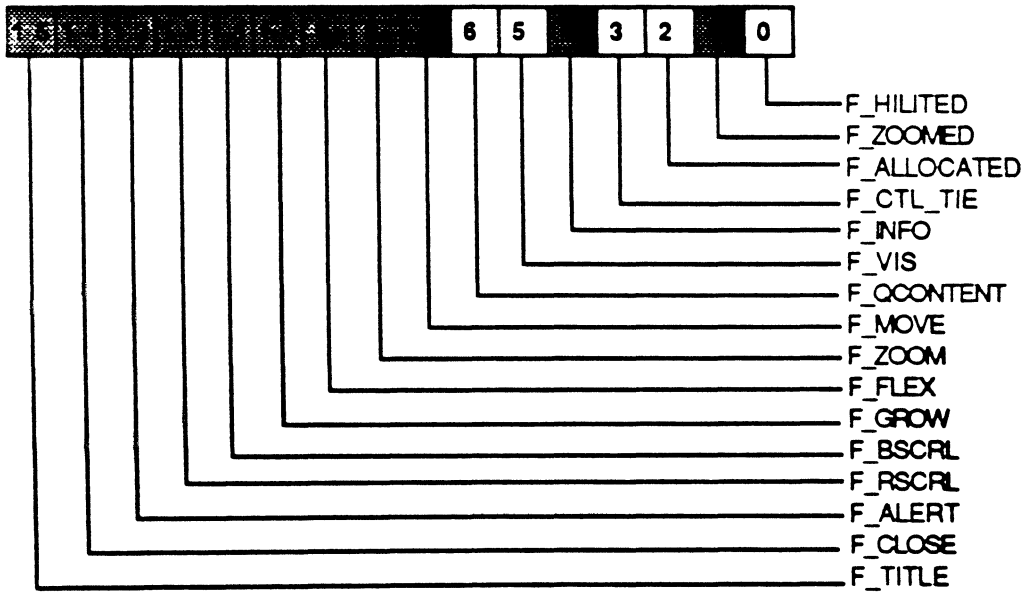
LONG - Handle of first control in window's frame.

WORD - Flags that define window.

BYTE[n] - Additional data space defined by window's defProc.

The wReserved field is a new data field reserved by Apple for future expansion.

The wFrame field is illustrated below. The shaded bits in the diagram are for use by window defProcs. The values named in the diagram are those used by the standard document window defProc. Unshaded bits are reserved bythe Window Manager and are the same for all windows.



Bug fixes:

GetFrameColor now works as documented.

SetDataSize was using incorrect minimum values. This has been fixed.

Some call the Window Manager made to application code did not work if the code was located at the first byte of a bank. For example, the window content-drawing routine would not work in this situtation. This has been fixed.

Calling BringToFront and SelectWindow with an invisible window no longer erases the contents of windows under the invisible one.

The width of vertical lines in alert window frames has been changed. The vertical lines are now 2 pixels wide instead of 3, so that their dithered color will be the same as the content.

The manual's description of SetZoomRect is incorrect. The correct description is:

> SetZoomRect sets the fZoomed bit of the window's wFrame record to zero. The RECT passed to SetZoomRect then becomes the window's zoom RECT. The window's size and position when SetZoomRect is called becomes the window's unzoomed size and position, regardless of what the unzoomed characteristics were before SetZoomRect was called.

The standard document window positioned the info bar one pixel too high when created without a title bar. This has been fixed.

Correction to the Toolbox Reference:

> Apple IIGS Toolbox Reference page 25-126, third line:
>
> > If wmTaskMask bit tmInfo (bit 15) = 1
>
> should read:
>
> > If wmTaskMask bit tmInfo (bit 15) = 0

GetFirstWindow returned a pointer to the first window's Window Record, and not to its grafPort as documented. This has been changed so that the call now behaves as documented.

The calls FromDesk, ToDesk, and SetDeskTop have been fixed so that they will now recompute the visRgn of every window.

When used with a window which does not have scroll bars, the call WindNewRes will call the window's defproc to recompute window regions. A call to SizeWindow is no longer necessary under these circumstances.

DragRect has been fixed so that when TaskMaster calls DragWindow (and thus, DragRect) in 640 mode, the x position will remain on the specified Grid.

The Window Manager had some problems displaying close-boxes and Titles, which have been fixed. Specifically, a long title sometimes conflicted with the close box when a window was resized to a narrow shape, and the close box was not correctly clipped by the menubar.

New Calls:

$580E **GetWindowMgrGlobals** returns: LONG parameters: none

Returns a pointer to the Window Manager global data area.

$5A0E **StartFrameDrawing** returns: none parameters: windowPtr: LONG

Sets up to draw a window frame. Should only be called by window definition procedures. Must be balanced by a call to EndFrameDrawing when drawing is completed.

$5B0E **EndFrameDrawing** returns: none parmaeters: none

Restores Window Manager variables after a call to StartFrameDrawing.

$5C0E **ResizeWindow** returns: none parameters:

hiddenFlag: WORD

rectPtr: LONG

grafPortPtr: LONG

Moves, re-sizes, and draws the window specified by grafPortPtr. rectPtr is a pointer to the window's content region. hiddenFlag is a Boolean parameter. A TRUE value specifies that portions of the window which are covered should not be drawn. If the value is FALSE, the entire content of the window is drawn.

$550E **DrawInfoBar** returns: none parameters:

grafPortPtr: LONG

Redraws the info bar of the window specified by grafPortPtr. The method used to redraw the info bar 's interior is the routine specified by the wInfoDefProc field of the paramList passed to NewWindow when the window is created. The Window Manager will automatically clip info bar drawing to the dimensions of the info bar , and to the visible region of the window.

$590E **AlertWindow** returns: Word parameters:

stringType: Word

subStrPtr: Long

alertStrPtr: Long

Creates an alert window which displays a message pointed to by alertStrPtr. The message can be either a C or Pascal string, as specified by stringType. A value of 0 signifies that the message is a C string, and a value of 1 that it is a Pascal string. subStrPtr points to an array of substitution strings for use with substitution characters. See the Window Manager section "Alert Windows" for a more detailed description of the use of Alert Windows.

Drivers

ImageWriter Drivers

ImageWriter LQ Driver

This is a new file in System Disk 3.2. It is not a special driver for the ImageWriter LQ, but simply allows the Apple IIGS to find the ImageWriter LQ on an Apple Talk network. It does not support the ImageWriter LQ's special features.

ImageWriter Driver v2.1

The ImageWriter driver has been completely re-written. It contains faster imaging routines, determines more quickly when no image is to be printed, and will not clear buffers which have already been cleared. Low level routines have been modified, restructured, and eliminated. Low level routines that were undocumented have been removed.

The new driver incorporates faster imaging routinesand faster determination of when no image is to be printed, and does not clear buffers which have already been cleared.

iVres now supports vertical condensed printing.

The procedure PrintCheck and its corresponding routines have been commented out. None of the code was being used, and it caused problems when printing to a spooler.PrintCheck sent a query on AppleTalk to determine the configuration to the ImageWriter it was printing to. The code then never made a PAP read to retrieve this information (whether the printer had a color ribbon, the printer's ID, and so forth). A spooler would then wait for the imagewriter to do a PAP read so that it could do a PAP write of the information form the initial query.

Bug fixes:

320 mode landscape printing was imaging too high on the page, This has been fixed.

Insertion point and editing now work correctly in the job dialog's FROM LineEdit box.

The driver did not update choices selected in the style dialog and the print dialog due to internal version number inconsistencies. This has been fixed.

The printer intermittently skipped print lines when the 'Best Text' quality was selected. This has been fixed.

The portRect of the grafPort is now the page RECT.

Landscape mode with gaps between pages did not work correctly with version 2.0 of the driver. This has been fixed.

The driver startup routine now initializes the serial port. This fixes a problem in which the first control code was lost when printing in draft mode.

The driver now has added filtering for the c cedilla character in LLDText. This allows it to be printed in draft mode.

The driver formerly crashed if you attemptyed to print when the printer was off-line or the cable was disconnected. This has been fixed.

LaserWriter

No prep file is necessary, making this driver compatible with all Macintosh LaserWriter drivers.

Insertion point and editing now work correctly in the job dialog's FROM LineEdit box.

Command-F (Open-Apple-F) redirects PostScript printing to a disk file. The resulting file contains the PostScript text for the redirected print job. The file will be named POSTSCRIPT.GS, and is created in the */SYSTEM/DRIVERS folder.

The current version of the LaserWriter file can print LaserWriter Plus fonts, and supports downloading bitmap fonts. The driver now supports printing with the following LaserWriter Plus fonts:

Font IDs (all values are decimal)

Zapf Dingbats	13
Bookman	14
Helvetica Narrow	15
Palatino	16
Zapf Chancery	18
Avant Garde	33
New Century Schoolbook	34

The LaserWriter driver can process the following PostScript picComments:

Name	Kind	Size	Handle
PostScriptBegin	190	0	NIL
PostScriptEnd	191	0	NIL
PostScriptHandle	192	-	PSData
PostScriptFile	193	-	FileName
TextIsPostScript	194	0	NIL

picComments allow a program to send PostScript commands directly to a LaserWriter. This ability gives applications access to the advanced capabilities of PostScript, and allows them to create needed effects without using more general QuickDraw II routines. A picComment must begin with PostScriptBegin and end with PostScriptEnd.

There are three different ways to send PostScript commands to a LaserWriter.

1) The application may pass a PostScriptHandle to the LaserWriter. This handle identifies a buffer containing PostScript commands in the form of ASCII data.

2) The application can pass a FileName, which points to the pathname of a file containing PostScript commands.

3) The application can use the TextIsPostScript command to send PostScript in the form of QuickDraw StdText calls.

The size fields of the PostScriptHandle and PostScriptFile commands must contain the size of the data to be read.

picComments do not take the place of normal Print Manager calls. From the point of view of the Print Manager, a picComment is simply data sent to the printer, just like any other data. An application should make all the Print Manager calls normally whether it uses picComments or not.

A few rules for the use of PostScript piComments:

The LaserWriter driver does not check for PostScript errors, so the data sent to the LaserWriter must be correct.

Always terminate PostScript text with a return character.

The transformation that the driver uses will flip text and print it upside down on the page. Applications should set up their own transformation matrices to serve their needs.

Never nest PostScript Begins and Ends.

Never use the LaserWriter's userdict. Instead, define a local dictionary for you application's use.

See chapter 3 of the LaserWriter Reference Manual for examples of how to use picComments.

The following PostScript commands can alter certain conditions that the driver sets up for the print job. Applications should therefore not use these calls:

exitserver
initgraphics
grestoreall
erasepage
showpage

Bug fixes:

The driver now consistently reports error conditions returned by the printer, such as 'out of paper', or 'font not found'.

Safeguards are now provided against pixel maps whose width is not on a byte boundary.

Printer Port Driver v1.1

Color selection escape code 5 caused problems with the ImageWriter I. This has been fixed.

Modem Port Driver v1.2

Some files could not be printed without initializing the printer. Code has been added to DevOpen to fix this problem.

Color selection escape code 5 caused problems with the ImageWriter I. This has been fixed.

AppleTalk Port Driver v2.0

The AppleTalk port driver now supports printing over zones.

DevPrChanged has been modified. It now just checks whether the AppleTalk Port has been selected in the Control Panel. If not, it displays an alert box with the message "AppleTalk is not selected in the Control Panel".

DevStartup formerly accepted one parameter, which had to be a correctly formed network name for a printer on the AppleTalk network. The call now accepts three parameters:

- pointer to a printer name
- device name
- zone name

The call concatenates these names, leaving out any unnecessary characters at the end of each one, to form the correct network name.

Strings, printer names, device names, and zone names may now be up to 31 characters long.

The auxiliary file type of the AppleTalk Port driver has been changed from 2 to 3.

AppleTalk IWEM v1.2.1

The IWEM file has been reduced in size by removing comments and white space.

There was a bug which occurred in graphic printing in the ImageWriter Emulator .1.1.2 on LaserWriters with ROM versions 3.0 or higher. This has been fixed.

Line wrapping has been added to allow for long strings of data.

The ESC-S command, which is the equivalent of an ESC-G command, did not generate graphics properly in IWEM v1.2. This has been fixed.

AppleMIDI (drivers) v1.0

This system disk includes MIDI device drivers for the MIDI Tool Set, the Apple MIDI Interface, and the Passport MIDI interface card (card6850.MIDI).

Utilities

Finder 1.1

Bug Fixes:

The Finder has been altered so that its handle remains fixed while another application is running. Formerly the Finder could be moved during memory compaction, causing crashes.

Program Launcher v2.2

Four new tools have been added to the list of versions retrieved.

029	Audio Compression and Expansion tools (A.C.E.)
030	Reserved for future use
031	Reserved for future use
032	MIDI Tools

Start

Start will not run the Finder unless it finds at least 375K of free memory.

System Utilities v3.1

Numerous bugs have been fixed.

AppleTalk Utilities

Chooser v1.2

A number of bugs have been fixed.

Namer v1.2

A number of bugs have been fixed.

Bug fixes:

The device type window and the names window no longer scroll simultaneously.

Documentation Développeurs
Apple Computer France 1987

Document développeur numéro 87

System Disk Release 2.0D3

type d'upgrade de ce ducument : 7
1 Documentation de première catégorie inchangée
2 Documentation de deuxième catégorie mise à jour
3 Documentation de deuxième catégorie inchangée
4 Mise à jour payante de la documentation de première catégorie
5 Mise à jour gratuite de la documentation de première catégorie
6 Nouveautés payantes non vitales
7 Nouveautés gratuites et vitales

Taille : 65 page(s) environ

Domaine : Dev : System

VERSION : 2.0D3
DATE : 4.03.87

System Disk Change History by Component

Version 2.0 D3
5 Mar 87

Special Note. All ProDOS 16 load files (except ATINIT) on this disk have been compacted. This includes all tools and drivers.

ProDOS 8
- 1 65C02 instruction removed
- 1 Disk II reset phase routines changed.

ProDOS 16
- 1 Bug in Format call fixed
- 1 Erase_Disk Call added
- 1 Get_Dir_Entry Added
- 1 New stack based entry point added
- 1 I/O handles are all zero on Open calls.
- 2 Added D_Info call
- 2 Bug in Get_Dir_Entry fixed.

Loader
- 1 Loader supports compacted files and latest OMF. Supports all calls in ERS.

Launcher
- 1 New version reporter.
- 2 Compacted.

Tool Setup

Fixed
- 1 Compacted.

ADB:
- 1 AbsOn, AbsOff and ReadAbs now work.

SANE:
- 1 Version Call works.

Desk Manager:
- 1 Shuts down classic desk accessories correctly.

Tool Locator:
- 1 ProDOS errors are now returned by LoadTools and LoadOneTool.
- 1 UnloadOneTool read the tool number to unload incorrectly off stack.

QuickDraw
- 1 PenMasks did not work in 640 mode when origin was not multiple of 8.

Not Yet Fixed or Verified

QuickDraw
- 1 Painting from the screen does not hide the cursor.
- 1 Lines with end points outside a port but with pen size large enough to intersect the port are over clipped?
- 1 TextBounds calls do not adjust to current style?
- 1 QuickDraw is not yet good at drawing across bank boundries.
- 1 New FontFlags Feature.

Desk Manager:
- 1 RestoreScreen and SaveScreen Crash

ATINIT

No Changes.

Window Manager

Fixed:

2 Compacted.
2 GrowWindow and SizeWindow were not correcly calculating the content region
2 GrowWindow displays the outline of the frame a little differently.
2 ValidRgn and InvalRgn change the given regio into global coordinates but for compatibility reasons this cannot change. Doc will change.
2 New TaskMaster feature added
2 WindowGlobal call added.
2 Feature added to Desktop call.

Not Yet Fixed or Verified.
None known.

Menu Manager

Fixed:

2 Compacted.
2 MenuNewRes works when switching from 320 to 640 mode.
2 InitPallete in 640 mode works more than once.
2 New MenuGlobal call added.
2 New menu help feature added.
3 MenuNewRes really works when switching from 320 to 640 mode.

Not Yet Fixed:
None known.

Control Manager

Fixed:

2 Compacted.
2 TrackControl returned the wrong part code for check boxes.
2 HiliteControl and ShowControls no longer change the current port.
2 Control manager preserves current port when drawing.

Not Yet Fixed:
None known.

Line Edit

Fixed:

1 New call TextBox2 works like TextBox but handles word wrap and justification.
2 Compacted.
2 Triple clicking an empty linedit item no longer loses the carret.
2 Changing the text contents when text is selected no longer loses the carret.
3 Inactive empty fields no longer have carrot droppings

Not Yet Fixed:
None known.

Dialog Manager

Fixed:

1 UpdateDialog does not crash.
1 The dialog items which use custom controls no longer crash randomly (internal def procs have been changed).
2 Compacted.

Not Yet Fixed or Verified
IsDialogEvent claims all update and activate events.
Tabbing order weird when multiple LE items in dialog

QuickDraw Auxilary

1 Pictures are more solid.
1 CopyPixels is more solid.
1 DrawIcon has been added.
2 Compacted.
2 DrawPicture works again.

PrintManager

1 New This Time.
2 Compacted.

Scrap Manager

2 Compacted.
No Changes in code.

Standard File

2 Compacted.
No Changes

Note Synthesizer

2 Compacted.
No Changes

Font Manager

1 Bug setting size field in port is fixed.
1 Font scaling works.
2 Compacted.
3 Call FontNum2ItemID was added
3 Choose font does not display the sizes of created fonts.

List Manager

1 Bug in port setting on first drawing is fixed.
1 Amount of scrolling for clicking in the page region is now calculated correctly.
2 Compacted.

ImageWriter Driver

1 New.
2 Compacted.

Laserwriter Driver
1 New.
2 Compacted.
2 Can print text in fonts other than Courrier even when Mac is not on the network.

Printer Driver
1 New.
2 Compacted.

Modem Driver
1 New.
2 Compacted.

AppleTalk Driver
1 New.
2 Compacted.

DeskTop II (MouseDesk)
No Changes.

AppleTalk Chooser
No Changes.

AppleTalk Namer
No Changes.

AppleTalk IWEM
Fixed
None.

Not Yet Fixed
Cannot handle ascii code zero.

System Utilities
No Changes.

BASIC.SYSTEM
No Changes.

BASIC Launcher
No Changes.

Fonts
No Changes, one addition: Shaston16

Apple Desktop Bus Tool

Changes Since Last Time

The three calls that did not work now work:

ABSOn
ABSOff
ReadABS

Latest Documentation

The latest version of the ERS is dated May 15, 1986. The pre-Beta version of the reference manual is also useful.

Control Manager

Changes from last release:

1.) The control manager no longer changes the contents of a window's grafport when drawing controls.

2.) TrackControl returned a part code for check box when the control was actually a radio button. The Control Manager has been changed to return the proper part code for radio buttons.

Documentation Issues:

1.) Creating a grow control was overlooked in the documentation. So, here's how it's done:

Call **NewControl** with the following parameters:

TheWindowPtr	Pointer to the window owner.
BoundsRect	Enclosing RECT.
TitlePtr	Not used, any value is OK.
Flag	Bit 7 clear for visible, set for invisible, bits 0-6 all clear.
Value	Not used, any value is OK.
Param1	Not used, any value is OK.
Param2	Not used, any value is OK.
DefProc	$08000000.
RefCon	Any value you wish.
ColorTable	Zero for default, or pointer to a custom color table.

Color Table:

GrowOutline	WORD Bits 4-7 = outline color, other bits must be zero.
GrowNorBack	WORD Color of interior: Bits 0-3 = icon's foreground color. Bits 4-7 = background color. Bits 8-15 = zero.

The grow control should never be highlighted. **TrackControl** should not be called for a grow control. **GrowWindow** should be called to track the grow control when FindControl returns a hit in a grow control. You should use your own tracking routine if you are using the grow control for a purpose other than resizing windows.

2.) The color tables for controls aren't defined very well. Here's another try:

The simple button color table is defined as:

SimpOutline	= outline color: Bits 0-3 = zero. Bits 4-7 = outline color (bold outline and drop shadow if used). Bits 8-15 = zero.

SimpNorBack = interior color when not highlighted:
Bits 0-3 = zero.
Bits 4-7 = background color.
Bits 8-15 = zero.
SimpSelBack = interior color when highlighted:
Bits 0-3 = zero.
Bits 4-7 = background color.
Bits 8-15 = zero.
SimpNorText = text color when not highlighted:
Bits 0-3 = foreground color (only bits 0-1 used in 640 mode).
Bits 4-7 = background color (only bits 4-5 used in 640 mode).
Bits 8-15 = zero.
SimpSelText = text color when highlighted:
Bits 0-3 = foreground color (only bits 0-1 used in 640 mode).
Bits 4-7 = background color (only bits 4-5 used in 640 mode).
Bits 8-15 = zero.

The check box color table is defined as:

CheckReserved = reserved, must be zero.
CheckNorColor = color of check box when not highlighted:
Bits 0-3 = foreground color (only bits 0-1 used in 640 mode).
Bits 4-7 = background color (only bits 4-5 used in 640 mode).
Bits 8-15 = zero.
CheckSelColor = color of check box when highlighted:
Bits 0-3 = iforeground color (only bits 0-1 used in 640 mode).
Bits 4-7 = background color (only bits 4-5 used in 640 mode).
Bits 8-15 = zero.
CheckTitleColor = color of title's text:
Bits 0-3 = foreground color (only bits 0-1 used in 640 mode).
Bits 4-7 = background color (only bits 4-5 used in 640 mode).
Bits 8-15 = zero.

The radio button color table is defined as:

RadioReserved = reserved, must be zero.
RadioNorColor = color of radio button when not highlighted:
Bits 0-3 = foreground color (only bits 0-1 used in 640 mode).
Bits 4-7 = background color (only bits 4-5 used in 640 mode).
Bits 8-15 = zero.
RadioSelColor = color of radio button when highlighted:
Bits 0-3 = foreground color (only bits 0-1 used in 640 mode).
Bits 4-7 = background color (only bits 4-5 used in 640 mode).
Bits 8-15 = zero.
RadioTitleColor = color of title's text:
Bits 0-3 = foreground color (only bits 0-1 used in 640 mode).
Bits 4-7 = background color (only bits 4-5 used in 640 mode).
Bits 8-15 = zero.

The scroll bar color table is defined as:

ScrollOutline = outline color:
- Bits 0-3 = zero.
- Bits 4-7 = outline color for arrow boxes, thumb and page region.
- Bits 8-15 = zero.

ArrowNorColor = color of arrows when not highlighted:
- Bits 0-3 = foreground color (only bits 0-1 used in 640 mode).
- Bits 4-7 = background color (only bits 4-5 used in 640 mode).
- Bits 8-15 = zero.

ArrowSelColor = color of arrows when highlighted:
- Bits 0-3 = foreground color (only bits 0-1 used in 640 mode).
- Bits 4-7 = background color (only bits 4-5 used in 640 mode).
- Bits 8-15 = zero.

ArrowBackColor = color of arrow box interior background:
- Bits 0-3 = zero.
- Bits 4-7 = background color .
- Bits 8-15 = zero.

ThumbNorColor = thumb's interior color when not highlighted:
- Bits 0-3 = zero.
- Bits 4-7 = background color.
- Bits 8-15 = zero.

ScrollReserved = reserved, can be any value.

This entry was documented as the color of the thumb when selected but was never used by the Control Manager. It is not going to be installed because it would mean redrawing the thumb when selected.

PageRgnColor = page region's interior color:
- Bits 0-3 = second dither color.
- Bits 4-7 = first dither color and color if solid.
- Bit 8 = 1 for dither pattern, 0 for solid color.
- Bits 9-15 = zero.

InactiveColor = color of scroll bar's interior when inactive.
- Bits 0-3 = zero.
- Bits 4-7 = color of interior's background.
- Bits 8-15 = zero.

Desk Manager

Changes Since Last Time

CDA shut down calls are now made to all installed CDAs.

Latest Documentation

The latest ERS is dated August 24, 1986. The pre-Beta reference manual is also useable.

Dialog Manager

Changes Since Last Time

The **UpdateDialog** now cleans up the stack correctly.

A number of the internal def procs were modified to work correctly when called by the control manager.

SetDAFont has been fixed.

Documentation Issues:

Not Yet in Documentation: ModalDialog2. This call works just like **ModalDialog** but returns both the ID of the item hit and the PartCode of the item hit.

ModalDialog2 call #$2C15

Stack before call

Previous Contents	
Result Space	LONG
FilterProc Ptr	LONG

Stack after call

Previous Contents	
ItemHitInfo	LONG

Where the low word of ItemHitInfo is the item hit and the high word of ItemHitInfo is the partcode of that item. (The possible partcodes are listed below.)

When a key is pressed and there is an EditLine item in the dialog, the PartCode is inEditLine, and the itemID is the ID of the current active EditLine.

Two Call Numbers Changed. Two calls used reserved call numbers that they should not have used.

ErrorSound was call #$07, it is now call #$09
SetDAFont was call #$08, it is now call #$1C

Calls 7 and 8 are reserved in all tool sets for future use. These call numbers were assigned by accident and have been changed. Calls 7 and 8 are still entry points to these routines but may not be supported in the future. Please switch to the new call numbers as soon as possible.

New Item Type: UserCtlItem2. For a UserCtlItem2 (value = 21) the ItemDescr parameter (passed to NewDItem at create time) is a pointer to a parameter block of additional data.

ItemDescr = pointer to following structure:

DefProcParm :	LONG	Address of definition procedure (same as defProc for NewControl)
TitleParm :	LONG	Pointer to title string (same as title for NewControl)
ParamLong :	LONG	LoWord = Param2, HiWord = Param1 (same as Param1 and Param2 for NewControl)

This makes it possible to use a Scroll Bar or any kind of control as a UserCtlItem2. For a scroll bar, just pass the scroll bar defproc ($06000000) in the DefProcParm, the view size in Param1 and the total size in Param2. Then you can handle the different part codes using **ModalDialog2.**

Non-item Controls. If you want to put a control in a dialog, but you don't want it to be handled by the Dialog Manager, just give it a Refcon = 0. You can temporarily prevent the Dialog Manager from handling an item by getting its Refcon, saving it and clearing it. Then when you change your mind, you can just restore it. The calls to use for this are **GetControlItem, GetCtlRefCon** and **SetCtlRefCon.**

Latest Documentation

The latest ERS is dated August 28, 1986. The pre-Beta reference manual is also useable.

Important Information

Possible PartCodes

0		No part.
1		Reserved for internal use.
2	inButton	Simple button.
3	inCheckBox	Check box.
4	inRadioButton	Radio button.
5	inUpArrow	Up arrow.
6	inDownArrow	Down arrow.
7	inPageUp	Page up.
8	inPageDown	Page down.
9	inStatText	Static Text.
10	inGrow	Grow box icon.
11	inEditLine	Editable line.
12	inUserItem	User item.
13	inLongStatText	Long static text.
14	inIconItem	Icon.
15-31		Reserved for internal use.
32-127		Reserved for application's use.
128		Reserved for internal use.
129	inThumb	Thumb.
130-159		Reserved for internal use.
160-253		Reserved for application's use.

254-255 Reserved for internal use.

Dialog Equates.

```
;
; Dialog Manager Errors
;
DLGERR          equ  $1500      ; Error Base
BadItemType     equ  10         ; error number is DLGERR+ErrorNum
NewItemFailed   equ  11
ItemNotFound    equ  12
NotModalDialog  equ  13

; Item Types
;
ButtonItem      equ  10
CheckItem       equ  11
RadioItem       equ  12
ScrollBarItem   equ  13
UserCtlItem     equ  14
StatText        equ  15
LongStatText    equ  16
EditLine        equ  17
IconItem        equ  18
PicItem         equ  19         ; not implemented yet.
UserItem        equ  20
UserCtlItem2    equ  21

MinItemType     equ  ButtonItem
MaxItemType     equ  UserCtlItem2

;
; Part Code numbers for non-standard control item
;
inStatText      equ 9
inEditLine      equ 11
in UserItem     equ 12
inLongStatText  equ 13
inIconItem      equ 14

;
; To disable an item for the dialog manager, add ItemDisable to
; the ItemType
;
ItemDisable equ $8000

;
; Icon structure
;
;  - An Icon Handle is a handle to an IconRecord
;  - The definition of an Icon Record follows.
;
iconRect        equ 0           ; width must be multiple of 8
iconImage       equ iconRect+8  ; pixel image for the icon
```

```
;
; Command Number for the Dialog Scroll Bar Action Procedure
;
getinitview     equ 1
getinittotal    equ 2
getinitvalue    equ 3
scrolllineup    equ 4
scrolllinedown  equ 5
scrollpageup    equ 6
scrollpagedown  equ 7
scrollthumb     equ 8

;
; Item Template
;
itItemID        equ 0
itItemRect      equ itItemID+2
itItemType      equ itItemRect+8
itItemDescr     equ itItemType+2
itItemValue     equ itItemDescr+4
itItemFlag      equ itItemValue+2
itItemColor     equ itItemFlag+2

;
; Modal Dialog Template
;
dtBoundsRect    equ 0
dtVisible       equ dtBoundsRect+8
dtRefCon        equ dtVisible+2
dtItemList      equ dtRefCont+4

;
; dtItemList is a list of pointers to item templates terminated by
; a NIL pointer.
;

;
; Alert Template
;
atBoundsRect    equ 0
atAlertID       equ atBoundsRect+8
atStage1        equ atAlertID+2
atStage2        equ atStage1+1
atStage3        equ atStage2+1
atStage4        equ atStage3+1
atItemList      equ atStage4+1

;
; Additional Parameter Block for UserCtlItem2
;
DefProcParm     equ 0           ; custom control defproc
TitleParm       equ 4           ; pointer to control title
ParamLong       equ 8           ; equivalent to Param1, Param2
```

Event Manager

Changes Since Last Time

None.

Latest Documentation

The latest version of the ERS is dated June 2, 1986. The pre-Beta reference manual is also usable.

Font Manager

Changes Since Last Time

Font Scaling is implemented. If a program asks for a font in a size that is not available, a font is created in the correct size.

Latest Documentation

The latest version of the ERS is dated November 21, 1986. The pre-Beta reference manual is also usable except for the FMStartup call and the ChooseFont call.

Integer Math

Changes Since Last Time

None.

Latest Documentation

The latest version of the ERS is dated August 7, 1986. The pre-Beta version of the reference manual is also useful.

Line Edit

Changes Since Last Time

A new call has been added: LETextBox2. LETextBox2 takes the same inputs as LETextBox but handles word wrap and justification. Future changes will allow it to support imbedded style, font, size and color changes. A special escape code will be used to signal non-text information. We are currently planning to use ASCII code $01 for this. If you have any objections, let us know quickly.

Triple Clicking on an empty line edit time does not freeze the carrot.

Setting the text in a line edit record with SelStart <> to SelEnd does not freeze the carrot.

Not Yet In Documentation

LETextBox2 Call Number $2014

Draws the text in the rectangle specified by BoxPtr with the specified justification. LETextBox2 is not limited to one line of text; it supports word wrap, and wraps on specified carriage returns. The rectangle pointed to by BoxPtr is specified in local coordinates of the current GrafPort

Stack Before Call

Previous Contents	
TextPtr	LONG - POINTER to the text
Length	WORD - integer, length of text including carriage returns
BoxPtr	LONG - POINTER to a rectangle
Just	WORD - INTEGER
	<- SP

Stack After Call

Previous Contents	
	<- SP

The *Just* can be one of four values:

0 = left justified
1 = center justified
-1 = right justified
2 = fill justified

LETextBox2 is compatible with LETextBox. It erases the rectangle in BoxPtr before drawing in it. LETextBox2 does not create a line edit record and the text is not editable. No copies are made, nor space allocated for the text string.

Text should not contain ascii code 1. Any text with ascii code 1 in it will be impatable with future versions of LETextBox2.

Latest Documentation

The latest version of the ERS is dated August 12, 1986. The pre-Beta version of the reference manual is also useful.

List Manager

Changes Since Last Time

Bug in routine that first draws the list has been fixed. (The list was not always drawn correctly the first time.)

How far to scroll up and down when user clicks in page region is no longer fixed at 13. It is calculated from data in list record.

Latest Documentation

The latest version of the ERS is dated November 11, 1986. The pre-Beta reference manual does not contain any information on this tool.

Memory Manager

Changes Since Last Time

None.

Latest Documentation

The latest version of the ERS is dated July 29, 1986. The information in the Pre-Beta reference manual is also accurate.

Menu Release Notes

Changes from last release:

1. MNewRes now works if you start in 320 mode and switch to 640 mode.
2. InitPallete in 640 mode works more than once.
3. CountMItems now works.
4. New call MenuGlobal has been added.

MenuGlobal Call# $230F

input:	chgFlag:WORD	Negative to clear bits, positive to set bit, zero for nothing.
output:	newFlag:WORD	Current global menu flag after changed.

MenuGlobal is used to set and clear bits that effect how the Menu Manager performs tasks. If chgFlag has bit 15 set, it is ANDed with the global flag. If chgFlag has bit 15 clear, it is ORed with the global flag. Therefore if chgFlag is zero, the global flag will not change. MenuGlobal will always return the state of the global flag after changing the desired bit.

Only bit 0 of the menu global flag is defined. Bit 0 is defined as a help bit and is defined in the "Menu Help" section. The only values passed to MenuGlobal are:

$0000	Flag does not change, used to get the current state of the flag.
$0001	Turn menu help on.
$FFFE	Turn menu help off.

The values returned by MenuGlobal are:

$0000	Menu help off.
$0001	Menu help on.

5. A new feature has been added to the Menu Manager to enable applications to let users choose inactive (dimmed) menu items. The application can use the information to tell the user how the item can be made active. The following section describes how to initiate the help feature.

Menu Help

Generally, there is no way for a user to choose a dimmed menu item. Actually, the purpose of dimmed items is to show it cannot be chosen. However, the user does not have a way of knowing what it take to undim a dimmed item. This can be frustrating for the user. However there is a way of letting the user choose a dimmed item for the purpose of getting more information about the item.

The first step in initiating help is to pass $0001 to MenuGlobal. This tells the Menu Manager that your application knows about the help feature and will display a help message when the user does choose a dimmed item.

Once bit 0 of the menu global flag is set the Menu Manager will do the following things:

- dimmed items will appear dimly highlighted when the user moves the cursor over them
- MenuSelect will return the ID number of a selected dimmed item in the high word of the TaskData field of the Task record passed to MenuSelect.

Using TaskMaster require the setting of some additional parameters. Bit 14 of the the TaskMask field of the Task record passed to TaskMaster must be set. TaskMaster will then return the flag wInactMenu ($001C) when the user chooses a dimmed menu item. The dimmed item's ID number is returned in the high word of the Task record's TaskData field.

Clarification in documentation

There is a term used in the menu definition procedure, under custom menus, that could be confusing. The term item number and item ID number refer to two different things. Item number is the slot in the menu that an item occupies. One would be the first item in the menu and zero means not in the menu. Item ID number is a value given by the application to uniquely identify an item from all the other items in the menu bar. There could be a conflict if the ID number is returned by the mChoose function in a custom menu definition procedure. ID numbers can be greater than $7FFF and mChoose must set bit 15.

Misc Tools

Changes Since Last Time

None.

Latest Documentation

The latest version of the ERS is dated October 29, 1986. The pre-Beta version of the reference manual is also useful.

Note Synthesizer

Changes Since Last Time

None.

Latest Documentation

The latest version of the ERS is dated September 25, 1986. The pre-Beta version of the reference manual does not include a chapter on this tool.

ProDOS

Changes Since Last Time

Three new system calls have been added: Get_Dir_Entry ($1C), Erase_Disk ($25) and D_Info ($2C).

There is also a new stack based method for making system calls.

ProDOS 16 no longer provides access to the I/O buffer when opening files.

Latest Documentation

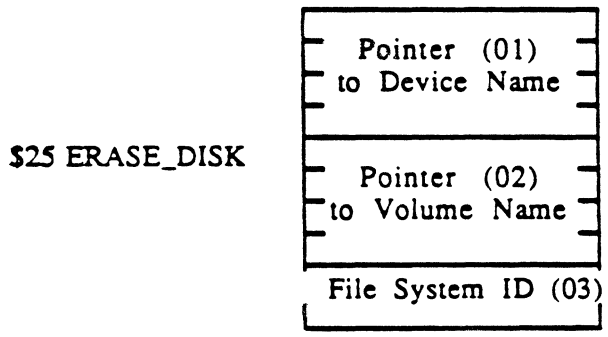
The reference manual describes everything but these new features. The new features are described below.

Not Yet In Documentation

When making the stack based system call, instead of the usual JSL $E100A8 followed by the call number and parameter pointer, push the pointer to the parameter block, push the call number and JSL $E100B0. After the call returns, control returns to the instruction after the JSL. The pointer to the parameter block and the call number will be removed from the stack.

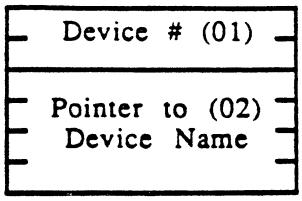
Previous versions of ProDOS/16 returned a handle to the I/O buffer when opening files, this version of ProDOS/16 will now return a handle value of $0000. Do not try to dereference a handle of $0000.

The Erase_Disk call works logically like the Format ($24) call except that Erase_Disk does not format the disk. It only writes out the boot blocks, the empty directory and the bit map. The remaining blocks on the disk are left alone. The input and output parameters are the same as the Format call.



The D_Info call has two parameters, a 2-byte device number that is inputted and a 4-byte pointer to a buffer that is also inputted. ProDOS/16 returns the device name into that buffer.

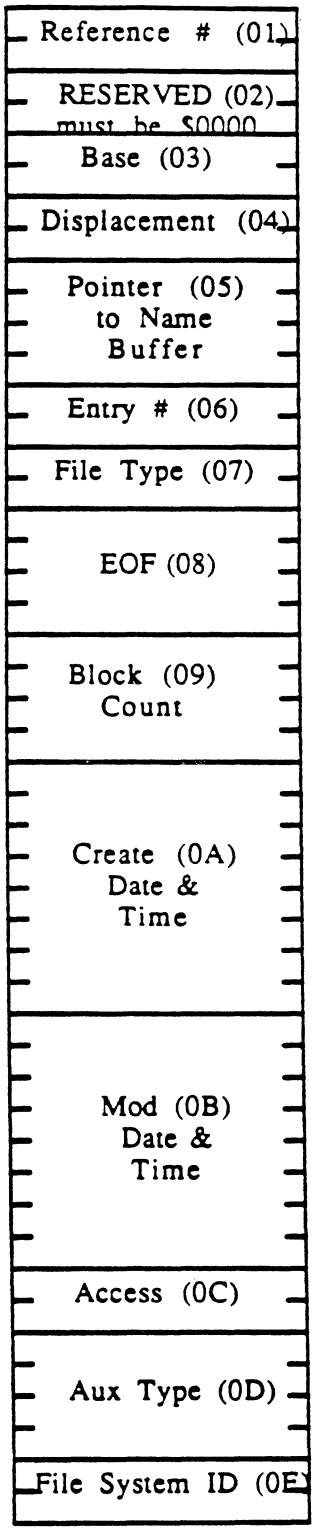
$2C D_INFO



The Get_Dir _Entry will return the pertinent information of active files from the file's directory. The old days of reading from the directory and interpreting the results are over! In addition to absolute retrieval, Get_Dir_Entry also allows relative return of file information from the last mark within the directory. For example, to simply catalog a directory, the caller supplies a valid file reference number, a value of $0001 in the base field and a value of $0001 in the displacement field. This will return the first active file entry going in the forward direction. Put this into a loop while incrementing the displacement field and presto! you can catalog a directory.

A value of $0000 in the base field signifies absolute entry. A value of $0002 in the base field signifies backwards direction, in case you ever wanted previous file entries. The displacement value tells the system which file entry to return based on the base field. If for example, the base value is $0001 and the displacement value is $0002, then every time you make the Get_Dir_Entry call, the system will return every other active file entry going in the forward direction.

$1C GET_DIR_ENTRY
(Directory Call)



QuickDraw II

Changes Since Last Time

Pen Masks work when the origin is not a multiple of 8.

Latest Documentation

The latest ERS is dated August 11, 1986. The Pre-Beta version of the reference manual accurately describes the calls as well.

Important Information

QuickDraw requires three (3) consecutive pages of bank zero memory for direct page (not two as described in early verions of the ERS and parts of the reference manual).

Not Yet In Documentation

InflateTextBuffer Call Number **$D704**

This call uses the inputs to see if the text buffer is at least as big as indicated, making it larger only if necessary

Stack Before Call

```
| Previous Contents |
|    NewWidth       |  INTEGER
|    NewHeight      |  INTEGER
|                   |<- SP
```

Stack After Call

```
| Previous Contents |
|                   |<- SP
```

This call was added specifically for the Font Manager, but may be useful for programs which deal with fonts without the Font Manager's help. It allows you to make sure the internal text buffer is large enough resizing only when necessary.

You pass the newheight and MaxWidth of a font you want to use. QuickDraw checks the current text buffer to see if it is large enough to accomodate these sizes and if it is not large enough, it is resized.

GetRomFont Call Number **$D804**

This call fills a record describing information about the font in ROM.

Stack Before Call

```
| Previous Contents |
|   Ptr To Record   |  POINTER
|                   |<- SP
```

Stack After Call

```
| Previous Contents |
|                   |<- SP
```

The record has the following form:

Family Number	integer
Style	word
Size	integer
HandleToFont	handle
NamePtr	pointer
fbrExtent	integer

for a total of 16 bytes.

GetFontLore Call Number **$D904**

This call is very similar to the GetFontGlobals call except that you can specify the maximum number of bytes it will return.

Stack Before Call

```
| Previous Contents |
| Space For Result  |  WORD
|   Ptr to Record   |  LONG
|    Record Size    |  LONG
|                   |<- SP
```

Stack After Call

```
| Previous Contents |
| Num bytes Xfered  |  integer
|                   |<- SP
```

Today's record has the following format:

FontID	: integer
Style	: word
Size	: integer
Version	: word
WidMax	: integer
FBRExtent	: integer

The GetFontGlobals call returns information about the current font. If we want to change the font format to include more information in the future, we will have to return more information in the GetFontGlobals call. To make this possible, we have a call which returns the size of the record filled in by GetFontGlobals. Unfortunately, this makes it very hard on programs wanting to set aside a fixed amount of space.

The GetFontLore makes it possible to set aside a fixed amount of space. It allows the program to specify the maximum number of bytes it can handle. GetFontLore will never return more.

A program can specify a number equal to the number of bytes being returned in the version of QuickDraw running on the day the program is written. Future versions of QuickDraw will not cause this program any problems.

QuickDraw Auxiliary

Changes Since Last Time

A number of bugs have been removed from the picture code

A new DrawIcon call has been added.

Latest Documentation

The latest version of the ERS is dated February 11, 1987. There is no information in the pre-Beta version of the reference manual about this tool set.

SANE

Changes Since Last Time

The SANEVersion call returns a reasonable result.

Latest Documentation

The latest version of the ERS is dated June 24, 1986. The pre-Beta version of the reference manual is also useable, but the most useful documentation for SANE is the manual for the Standard Apple Numeric Environment. It describes how SANE works on all Apple Machines.

Scheduler

Changes Since Last Time

None.

Latest Documentation

The latest version of the ERS is dated April 15, 1986. The pre-Beta version of the reference manual is also useable.

Scrap Manager

Changes Since Last Time

None.

Latest Documentation

The latest version of the ERS is dated June 26, 1986. The pre-Beta version of the reference manual is also useful.

Sound Manager

Changes Since Last Time

None.

Latest Documentation

The latest version of the ERS is dated June 26, 1986. The pre-Beta version of the reference manual is also useful.

Standard File

Changes Since Last Time

None.

Latest Documentation

The latest version of the ERS is dated September 22, 1986. The pre-Beta version of the reference manual is also useful but not complete. It does not yet include information about the SFAllCaps calls and does not include the templates for the standard calls.

Text Tools

Changes Since Last Time

None.

Latest Documentation

The latest version of the ERS is dated June 26, 1986. The pre-Beta version of the reference manual is also useful.

Tool Locator

Changes Since Last Time

A bug in UnloadOneTool was fixed.

Latest Documentation

The latest version of the ERS is dated February 19, 1986 and is out of date in that it does not include all the calls. The pre-Beta version of the reference manual is much more useful.

Not Yet In Documentation

Tools Not in the ERS. RAM-based tools are only loaded on request of the application. There are five Tool Locator calls to handle this: LoadTools, LoadOneTool, UnloadOneTool, TLMountVolume and TLTextMountVolume.

The Load Tools call allows the application to load a number of tools in a single call. Tools loaded in this way stay in memory until the applicaiton makes the TLShutdown call (or are unloaded individually). The LoadOneTool and UnloadOneTool calls lets the application load tools one at a time. These are used for tools you do not want to have in memory all the time. You can load the tool just before you use it and unload it just after you are done.

Th mount volume calls are used to inform the user that volumes needed to load tools or other important software are not available.

Where Tools Are Kept. Tools are kept in the Tools subdirectory of the system directory of the boot volume. They are load files of type $BA and each tool starts with its function pointer table. Tools are named after their tool number: Tool 23 is in a file named TOOL023, Tool 24 is in a file named TOOL024, etc.

Finding The System Disk. Since tools are kept on the boot disk, the user must have the boot disk around to load the tools. A problem arises when the boot disk is not on line. The user would be prompted to insert the disk. But how? There are many display modes on the Apple II. Which one is this application using now? Has it been intialized? What should the message say? Because of all these issues the tool locator will not automatically prompt the user to insert the boot disk when a load tool call is issued. It is the application's responsiblitiy to recover from a volume not found error after a load tools call is issued.

A Little Bit of Help. For desktop and text based applications the tool locator will provide a little help. One mount volume call uses the super hi-res display and prompts the user with program supplied messages. This call displays what looks like a dialog box but is not, since the dialog manager is a RAM based tool and may not be loaded. The other mount volume call uses the 40 column text display to prompt the user with program supplied messages.

Summary of Tool Set Numbers Now Assigned

1	Tool Locator
2	Memory Manager
3	Miscellaneous Tools
4	QuickDraw II
5	Desk Manager
6	Event Manager
7	Scheduler
8	Sound
9	Apple Desktop Bus
10	SANE
11	Integer Math Tools
12	Text Tools
13	Used Internally
14	Window Manager
15	Menu Manager
16	Control Manager
17	Loader
18	QuickDraw Auxilary Routines
19	Printing Manager
20	Line Edit
21	Dialog Manager
22	Scrap Manager
23	Standard File
24	Disk Utilities
25	Note Synthesizer
26	Note Sequencer
27	Font Manger
28	List Manager

The New Calls

LoadTools Call Number $0E01

Loads the requested tools from the boot disk.

```
Stack Before Call
    |  Previous Contents   |
    |   Ptr to ToolTable   |  LONG
    |                      |<- SP

Stack After Call
    |  Previous Contents   |
    |                      |<- SP
```

where ToolTable is

```
dc i'NumToolsRequired'
dc i'ToolNum1, MinVersion'
dc i'ToolNum2, MinVersion'
...
dc i'ToolNumN, MinVersion'
```

Tools are loaded from the TOOLS subdirectory of the SYSTEM directory of the boot volume. An application must give the load tools call before it makes a call to any RAM based tools.

RAM fixes and enhancements to the tools in ROM are not loaded this way. They are laoded automatically at boot time. However, you can use the LoadTools call to check the version of a ROM based tool.

Possible Errors

$0110	Version Error
$XXXX	Any Loader/ProDOS error is returned unchanged.

LoadOneTool Call Number $0F01

Loads the requested tool from the boot disk.

```
Stack Before Call
    |  Previous Contents   |
    |    Tool Number       |  integer
    |  Minimum Version     |  word
    |                      |<- SP

Stack After Call
    |  Previous Contents   |
    |                      |<- SP
```

Possible Errors

$0110	Version Error
$XXXX	Any Loader/ProDOS error is returned unchanged.

UnloadOneTool Call Number $1001

Unloads the requested tool from memory.

```
Stack Before Call
 |  Previous Contents  |
 |    Tool Number      |  integer
 |                     |<- SP

Stack After Call
 |  Previous Contents  |
 |                     |<- SP
```

Possible Errors

None.

TLMountVolume Call Number $1101

```
Stack Before Call
 |  Previous Contents  |
 |  Space For Result   |  word
 |      WhereX         |  integer
 |      WhereY         |  integer
 |      Line1Ptr       |  pointer to string
 |      Line2Ptr       |  pointer to string
 |      But1Ptr        |  pointer to string
 |      But2Ptr        |  pointer to string
 |                     |<- SP

Stack After Call
 |  Previous Contents  |
 |    Button Number    |  integer
 |                     |<- SP
```

This routine displays a dialog box which asks the user to mount a volume. It has two buttons: OK and Cancel.

The pseudo dialog box is displayed at the location specified by whereX and whereY. The contents of the screen under the dialog are saved before the dialog is drawn and restored after the user responds.

All of the text displayed is passed by the application. This way the message can be localized along with the application. Line1Ptr points to a string that will appear at the top of the "dialog". The application could put a message like "Please insert the disk" in this line.

Line2Ptr points to a string that will appear just below line 2. The application could put the volume name on this line. But1Ptr points to a string that will appear in button 1. But2Ptr points to a string that will appear in button 2. The application could put "OK" and "Cancel" in these strings.

When the call returns, it returns the button number chosen by the user. The user can push a button with the mouse or s/he can press RETURN for button 1 or ESC for button 2.

Because the application passes all the text displayed, this call can be used for other kinds of messages. But the application must take care that the keyboard equivalents for the buttons make sense in whatever other use is planned for the dialog box. In addition, the application should take care in selecting text so that it is sure to fit in the space provided.

Note: both QuickDraw and the Event Manager must be initialized for this call to work. The cursor should also be visible.

TLTextMountVolume Call Number $1201

Stack Before Call

Previous Contents	
Space For Result	word
WhereX	integer
WhereY	integer
Line1Ptr	pointer to string
Line2Ptr	pointer to string
But1Ptr	pointer to string
But2Ptr	pointer to string
<-	SP

Stack After Call

Previous Contents	
Button Number	integer
<-	SP

This routine displays messages on the 40 column text screen and waits for the user to press RETURN or ESC.

All of the text displayed is passed by the application. This way the message can be localized along with the application. Line1Ptr points to a string that will appear at the top of the display. Ths application could put a message like "Please insert the disk" in this line. Line2Ptr points to a string that will appear just below line 2. The application could put the volume name on this line. But1Ptr points to a string that will appear right justified on the bottom line. But2Ptr points to a string that will appear left justified on the bottom line. The application could put "OK" and "Cancel" in these strings.

When the call returns, it returns the button number chosen by the user. The user must press RETURN for button 1 or ESC for button 2.

Because the application passes all the text displayed, this call can be used for other kinds of messages. But the application must take care that the keyboard equivalents for the buttons

make sense in whatever other use is planned for the dialog box. In addition, the application should take care in selecting text so that it is sure to fit in the space provided.

Window Release Notes

Changes from last release:

1. Change to GrowWindow and SizeWindow. Both routines were not correctly calculating the the width and height of content regions. The result was a larger content than actually specified by the user. The error was one or two pixels. The problem has been fixed.

2. GrowWindow had been expanding the content region by 1 pixel for the inner xor outline when resizing a window. However, if the content is only one pixel inside the structRgn (no scroll bars in the window frame) the xor line from the content and structRgn overlap and cancel each other out. So, I no longer expand the content to get the inner xor RECT for resizing.

3. ValidRgn and InvalRgn change the given region to global coordinates. This is not desired, but will have to remain this way for compatibility.

4. There is a change in TaskMaster to take advantage of a new feature in the Menu Manager. The Menu Manger is now able to return a selection of a dimmed menu item. See the Menu Manager release notes for a description of the new "Menu Help" feature. The change to TaskMaster is:

   - Bit 14 of the TaskMask field of the Task record should be set to use the new Menu Manager help feature.

   - TaskMaster will return a result of wInactMenu (value $001C) if the user chooses a dimmed item. The high order word of the Task record's TaskData field will contain the ID number of the chosen dimmed menu item.

5. New call added:

   WindowGlobal Call# $560E

input:	chgFlag:WORD	Negative to clear bits, positive to set bit, zero for nothing.
output:	newFlag:WORD	Current global window flag after changed.

   WindowGlobal is used to set and clear bits that effect how the Window Manager performs tasks. If chgFlag has bit 15 set, it is ANDed with the global flag. If chgFlag has bit 15 clear, it is ORed with the global flag. Therefore if chgFlag is zero, the global flag will not change. WindowGlobal will always return the state of the global flag after changing the desired bit.

   Only bit 0 of the window global flag is defined. Bit 0 is defined as a global way to turn window highlighting off and on. The only possible values passed to WindowGlobal are:

$0000	Flag does not change, used to get the current state of the flag.
$0001	Stop the Window Manager from highlighting and unhighlighting windows when NewWindow and CloseWindow. This function is intended to be used to speed up window drawing in some circumstances. See the example below.
$FFFE	Returns the Window Manager to normal highlighting operation.

Here is a possible sequence that could use WindowGlobal:

NewWindow(Param)	Put up a document window. It is now hilited.
WindowGlobal($0001)	Turn hiliting off.
doAlert()	Put up an alert window, let the user choose something, close the alert. The document window, although under the alert window, is never unhilited or hilited. It remains hilited.
WindowGlobal($FFFE)	Return Window Manager to normal operations. The document window is now back on top and is hilited.

6. Added an operation to the Desktop call to make it easier to draw objects on the desktop at any time, not just when called by the Window Manager.

Desktop

input:	operation:WORD	Values 0-7.
	param:LONG	Different parameter for each operation.
output:	result:LONG	Defined by each operation.

Operations 0-6 are defined in the current Tools manual.

Operation 7 has been added and is defined:

Operation	#	Description
BackGroundRgn	7	param = handle of a region.

The region passed will be set to the Desktop's region less any windows. The region is automatically updated when windows are added, removed, sized, and moved.

Operation BackGroundRgn is provided to applications an easy way of drawing objects directly on the desktop. A possible sequence for drawing objects on the desktop might be:

```
pha                         ;Space for result (not used).
pha
pea  SetDeskPat             ;Operation number 5.
```

```
pea   MyDeskDrawl-16          ;Pass address of my routine
pea   MyDeskDraw              ;that will draw the desktop.
_Desktop
pla                           ;Result not used.
pla

pea   MyDeskPortl-16          ;Open a port for my desktop.
pea   MyDeskPort
_OpenPort

pha                           ;Space for result (not used).
pha
pea   BackGroundRgn           ;Operation number 7.
lda   MyDeskPort+VisRgn+2     ;Pass handle of my desktop port's visRgn.
pha
lda   MyDeskPort+VisRgn
pha
_Desktop
pla                           ;Result not used.
pla
```

After the above code is executed, the routine MyDeskDraw will be called by the Window Manager whenever the desktop needs to be drawn. This part of Desktop and the Window Manager has not changed and does provide a way of drawing the desktop. However, there was not a nice way to draw something on the desktop when the application wanted, without making the Window Manager redraw the entire desktop.

That's where operation number 7 comes in. The code above passed a visRgn of a port to the Window Manager. The Window Manager will then use that region to compute the visible desktop. Now when an application wants to draw an object on the desktop, it can switch to MyDeskPort and draw. All drawing will be clipped to the current visible desktop.

Note: The address of the routine passed to Desktop for operation SetDeskPat (number 5) is still a routine that is called with the current port being the Window Manager's with its clip region set to the visible desktop. To draw whenever you want, you'll have to use your own port and your own visRgn.

Apple IIGS System Disk Change History by Component
Version 3.2d8
January 25, 1988

Summary. This is a history of the changes made to the System Disk since version 3.1.

Finder v1.1d1

D7 When the Finder quits to another application, its handle is unlocked and therefore can move. Because of this, running a program which compacts memory and then quits back into the Finder causes the Finder to crash. This has been fixed by setting the fixed block attribute for the handle to the Finder.

ProDOS 8 v1.5d10

D1 Fixed write overrun problem. A write past the 32 MByte boundary now returns the "#POSNERR" error rather than modulo wrapping the mark on the file.

Code crunched and moved to establish patch areas.

D2 When a status call is made, interrupts are now turned off, which of necessity moves around P8 code.

D4 Changed AppleTalk Virtual Disk Driver for correct Language card switching.

Removed AppleTalk boot check by-pass.

Made sure OS_Boot flag is set correctly booting into P16. (When the OS_Boot flag was set incorrectly, this caused the Chooser and Namer from finding the file ATINIT in the /SytstemDisk/System/System.setup subdirectory. This file holds the user's settings of the Chooser and Namer screens).

ProDOS 16

The last version number assigned to ProDOS 16 was v1.3. This new ProDOS 16 is only compatible with the latest ProDOS 8 v1.5 therefore the version number of ProDOS 16 has been bumped up to v1.5 also.

ProDOS V1.5d10

D1 Added special case checking for booting over AppleTalk (PLOADER)

Added RAM Disk Driver for use in booting over AppleTalk (PLOADER)

Added byte of nop to READBLOCK routine in PLOADER for patching by RAM Disk Driver.

D2 PQuit has changed to hook into newly arranged P8 (CALLDISP).

D3 SetUp and DeskAccs files will not be loaded it bit 15 of their AUX_TYPE is set.

A good deal of additional code is enabled if booting over the AppleTalk network. Some of these changes to PLoader include:

a) At the beginning of MAIN, memory is examined to see if ProDOS was loaded with the AppleTalk "Fizzy" boot. If so, a special flag "AT_Boot_Flag" is set to a non-zero value so that all the exceptions to normal read/writing can be applied.

b) Just after the AT_Boot_Flag is set, a patch is applied to PLOADER so that all block reads from PQUIT's READBLOCK routine are now routed to a specially written RAM disk driver whose RAM disk contains P16 and AppleTalk init files.

c) When START is looked for, a "file not found" error is avoided if booting over the network.

d) After P16 (and P8) is loaded, P8 is patched to perform its READBLOCKs through the same RAM disk as PQUIT's READBLOCK.

e) The LogOn init file re-patches P8 to point to its normal device dispatch table.

f) An additional patch is made to READBLOCK in PLOADER so that self-modifying dispatch is never executing (if it did, it would ruin the RAM disk driver patch).

g) Permanent space allocated for a four-byte handle to virtual RAM disk is made at $E1DFF8.

h) Patch is made in E1/PQUIT so switching from P8 to P16 will reload P16 from virtual disk.

AppleTalk and Serial Card interrupts are disabled with store's to $cxxx space upon booting.

D4 Upper cased .SYSTEM string. (Not having .SYSTEM string in upper case prevented user from booting into ProDOS 8 applications)

D5 Broke out AppleTalk Virtual Disk Driver as separate build file.

Fixed SCC bug by re-enabling the oscillator after reset.

D6 Changed error message "Unable to load Tool.Setup file" to a more generic message, "Unable to load System Setup File."

P16 v1.5d10

D1 Fixed null pathname problem in <MOVENAME>.

Fixed absolute addressing mode problem in <MOVE>.

D2 RENAME changed to work on a network volume.

Changed how OPEN checks to see if P8 has allocated a buffer.

CREATE has been moved to bank zero alt language card bank to free up space in bank 1.

Other changes were necessary to hook into newly arranged P8 (CHANGEPATH, TOTENT and PRODOS16.ENTRY ISITOPEN)

D3 Forced Get_Dir_Entry to refresh its directory buffer if preceded by any P16 call, other than itself.

D5 Put Get_Dir_Entry refresh flag in proper language card bank.

D6 Changed CHANGEPATH to recognize $88 "Network Device" rather than $28 "Device Not Connected."

Loader v1.4d1

D1 A bug in the buffering scheme which caused some files with large segments not to be loadable has been fixed.

A load segment with an align of exactly $10000 did not get bank aligned when loaded.

A load segment containg 1 byte could not be loaded (load crashed).

Launcher v2.2d1

D6 Changes were made to the Version Reporter part of the launcher. Four new tools were added to the list of versions retrieved. Tool029 is the Audio Compression and Expansion Tool (A.C.E.). Tool030 and Tool031 are reserved, so cannot be used by anyone internally, and Tool032 is the Midi Tool set.

Tool.Setup v2.2d4

This version of Tool.setup has now been broken up into three smaller files: TOOL.SETUP, TS1, and TS2. TOOL.SETUP is a small file that looks at the ROM and decides which of the other files to load. TS1 is loaded with the original ROM and TS2 is loaded with the new ROM. Nothing is loaded for a ROM that is not one of the first two. This change saves 8 to 10 seconds during boot.

Tool Locator v2.3d2

D1 Algorithm for loading and unloading tools has been changed. See ERS for more details.

New Call Added: SetDefaultTPT, call number $1601.
This call makes default Tool Pointer Table (TPT) equal to the current TPT. Normally, this call would only be made by system software; an application would not need to make this call. It would be used to install a tool patch permanently in the system.

D2 Fixed problem with the tool setup file (TS1) for old ROMS that caused system disk not to boot on old ROM machines. Things were being initialized in the wrong order.

TLMountVolume and TLTextMountVolume have been rewritten to run out of ROM.

D4 In the patch for old ROMs (TS1) the new call SetDefaultTPT was not added to the call table. Therefore ProDOS 8 applications run from the finder would not work on old ROM machines. This has been fixed.

Memory Manager v2.1d2

D2 New Call added RealFreeMem ($2F02)

This call returns the number of bytes that are free and the number of bytes that could be made free by purging. It is different from FreeMem in that it also includes the memory used by purgeable blocks that are not locked. This call gives a program a better idea of how much memory might be available.

Stack before call:

Space for Result	LONG
	Top of Stack

Stack after call:

Total Memory	LONG
	Top of Stack

D4 Fixed bug in SetHandleSize. If memory is compacted while the handle is resized and the original handle moves before the new memory is found, and some other handle is moved into where the original handle is, the resized handle will not have the correct data in it. The problem was that the memory manager did not re-dereference the handle after the compaction.

Desk Manager v2.3d2

D1 No longer uses in-line data storage: All NDA calls changed somewhat to reflect this.

System Task did a CLI instead of a SEI to access certain data with interrupts off.

D8 Call GetNumNDA now works correctly.

Sound Manager v2.2d1

D1 Fixed a bug where memory was getting trashed in bank $00 because the data bank register was not set up properly. That piece of code has been removed.

Patch to FFStopSound that was in its own file has now been incorporated into TS2 (the patch to ROM 2)

ATINIT

D2 This file contains the code to register the users name onto the network at system startup time. It also contains the data that is used by RPM to select which printer is to be chosen when printing, as well as other data structures described in the developer's notes.

D5 ATINIT now checks for debugger ROMs in addition to the enhanced ROMs.

When the message is displayed that you need the proper IIe system, you must reboot. It no longer runs the Selector Application on the workstation disk.

ATPatch (New file for System Disk v3.2.)

This file implements the RPM, PAP, EP, and ZIP appletalk protocals. This file also contains all the patches that were needed to correct bugs in the ROM. A list of patches made for this version follows.

D2 LapInit - changes the LapInit vector so that any calls that are made to LapInit after it is patched will call the old LapInit, and then install RAM based quarter second and packet IRQ vectors.

PacketPatch - this patch is needed to fix a problem in the ROM where the PacketIRQ routine was returning without clearing the NEEDDATA flag if there was no room in the packet buffer. It needs to clear this flag before dispatching packets via PDispatch and before exiting. The patch we installed clears the flag after the interrupt routine, the other patch which clears the flag before dispatching packets is done by the PDispatch patch.

DoAuxFix (PDispatch) - sets up a new vector for dispatching packets. This new vector takes care of making sure that if a packet is to be dispatched, the machine will be in a known state before dispatching the actual data This state is the same as when the quarter second interrupts are handled.

We have also implemented a stack saving routine that determines if the stack is too low, and if so, saves off the required amount of stack space, then sets the stack pointer back up to a higher location to ensure that enough stack will be available to go through all the protocol layers. The limit at which the stack will be saved is currently at $1A0. If the stack gets below this level the save will take place.

Another fix implemented by the new PDispatch is to be compatable with the scheduler. Basically what this means is that AppleTalk will tell the scheduler when it is busy, and therefore you should be able to make AppleTalk calls from other interrupt sources as long as you obey the rules set down by the task scheduler. The main reason for doing this was to allow things such as desk accessories that can make AppleTalk calls. If we did not support the scheduler you might interrupt an AppleTalk in progress and re-enter it when it was not expecting it.

InstallPatch - the AppleTalk dispatcher was patched out to support scheduler compatability on the 'front end' of AppleTalk. We also fixed a bug related to the calling of completion routines where the value at the YSAVE location was not being saved, and was getting changed when calls got nested, thereby destroying the original value.

GetInfoPatch - patched the GetInfo call to change the parameters that were returned. It is now consistent with the documentation.

GetGlobalPatch - fixes bug where GetGlobal would crash upon being called.

QtrPatch - patched the quarter second interrupt vector to support the StackSave call and to support the scheduler.

NBPPatch - fixes bug in ver. 1.0 ROMs where NBP was being case sensitive instead of case insensitive. This fix has been incorporated in ROMs later than 1.0 therefore this patch is only installed in ROMs 1.0 and before.

DDPPatch - fixed bug where DDPWrites would tell you the amount of data you were sending was too much when it really wasn't.

D5 Fixed bug in GetZoneList call where it would not return the proper count for the number of zones found, if it took more than one packet to get the needed information.

D6 Filetype changed from STR to $BC

SPLoad (New file for System Disk v3.2)

D2 This file implements ASP which currently allows up to 8 open sessions at one time.

D6 Filetype changed from STR to $BC

PFILoad (New file for System Disk v3.2)

D2 This file implements the ProDOS Filing Interface which allows ProDOS calls to be made transparently over the network.

D5 Added new pfi command number $35 (FIUSERBUF). This call has the same parameters as the FINAME call and is almost identical with two exeception. The first is that we allow the user to set or get a buffer of 256 bytes in length, and the second is that this buffer can be used for whatever purpose the use wants, not just setting of a name field.

Added new ProDOS 8 command number $45 (ChangePath). This call was installed so that ProDOS 16 could support its own changepath using network volumes. It is not clear whether we want ProDOS 8 users to make this call. (We might want it to be reserved for ProDOS 16's use only.)

Added code to bring PFI up to the level of the BullWinkle ROMs as of 11/19/87.

D6 Filetype changed from STR to $BC

ATStart (New file for System Disk v3.2)

D2 This file is responsible for starting AppleTalk and attaching it to the ProDOS MLI.

D6 Filetype changed from STR to $BC

ATSetup (New file for System Disk v3.2)

D6 This is a new file included for the first time in the D6 release. ATSetup determines if AppleTalk is active, and if so will load all of the other AppleTalk related files. The other files that were of filetype STR are now of type $BC.

Logon v1.0b10 (added in 3.2d6)

Logoff v1.0b7 (added in 3.2d6)

Access v1.0b8 (added in 3.2d6)

Window Manager v2.1 (TOOL014) No Changes

Menu Manager v2.0 (TOOL015) No Changes

Control Manager v2.3 (d6) (TOOL016)

D1 Code was changed so that control manager could now be put into ROM.

Controls are once again drawn in their window's port. This undoes what was once done in v2.1 (7/1/87) of the control manager.

D3 The dither pattern in the scroll bar page region is drawn correctly. This fixes a problem introduced in D1.

When you set pen mode to COPY before drawing boxes, the original mode is first saved and then restored after the drawing. This fixes a problem introduced in D1.

The font ID is saved and restored when switching to and from the icon font.

Here is a rule that has always been true but needs to be made clear:

> Controls are drawn in the grafPort of the window that owns the controls. This means that the state of the grafPort can be set to have some extra control over how controls appear. An example is to change a window's grafPort font to a font other than the system font. When controls with text are drawn in that window, the text will be in the new font. The bad side is that a window's grafPort cannot be left in any old state. For example, if the pen mask is changed then controls will be drawn using the mask which may not be wanted.
>
> If you are drawing in a window that also has controls, always make sure the grafPort is in the state you want controls to be drawn. This does not apply to controls created by TaskMaster because those controls are in the Window Manager's port not the window's.

DragControl now works correctly.

D5 **DisposeControl** returned an incorrect error code. Fixed.

SetCtlTitle did not redraw title for radio buttons and check boxes. Fixed.

Font flags changed by Control Manager affected other text calls in application. Flags are now saved and restored. This corrects a problem introduced in 11/30/87.

Radio buttons, check boxes, and arrows in scroll bars did not draw correctly

when the height of the item was not as high as the icon. Fixed.

Grow box control no longer shares the simple button default color table as its own default table. Its new table is below. Hiliting the size box had resulted in a solid black box, now it is a white icon in a black box.

Previous grow box default color table:
$0000 Black outline for box.
$00F0 Black icon in a white interior when not hilited.
$0000 Black icon in a black interior when hilited.

New grow box default color table:
$0000 Black outline for box.
$00F0 Black icon in a white interior when not hilited.
$000*F* *White* icon in a black interior when hilited.

The color table for the size box control in the manual is not correct. The correct table follows:

growOutline WORD Color of size box's outline:
Bits 8-15 = zero.
Bits 4-7 = outline color.
Bits 0-3 = zero.

growNorBack WORD Color of interior when not hilited.
Bits 8-15 = zero.
Bits 4-7 = background color.
Bits 0-3 = icon color.

Information omitted from manual:
growSelBack WORD Color of interior when hilited.
Bits 8-15 = zero.
Bits 4-7 = background color.
Bits 0-3 = icon color.

Colors in control tables use all four bits of the color in both 640 and 320 modes. Previously only bits 0-1 were used in 640 mode. This could affect existing applications that use color tables in 640 mode. However, there are not any known applications that use color controls in 640 so there should'nt be a problem. The worst case is that a control would be a different color. The benefit of the change is that dithered colors can now be used with controls.

The barArrowBack entry (4th word in the table (offset of 6)) in the scroll bar table is no longer used. The reason it is no longer used is because it was never implemented as it was intended. The color was intended to be the color inside the outline of arrows. However, because arrows are drawn as text characters, only 2 colors can be applied when 3 are needed. The 3 colors would be the arrow box interior, the arrow's outline and the arrow's interior.

The parameter passed to the draw command has its high word zeroed in some cases. This change is to help some code that did not follow the documentation.

TestControl returned a part code even if the control was invisible or inactive. Now if the control is invisible or inactive, TestControl will return zero (no part). (Bug #24328)

MoveControl made invisible controls visible when they were moved. Now MoveControl will move the control but invisible controls will remain invisible. (Bug #24329)

D6 Prototype bit taken out. The version number for the control manager now just reads 2.3.

QuickDraw Auxiliary v2.3d2 (TOOL018)

D1 Fixed bug in picture drawing routines that handled the chextra opcode.

D5 Earlier versions of DrawPicture did not preserve the PurgeStatus of fonts it used: any font used was made nonpurgeable. Now, DrawPicture uses the new font manager call (InstallWithStats) to determine what the PurgeStatus of the font was before it is used and makes it the same when it is done.

Print Manager v2.0d6 (TOOL019)

D3 The call PrChooser has been totally revamped. There is a new user interface and this version of Choose Printer now supports printing over zones.

The Choose Printer dialog now consists of two screens: 1) if direct connect is selected and 2) if AppleTalk is selected. The default is direct connect with the first driver in each of the Printer Drivers and Port Drivers lists selected. Because there are now two different screens that hold settings the user has selected the Printer.Setup file, which saves these changes, has also been slightly modified. Printer.Setup now keeps track of 1) settings for the direct connect screen: Printer Driver and Port Driver selections, 2) settings for the AppleTalk screen: Printer Driver, Zone Name, and Printer Name selections. Also User Name is still saved as well as a flag which tells Chooser Printer what screen to initially put when it is called. Because of this change to the Printer.Setup file, printer.setup files created from previous versions of the print manager will not be compatible and is ignored.

PMStartup no longer requires that any drivers be present or loaded when it is called. This makes startup time for running an application a little faster. Now the drivers are loaded, or checked to see if they need to be loaded, whenever a call is made that needs to be passed on to one of the port or printer drivers.

D4 The print manager no longer crashes when it trys to read an old printer.setup file created by earlier versions of the print manager (v1.2 and earlier). If an old printer.setup file is read the print manager deletes this file and creates a new printer.setup file using the new format and with the new default settings.

If the system disk is locked and you go into Choose Printer and change the current settings, these changes would always get reverted back to whatever the last saved settings in the printer.setup file were once you left the Choose Printer dialog (or changed back to the default settings if no printer.setup file was found). Now any changes you make are saved temporarily until you quit the application.

If you try to save any changes made in the Choose Printer dialog while the disk is locked an alert box will come up saying "Disk is locked, so any changes made to the Choose Printer dialog will not be saved.".

D5 The print manager will now start up even if no drivers are found. (The call PMStartup was modified.)

When a call is made to a driver that is not present the call will be passed to a "dummy driver" location which will determine what the call number is and pop off the required parameters and set the A register with the "Missing Driver" error number. An alert will also come up saying to "Make sure that a printer and a port driver is selected in the Choose Printer dialog." (In the Print Manager v1.0, whenever the last selected driver was not found, the next driver in the drivers list would be selected and loaded for the user.)

Fixed bug in the Dispatch and DPort routines which was not setting the data bank register to the current data bank.

In the Choose Printer dialog you could only see the first twelve files in the drivers folder (if the first twelve were not valid drivers, then you would see less) Problem was current code did not cross block boundaries correctly so now the ProDOS call Get_Dir_Entry is used which solves this problem.

Optimized driver dispatch routine to speed check for driver being in memory.

If not at least one port driver and one printer driver is present in the */system/drivers subdirectory the Choose Printer dialog will put up an alert saying so, and no dialog will appear.

D6 Fixed bug introduced in Printer Manager v2.0d3 where you go from the AppleTalk screen, in the Choose Printer dialog, to the Direct Connect screen and the number of members in the printer names list is greater than the number of members in the Port Drivers list. You get garbage for those members in the Port Drivers list that were in excess.

Fixed problem where the PrinterDriverID flag was not being set correctly in certain instances. This flag tells the print manager whether or not a print driver is loaded in memory or not. (Problems arose when a missing driver error occurred, setting the dispatch address to a dummy location. When you selected a driver that was actually present it would not get loaded because this flag still read that a driver was already loaded.)

Put up watch cursor when user selected "Print" or "Page Setup" and no drivers had been loaded yet. Previously the arrow cursor remained throughout the whole process even though there was a noticeable delay before the print or page setup dialog would come up.

D7 Alert messages now appear properly in 320 mode.

When switching from AppleTalk screen to Direct Connect screen or vice versa, I now blank out the Printer Names/Ports list before displaying any new information. This is also done when changing zones in the AppleTalk mode.

Line Edit v2.0 (TOOL020) No Changes

Dialog Manager v2.1d1 (TOOL021)

D3 A bug in GetNewModalDialog with a refcon value of non-zero would cause a crash or unpredictable result, has been fixed

A bug in IsDialogEvent, which didn't allow it to claim the following events WInDrag, WInGrow, WInGoAway, WInZoom, WInInfo, WInVerScr1, WInHorScr1, WInFrame, and WInDrop, is now fixed.

Scrap Manager v1.2 (TOOL022) No Change

Standard File v2.1d1 (TOOL023)

D1 Fixed bug in a common exit routine which was calling DisposeHandle with a random value. On occasion this could trash the entire system.

Note Synthesizer v1.1d1 (TOOL025)

D1 Fixed bug in AllocGen which caused AllocGen to return invalid generator numbers. It is now guaranteed that AllocGen will return a generator number in the range $00..$0F.

Note Sequencer v0.99 (TOOL026) New for System Disk v3.2

D1 AllocGen was retrieving garbage in the high word. It now only returns values in the range $0-$E.

D2 Multiple phrases are now supported.

Fixed problem with returning errors. WrongTypeErr is no longer returned.

Fixed bug of not being able to startup and shutdown and play a sequence multiple times.

Changed order of start sound call in SeqAppst call.

Caller's pointer in SetInst is not being changed anymore. Instnum now adds two to offset into inst table instead. This fixed the problem of multiple SetInstTable calls.

Fixed problem with the Regesters by aligning the Regester info to the right.

D3 Changed ppatt and chkoff so that if tick count and duration are equal the note will be turned off and the delay will not continue.

If increment is zero, note sequencer doesn't continue parsing but synthesizer does.

Fixed bug dealing with when tick count wraps.

Put in tables for Midi.

D4 LSB now sets mode 0 = interrupts and mode 1 = step mode.
MSB of mode sets midi. 1 = midi use enabled.

MSB of Duration means that the note is awaiting a NoteOff, is not being used, or has wrapped around. If it has wrapped around, the MSB of Track won't be set. Otherwise it will be set.

The high bit of the channel parameter acts the same as the track parameter in the above note.

The high byte of track in set track sets the channel if used with midi.

SETTRKINFO sets up the midi scratch register in teh TRKINFO table to determine if midi is on or off.

All midi info should be working correctly now.

D6 Midi only control command structure has been set up.

Put in midi3bytes, midi2bytes, and midi1byte routines.

Program change and pitch bend have had midi capability added to them.

Fixed wrap bug, if tickcount wraps.

Put in AllNotesOff and FlushBuffer call.

D7 SeqStartUp now shuts SoundTools if an error occured on NoteSynthStartUp.

Changed error checking to match .99P documentation with the exception of:
NoMIDIErr which the sequencer does not use, instead it returns miNoBufErr.

This version of sequencer is now compatible with the latest version (0.3P) of the MIDI Tool.

Added caller error vector error handling for errors in sequence.

Note: There is no check to see if the sequencer is started or not.

Font Manager v2.2d4 (TOOL027)

D2 Major change to FMStartUp call. In previous versions, FMStartUp opened every font file in the FONTS folder in order to construct a number of lists containing information on available fonts - their font ID's, names, etc. This was done every time FMStartUp was called.

FMStartUp now caches these lists, along with some other information, in a file called FONTS.LISTS in the FONTS folder. Subsequent calls to FMStartUp make use of the information in FONTS.LIST, as long as no fonts in the FONTS folder have been added, removed, or altered. In this case the individual font files do not have to be opened by FMStartUp, so the call is much faster.

If FMStartUp detects that fonts have been added, removed, or altered since FONT.LISTS was last updated, then it gets its font information "the old-fashioned was", by opening every font file. When this is done, FMStartUp updates FONT.LISTS. Similarly, if FONT.LISTS does not exist (either because

a user has deleted it, or because the new FMStartUp is being run for the first time on the system in question), then FMStartUp will create it, and then cache the font information on it.

The "validity check" to see if any fonts have been added, removed, or altered since the last update of FONT.LISTS is done by comparing font file names, create dates & times, and last modification dates & times. This information is also cached in FONT.LISTS so that the validity check can be done.

D4 Several changes were made to the implementation in order to enable the code to run from ROM. In particular, all ProDOS 16 calls were made stack-based, and some data areas that had been included in-line with the code were moved into a block of RAM. These changes should be invisible to the calling application and user.

D5 New font manager call: InstallWithStats Call Number: $1C1B

Stack before call:

previous contents	
desiredID	LONG
scaleWord	WORD
ResultPtr	LONG
	Top of Stack

Stack after call:

previous contents	
	Top of Stack

InstallWithStats combines InstallFont with a variant of FindFontStats. InstallWithStats does a normal InstallFont, using desiredID and scaleWord. It also returns (in the buffer pointed to by resultPtr) information about the font it actually installed (which can be different than the font asked for), in the form of a FontStatRec that reflects the stats that the now-installed font had before this call was made.

The call was designed for the use of high-level tools that wish to use the Font Manager in a way that is transparent to the application. In particular, it allows the calling routine to remember the prior purge status of the installed font. InstallFont (and InstallWithStats, as well) can change the purge status of a font it installs; this makes it hard to later restore the purge status, particularly since you don't always know in advance what font an InstallFont call will install (due to the best-fit algorithm, scaling, etc.) InstallWithStats supplies this information.

InstallWithStats is also useful when you just want to know what font the installing call put in; it saves the trouble of an FMGetCurFID call. In this case it is important to note that, even though the fontID part of the returned-FontStatRec is the fontID of the currently installed font, the FontStatBits word contained in the

FontStatRec reflects the recent status of the font, but not necessarily the current status. If you need to know the current status, you'll have to make a FindFontStats call.

D6 Changes to the Font Manager were made to hande the following problem: at FMStartUp time, if the system disk was locked, and FONT.LISTS was either non-existent or not up to date, then when FMStartUp tried to create and/or update FONT.LISTS, it would pass the error up to the application, and abort itself; the Font Manager would not be intialized.

This version pretty much ignores any errors encountered in creating, reading and/or updating the FONT.LISTS file. They do not cause FMStartUp to abort, and no error condition or error code is reported to the application. In the case of bad openings or bad reads, FMStartUp acts as though FONT.LISTS is simply invalid; it ignores FONT.LISTS and tries to get the information it needs from each individual font file. This cure is considerably broader than the disease, because the specific error "Disk Write-Protected" is not special cased - all ProDOS errors concerning FONT.LISTS are handled in this same new way. Since there are so many different ProDOS errors to consider, we should keep an eye out for any bugs that may inadvertently result from this change.

List Manager v2.1 (TOOL028) No Change

A.C.E. v0.2 (TOOL029) Audio Compression and Expansion Toolkit

D6 This is a new tool for system disk 3.2. A.C.E. is a set of utilities used to minimize the amount of storage space or data transfer time necessary to work with digitized audio samples of reasonable fidelity and length. A.C.E. compresses eight-bit digital audio data to half its original size or less, or to expand previously compressed data to its original size (in preparation for playback). By compressing audio data, you expedite its storage and retrieval: not only will any storage medium be able to hold at least twice as much compressed as uncompressed audio, but the effective data transfer speed between the computer's CPU and secondary storage devices, such as floppy disk or hard disk, is twice as fast for compressed as for uncompressed data.

Refer to the A.C.E. ERS for more information.

Midi Tools v0.3 (TOOL032)

D7 This is a new tool for system disk 3.2. The MIDI tool set provides a software interface that allows developers to communicate with external musical synthesizers and other equipment that accepts the MIDI (*M*usical *I*nstrument *D*igital *I*nterface) communication protocol.

Refer to the Midi Tool Set ERS for more information.

ImageWriter Driver v2.0d3

D2 This is a major rewrite of the driver. Low level routines were modified, restructured, or even eliminated. Low level routines that were never published but accessible by the user have now been eliminated.

The following speed enhancements have been incorporated:
a. Faster imaging routines
b. Faster determination that no image is to be printed.

c. Not clearing buffers that were already clear.

iVres now reflects vertical condensed printing.

320 mode landscape with vertical condensed was fixed. It was imaging too high on the page

The job dialog's FROM line edit box was fixed so that the insertion point and editing is correct.

D3 Choices selected in the style dialog and print dialog would not be updated due to internal version number inconsistency.

When the quality of printing selected is "Best Text" the imagewriter driver would intermittently skip lines. This has been fixed.

The call PrChanged has been renamed to GetDeviceName. The PrChanged call used to get the address of the printer device string ("ImageWriter") and then make a call to the port driver that was currently loaded and pass this address to that port to handle. GetDeviceName now just gets the address of the printer device string and returns back to the calling program with this information.

D4 The call GetDeviceName which was first implemented in the previous version has now been changed back to PrChanged. The functionality of this call is the same as it was in v1.3 of the imagewriter driver. This call was changed back to the way it has always been to insure compatibility with the old print manager and appletalk port driver.

LaserWriter v2.0d7

D1 This driver will eventually replace the existing LaserWriter driver. The main feature of GSLaser is its independence from a resident Laser Prep file. This feature will eliminate the need to reinitialize the printer whenever printing is done from machines running different versions of drivers and prep files.

Currently, GSLaser only supports text printing and minimal font style support. Those font styles not supported are: underline, outline, and shadow. Downloading of bitmap fonts are also not supported. Until these features are supported the original LaserWriter driver will continue to be included in the release.

D2 The font styles: underline, outline, and shadow are now supported.

D3 Pixel map printing has been added, but the call PrPixelMap is still not working.

The job dialog's FROM line edit box was fixed so that the insertion point and editing is correct.

D5 The following are now functional:
1. QuickDraw objects: Rects, RoundRects, Ovals, Arcs, Polygons, Lines
2. Patterns
3. GreyScale/Color Text

In the previous version of GSLaser, fonts other than Helvetica, Shaston, Courier,

and Monaco wouldn't print. Now all fonts are printable.

No outline and shadow styles for Courier. (You can select these styles but they just won't do anything while you have Courier selected.) Also, no outline and shadow styles for Monaco if font substitution is on.

D6 Changed name from GSLaser to LaserWriter as all features in the old LaserWriter driver are now implemented.

Status update bug fixed.

This version can now print using LaserWriter Plus fonts.
FontID: (Decimal values, not hex.)
ZapfDingbat = 13
Bookman = 14
Helvetica Narrow = 15
Palatino = 16
ZapfChancery = 18
AvantGarde = 33
NewCenturySchoolbook = 34

D7 picComments processed by the LaserWriter driver:

Name	Kind	Size	Handle
PostScriptBegin	190	0	NIL
PostScriptEnd	191	0	NIL
PostScriptHandle	192	-	PSData
PostScriptFile	193	-	FileName
TextIsPostScript	194	0	NIL

These picComments allow the application to bypass QuickDraw II and send PostScript commands directly to the LaserWriter. The application can use these picComments to take advantage of the advance capabilities of PostScript that is not available through QuickDraw II. The application can also optimize its LaserWriter printing code to create the effects it needs without going through the more generalized QuickDraw II procedures.

There are three ways to send the PostScript commands: the comment handle can point to the actual ASCII data (PostScriptHandle) or point to the pathname of the file which contains the commands (PostScriptFile); or the commands can be sent via QuickDraw StdText calls (TestIsPostScript). The data size of the picComments PostScriptHandle and PostScriptFile should contain the length of the data. Be sure to bracket all Postscript picComments with matching PostScriptBegin and PostScriptEnd to notify the driver of the mode switch.

Warnings:

Unexpected results will occur if these guidelines are not followed:

- Make sure the PostScript commands are syntactically correct since no error checking is performed by the driver on the PostScript text.

- These comments operate within the framework of a single print job.

Normal Print Manager calls should still be used to set up the printing loop.

• The following PostScript operators should not be used since they can alter some print job related conditions set up by the driver:

exitserver
initgraphics
grestoreall
erasepage
showpage

• Terminate the PostScript text with a return character.

• The current transformation matrix has been set by the driver to correspond to the coordinate system and navigational needs of QuickDraw II. Any text printed in this matrix will be flipped vertically across the center of the page, *i. e.* upside down. The application will have to set the current transformation matrix according to its own needs.

• Be "Good PostScript Citizens" as defined by Adobe, *e. g.* define a local dictionary rather than using userdict; leave the persistent parameters alone; etc.

• NEVER nest PostScriptBegin and PostScriptEnds.

Examples of use of these comments can be found in Chapter 3 of the LaserWriter Reference Manual.

Printer Driver v1.0 No Changes

Modem Driver v1.1d1

D1 Code added in DevOpen to initialize printer. Somes files could not be printed without this initialization.

AppleTalk Driver v2.0d2

D3 Several new calls have been added to support printing over zones.

GetZoneList - call number $3113
This call goes out on the net and returns the number of zones found and a list of all the zone names.

GetMyZone - call number $3213
This call returns the name of the zone that your computer is physically hooked up to.

GetPrinterList - call number $3313
This call requires as input a zone name and device name (i.e. LaserWriter, ImageWriter, etc..) and returns a list of all printers of the type you specify and in the zone that you specify.

The call DevPrChanged has been modified so that all it does is check to see if the AppleTalk port has been selected in the control panel. If AppleTalk is not selected an alert box comes up saying, "AppleTalk is not selected in the Control Panel".

Before establishing connection to a device out on the net (i.e. LaserWriter,

ImageWriter, etc...) a network name must be formed which describes what and where this device is that you are looking for. The network name consists of: "PrinterName:DeviceName:ZoneName". The call DevStartup used to accept only one paramter from the calling program which was this network name. Now DevStartup accepts three parameters, a pointer to the Printer Name, Device Name, and Zone Name, which it takes and appends together (leaving out any unnecesary characters at the end of each name) forming the network name.

The strings, printer name, device name, and zone name have been increased from a limit of 16 characters long to 32.

The auxiliary file type of the AppleTalk driver has been changed from 2 to 3.

D4 The paramter block for the appletalk call GetInfo has been changed. In the previous version this parameter block did not reserve space for an extra 2 bytes of data returned from the GetInfo call. This extra 2 bytes of data was spilling over onto the parameter block of the next call, which was GetZoneList. Therefore no zones were found.

Several changes were also made due to changes to the print drivers changing the call GetDeviceName back to PrChanged.

AppleTalk Chooser v1.2b4

D2 Corrected the device type for the "Business Writer" printer so the program now recognizes them.

Created some new overlays.

Selecting "Serial Port" now causes the names window to clear out and also displays a message confirming that the serial port was selected.

A fourth window has been added to handle ports (there are no longer radio buttons to toggle)

If the Chooser is launched from a local volume, the ATINIT file in the root directory of that volume is the one that is used. If the Chooser is launched from a network volume, the ATINIT file used is the one found in the following folder: /User Volume Name/USERS/User Name/. For example: if a user named "FRED" launches the Chooser from the network volume "JUNK", and the first User volume mounted is called "SYSTEM.USERS', the ATINIT file will be found in the following folder: /SYSTEM.USERS/USERT/FRED/.

The User name is now stored on the card by the Chooser program.

The User name is non-editable if the Chooser is launched from a network volume.

Hitting the <ESC> key while in the edit box no longer causes a '_' character to appear.

The Chooser program now conforms to the new ERS (August 11, 1987) in terms of window and button placement, button codes, help text, etc. .

The word "AppleTalk" is now spelled correctly in all places

The Chooser now locates teh correct ATINIT file if launched from ProDOS 16 on a IIGS using built-in AppleTalk (not a workstation or server card).

Added another device type - "AppleTalk ImageWriter I". Changed device type "AppleTalk ImageWriter" to "AppleTalk ImageWriter II".

If the Chooser is run off the network by "<Any User>" and the serial port is selected, this configuration will not be saved in the "<Any User>" ATINIT file.

If the serial port is selected by a user and saved, the next time the Chooser is run the port window will initially show "Serial" highlighted.

Fixed bug where two printers with similar but different names (for example: "aaa" and "aaax") would confuse the program as to which one to highlight as the selected printer.

Expanded the names window to display names 31 characters long.

D4 If a new printer is selected or a new user name is typed, the program will bring up a dialog window asking whether to save the changes or not.

The "AppleTalk" port has been renamed the "LocalTalk" port.

D5 The chooser now looks for the ATINIT file in the ../USERS/user name/Setup/ folder when launched off a network volume.

AppleTalk Namer v1.2b2

D2 A keyboard interface now has been added.

The LQ AppleTalk ImageWriter device was added.

Hitting the <ESC> key while in the edit box no longer causes a '_' character to appear.

The Namer now conforms to the new ERS (August 13, 1987). The "Rename" button has been removed and an "Accept" button put in it's place. Also, the various windows have been moved around slightly.

The program now displays up to 32 different names per device.

Changed device type "AppleTalk ImageWriter" to "AppleTalk ImageWriter II". Changed device type "LQ AppleTalk ImageWriter" to "AppleTalk ImageWriter LQ".

Fixed bug where both the device type window and the names window would scroll simultaneously.

Program now checks if an error occured while renaming the printer.

The user can no longer enter illegal characters (=, :, @, *). The Namer will now beep and not display the character.

Expanded the names window so it will display 31 characters.

Added a delay when confirming renamed LaserWriters so an error wouldn't be incorrectly reported.

AppleTalk IWEM v1.2 No Changes

System Utilities v3.1d9

D1 Before deleting all the files in a subdirectory, system utilities asks the user for permission to delete all locked files in the subdirectory without prompting for each one. No matter which way the user replied, system utilities would prompt for each locked file. Now if the user selects "delete ALL locked files", they are all deleted with no further prompts.

System utiliites did not return an error when the user tried to save the serial port settings on a write-protected disk. It now checks for an error when saving the settings file and displays the appropriate error message should one arise.

When "Copy Files" copied a file, it would not detect the end-of-file properly and would perform about 60 writes of zero bytes (doing nothing). "Copy Files" now detects the correct end-of-file and stops there.

On the "Rocky and Bullwinkle" network, deleting a list of files that was nested several subdirectories deep would not work (It would delete every other file, then give an I/O error). All files can now be deleted correctly across the network.

D2 The tree walking routines used in "Delete Files" and in "Copy Files" were rewritten for speed. The new ones are about 2.5 times as fast as 3.1d1 and about 20% faster than System Utilities 3.0.

Network devices are removed from the device table on startup and restored on exit. This allows System Utilities to work with the newest version of the Rocky & Bullwinkle firmware and software.

Saving serial port settings on a write protected disk did not generate an error. Now it does.

Copying files from an unformatted disk or a Macintosh formatted disk use to hang. Now it generates a read/write error.

Renaming a file to a duplicate name aborted the entire rename process. It now brings up the screen "Enter New Name".

When duplicating or formatting a disk, the fifteenth character of the default name was dropped. This no longer happens.

Illegal volume names were accepted when duplicating disks. Now a "Bad Volume Name" error is returned.

Copying a large number of files resulted in an endless repetition of copying the first file. System Utilities now copies all the files and only once.

When copying or deleting files, if the names did not fit in memory an error is returned. However, if the names were such that the buffer would be exactly filled at one point, the error would not be detected and System Utilities would crash.

When asked to insert a source or destination disk on single drive operations, System Utilities aborted the operation if you did not insert a disk. Now it prompts the user to insert a formatted disk.

System Utilities now remembers the last choice the user makes between "Slot & Drive" and "ProDOS Pathname" and uses this as the default.

No error message was put up when trying to copy subdirectories that were nested too deeply. Now an error message is put up and then the process is aborted.

Duplicating a ProDOS disk onto a Pascal one now works.

For the operations Format, Verify, and Rename Volumes the user was prompted with having to enter "Slot and Drive". Before being able to actually enter a slot and drive number the user had to press return. This unnecessary step has now been eliminated.

The user was unable to abort from the "Renaming Files" process. Now when the <ESC> key is hit the renaming process is aborted.

The word "Done" is now displayed in the correct place after a single drive duplication.

When duplicating a DOS 3.3 disk, System Utilities would prompt for a DOS 3.3 volume name, and thus produce garbage on the screen. Utilities no longer prompts for a DOS 3.3 volume name.

Listing volumes and then cataloging a DOS 3.3 disk no longer crashes.

"Rename Files" now skips the rename call and continues to the next name if the original file name and the new file name are the same. This fixed a duplicate name error when renaming pascal files.

A breakpoint was left in the program in version 3.1d3. This caused System Utilities to crash after deleting or copying files. This has been taken out for v3.1d4.

Some minor code crunching was done and some obsolete strings were removed. This was done to provide space for the bug fixes and enhancements.

D4 When loading System Utilities from BASIC on an enhanced IIc, mousetext was not turned on. Now it is. (Bug ACH002RBU)

When translating files between operating systems, one dialog box had "cannot" misspelled as "can not". (Bug DAC034RBU)

System Utilities trashed memory whenever the "Please insert a disk dialog" was displayed. This caused several problems, all of which are now fixed. (Bug ELCUTL006)

When an error occurred in "Duplicate a Disk", trying to duplicate another disk would not work properly. This has been fixed. (Bug DAC026RBU)

System Utilities no longer prints "Done!" when you escape out of "Duplicate a Disk". (Bug DAC027RBU)

Copy files aborted after 198 subdirectories had been copied. Copy files will now copy as many subdirectories as needed. (Bug DAC011RBU)

Format a disk now allows a fifteen character volume name. (Bug DAC032RBU)

Copying files with a single drive would not work when copying subdirectories. (Bug DAC002RBU)

ESC now returns user to the main menu when it finishes from setting serial ports. (Bug DAC024RBU)

The "About System Utilities" function left two bytes on the stack. This would cause a crash after about 130 accesses. This has been fixed. (Bug DAC025RBU)

System Utilities lost serial port settings if you exited the "Set Serial Port" section. Now it will retain these settings until the next time you launch System Utilities and reenter the "Set Serial Port" section. (Bug WMP004WMP)

When copying files to network volumes, System Utilities would think the disk was full before it really was. Now System Utilities does not do any size checking on network volumes, so if the files won't fit the user doesn't get a message until the disk is already full.

In v3.1d1 of System Utilities a bug was introduced when copying large files (>513 blocks). The bug would cause large files to be truncated to 257 blocks when copied. This has now been fixed.

For the operations Format, Verify, and Rename Volumes the user was prompted with having to enter "Slot and Drive". Before being able to actually enter a slot and drive number the user had to press return. This unnecessary step has now been put back in. (Undoes a change made back in v3.2d2 of the system disk.)

D5 When a copy from one directory to another on the same disk was made, System Utilities thought this was a single drive copy where the disks needed to be swapped. This has been fixed. (Bug DAC047RBU and DAC048RBU)

Changed the "Duplicate Volume" error message to be more specific.

System Utilities crashed after running out of memory when deleting files. This problem has been fixed. (Bug JGM001RBU)

XON/XOFF were not being set correctly. System Utilities now sets these correctly, but the firmware does not use the default settings properly. This means there is no easy way to verify System Utilities is setting this byte correctly.

Copying DOS 3.3 files cleared the high bit of the file names, making the copies

inaccessable. This has been fixed.

The message that a file is locked, would sometimes overlap the boundaries of the box when DOS 3.3 files were being deleted. This box has been expanded and the message is now centered in the box.

When copying subdirectories from ProDOS pathname to ProDOS pathname, System Utilities needlessly prompted for the source and destination disks. Now it doesn't.

D6 When duplicating a disk, System Utilities gave an error if there was a slash on the end of the name. Now it longer does.

XOn/XOff has now been fixed. System Utilities now sets the correct byte in the screen holes, so the firmware will enable XOn/XOff.

When deleting files from the network, System Utilities got confused if it tried to delete a file inside a folder that had "make changes" disabled. Now utilities puts up a message saying the disk is write-protected and cancels that particular file.

More fixes to "Copy Files". Now Pascal and DOS 3.3 files are handled correctly, that is, they have the high bit set and cleared respectively.

Fixed DOS 3.3 renaming problems. Now the new DOS 3.3 name can contain up to 30 characters instead of being prematurely truncated at 16. Also, they can now contain slashes and spaces.

Fixed ProDOS to Pascal file transfers. If the source file was an odd number of blocks, the destination file used to be padded with about 32,000 blocks of zeroes. This would produce either a disk full error or an I/O error.

Fixed Pascal file copy bug. When copying over an already existing pascal file, System Utilities displayed a read/write error. This was due to improper deletion of the already existing file.

Copying over particular DOS 3.3 files generated an endless loop. This was due to improper error reporting from DOS 3.3 delete files. (Bug #DAC061RBU)

Fixed Pascal to ProDOS copy files problem. When System Utilities transferred files from Pascal to ProDOS, it would cut out pieces of the file at semi-random intervals. Now you can transfer a file from ProDOS to Pascal and back, compare the two files and get an identical result.

BASIC.SYSTEM (ProDOS AppleSoft Command Interpreter) v1.2b3

D2 Change CATALOG in <CMDS2>, B076 from BEQ ENDCAT to NOP/NOP, B07C from BCS CCATERR ($B0 $3A), and B0A1 from BPL ($10) to BCC ($90). The net effect of these changes is to fix a bug reported by Paramus (AppleTalk) in which a conflict occurs between TOTENT, the word specifying the number of entries in the ProDOS directory and the actual enumerated entries.

Splash screen changed to "1.2" and "COPYRIGHT APPLE, 1983-87"

D5 Control S on a CATALOG prematurely terminated if a subsequent "space" is

pressed. This has been fixed.

BASIC.Launcher v2.0 No Changes

Fonts v1.0
Courier, Geneva, Helvetica, Shaston, Times, Venice No Changes

D6 All the 12 point fonts have been removed.

Graphics Image File Format Standard for APPLE II GS
February 18, 1987

Unpacked format

PRODOS FILETYPE: $C1 AuxType: $0000

This type will contain a file with a full 32K Picture image - unpacked form. So far, this is the only one established and supported by APPLE. Currently, it is being used in PAINTWORKS from ACTIVISION also. Note: The first release of PAINTWORKS assumes the pallets colors have been ordered from highest to lowest luminence.

AuxType: $0001 -> $FFFF

These have not been defined as of yet and their assignment and administration will be set up by APPLE.

Packed format

PRODOS FILETYPE: $C0 AuxType: $0000

ACTIVISION file format. This has been assigned to ACTIVISION and is used as a special format for their first release of PAINTWORKS. Its format is as follows:

Bytes $000-$01F: Pallet
Bytes $020-$021: Background color
Bytes $022-$221: 16 Patterns, 32 bytes each
Bytes $222-...: Packed Picture

PRODOS FILETYPE: $C0 AuxType: $0001

PACKBYTES type packed file format. This type of file is created by passing the full 32K image (including SCB's and Color tables) to the _PACKBYTES ROM routine of the Miscellaneous tools toolset. N.B. it can be passed as sequential smaller buffers as defined in the _PACKBYTES routine.

PRODOS FILETYPE: $C0 AuxType: $0002

APPLE Prefered standard interface format. (This is what we recommend all developers to use - and be able to support). The file is composed of a series of blocks. Each block has the same format:

Length	LongInt (includes the size of longint too)
Kind	string with length byte (this string is case sensitive - upper case is recommend to avoid confusion.)
Block Specific Data	Variable amount of data

Below we define some standard blocks that most applications will want to put in the file. Other blocks can be defined by individual applications. Using this scheme, the format is extensible. Applications with differing needs can store everything they need about a document in a file, in a way that will allow other applications to ignore the information it cannot deal with.

The predefined blocks are MAIN, PATS, and SCIB. The MAIN block should be in every file of this format and every application which supports this format should be able to read a MAIN info block. The PATS block contains patterns which may be associated with the picture. The SCIB block contains information relating to the current drawing pattern for the "document." These are used by paint programs that want to save the foreground pattern, a background pattern and a frame pattern with the image.

The Pallets in the MainInfo block are numbered from 0 and correspond to the modes in scan line directory. If there are more than 16 palletes, the remaining ones are dealt with arbitrarily by the application.

The MAIN info block:

SizeOfBlock	LongInt	
IDString	str 'MAIN'	
Master Mode	ModeWord	(This comes from QuickDraw's MasterSCB)
PixelsPerScanLine	integer	(Must not be zero)
NumPallets	integer	(May be zero)
PalletArray	[0..NumPallets-1] of Pallet	
NumScanLines	integer	(Must not be zero)
ScanLineDirectory	[0..NumScanLines-1] of DirEntry	
PackedScanlines	[0..NumScanLines-1] of PackedData	

The PATS info block:

SizeOfBlock	LongInt
IDString	str 'PATS'
NumPats	integer
PatternArray	[0...NumPats-1] of PatternData

The SCIB info block:

SizeOfBlock	LongInt
IDString	str 'SCIB'
Foreground Pattern	PatternData
Background Pattern	PatternData
Frame Pattern	PatternData

Data Types:

Integer	word	16 bit signed quantity
LongInt	LONG	32 bit signed quantity
ColorEntry	word	The nibbles in the word are interpreted as RGB values as follows: $0RGB. The high nibble of the high byte must be 0. The low nibble of the high byte is the value for red. The high nibble of the low byte is the value for green, the low nibble of the low byte is the value for red.
Pallet:	array [0..15] of ColorEntry	
ModeWord	word	if high byte = 0 then low byte is the mode bit portion of the SCB for the scanline. Specifically, the low four bits which determine the pallete number and bit 7 which determines the mode are valide. Other bits are not valid and must be zero. Other modes not yet defined.
DirEntry:	Number of bytes to unpack : integer Mode : ModeWord	
PatternData:	32 bytes of pattern information	

Compact File

Version 1.0 B6

February 17, 1987

This utility will compact a Load File to make it smaller and load faster. It does this by converting Load Files which use Version 1 Object Module Format (OMF) to Version 2 and by compressing Version 2 OMF files through the use of SUPER compressed Load records.

WARNING!!! Files converted with COMPACT can only be run on systems using System Loader version 1.2 which has not been released yet. The System Loader version is displayed under the ProDOS version during the boot process.

COMPACT will also create certain Load Segment types that are not available under Version 1 OMF:

Reload Segments
No Special Memory Segments.

One use for Reload Segments is to make applications written in C restartable from memory. For example, if a Shell utility is written in C, it can be run through COMPACT which will convert it and mark the ~globals and ~arrays segments as Reload Segments. When this utility is "installed" into APW, it could be marked as "restartable" from memory. Restarting will work because the System Loader will restart the program segments of the utility from memory but will reload the data segments from the file so that all the utility's data is reinitialized.

Currently, COMPACT will perform these operations:

- RELOC records will be converted to cRELOC records if eligible.
- INTERSEG records will be converted to cINTERSEG records if eligible.
- SUPER compressed records will be created replacing all eligible INTERSEG, cINTERSEG and cRELOC records.
- Version 1 Segment Headers will be converted to Version 2 format:
    - Old KIND field converted to new KIND field
    - Old KIND=$1E will cause No Special Memory bit to be set

- Old KIND=$1F will cause Reload Segment bit to be set
- BLKCNT will be converted to BYTECNT
- VERSION will be changed from 1 to 2

- Version 1 Segments which are on block boundaries will be packed one after the other as defined in Version 2
- Data segments ~globals and ~arrays will be made into "reload" segments

Here is an example of the impact of a compaction:

C Compiler before compaction	C Compiler after compaction
Size: 319 blocks	225 blocks (29% smaller)
Load Time: 18 secs (from SCSI HD20)	14 secs (22% faster)

Calling Sequence:

COMPACT inputfile [outputfile]

Parameters:

inputfile - Load File to compact (may contain wildcards)
outputfile - New Load File to create

If only the "inputfile" is specified, the file will be converted in place. If a wildcard is specified in "inputfile", all files that satisfy the wildcard will be converted. Note: the wildcard facility can only be used in the "inputfile" with no "outputfile" specified. For example:

COMPACT /x/y/=

will convert all files in subdirectory "/x/y" that are Load Files (i.e. File Types $B3-$BE). Files which are not Load Files will be ignored.

QuickDraw Auxiliary Tool
External Reference Specification

Steven Glass

February 11, 1987

December 18, 1986	Initial Release.
February 11, 1987	Added DrawIcon.

QuickDraw Auxiliary Tool Set

The QuickDraw Auxiliary Tool Set contains functions and features that did not appear in the QuickDraw Tool Set for time and/or space reasons. These include:

CopyPixels	A call similar to the Macitosh QuickDraw call CopyBits which stretches or compresses images to make them fit in the indicated destination rectangle.
WaitCursor	This call changes the cursor from whatever it is to the watch cursor.
Pictures	These are the routines that record and play back drawing commands. The entry points to these routines are part of the QuicDraw tool set but through various forms of black magic, the code which does the work is part of this tool set.
Text Styling	The QuickDraw Tool Set knows how to make a font bold and to underline it. Macintosh QuickDraw supports outline, shadow, italics. With the QDAux toolset loaded you QuickDraw II can outline text and in the future will be able to shadow and italicize too.
DrawIcon	This routine is used to draw an icon in the current port.

QuickDraw Auxiliary Housekeeping Routines

QDAuxBootInit Call Number **$0112**

Internal routine called at load time to initialize the QuickDraw Auxiliary Tool Set.

No Stack Parameters

QDAuxStartup Call Number **$0212**

Call made by an application if it wants to use the QuickDraw Auxiliary routines.

No Stack Parameters

QuickDraw must already be initialized when this call is made. This call links the QDAux tool set into QuickDraw by intercepting many of QuickDraw's low level vectors. The shutdown call unlinks the tool set restoring QuickDraws original vectors.

The QuickDraw Auxiliary Tool Set does not need any additional direct page. When it needs space it shares the three pages already assigned to QuickDraw.

QDAuxShutdown Call Number **$0312**

Call made by an application to shutdown the QDAux tool set.

No Stack Parameters

This routine unlinks the QDAux tool set with QuickDraw putting back the original QuickDraw low level vectors.

QDAuxVersion Call Number **$0412**

Returns the version number of the tool set.

Stack Before Call

previous contents	
space for version	
	<-SP

Stack After Call

previous contents	
version number	
	<-SP

QDAuxReset Call Number **$0512**

Resets the Tool Set.

No Stack Parameters

This is called when the system is reset (CONTROL-RESET is pressed).

Returns true if the tool set is active, false otherwise.

Stack Before Call

```
|  previous contents  |
|  space for boolean  |
|                     |<-SP
```

Stack After Call

```
|  previous contents  |
|    boolean result   |
|                     |<-SP
```

CopyPixels Call Number $0912

Copies a pixel map from one place to another stretching and/or compressing as necessary to make the source pixels fit in the destination rectangle

Stack Before Call

previous contents	
SrcLocPtr	POINTER
DestLocPtr	POINTER
SrcRect	POINTER
DestRect	POINTER
XferMode	Pen Mode (WORD)
MaskRgn	HANDLE
	<-SP

Stack After Call

previous contents	
	<-SP

If the destination LocInfo record is the same as the LocInfo record of the current port, the pixels are also clipped to the current port's VisRgn and ClipRgn.

WaitCursor Call Number $0A12

Changes the cursor to the watch cursor. You can restore it to the Arrow cursor with an InitCursor call.

No Stack Parameters

A desk accessory or tool can make this call even if it does not know if this tool set is loaded or active. Since the call has no stack inputs, making the call an not checking for errors will not cause any trouble. If the tool is not installed, the dispatcher will return an error and nothing will happen.

QuickDraw Routines Available because the QDAux Tool Set is Active.

The picture routines are routines in QuickDraw, but do not do anything unless the QDAux tool set is active.

Picture Overview. A picture is a record of drawing commands. To begin recording commands you use the OpenPicture call. OpenPicture returns to you a handle to the picture being created. It also establishes the picture frame which is used in conjunction with the destination rectangle to map objects from one space to another when the picture is drawn.

The mapping occurs as follows. If the picture frame is [0,0,100,100] and the destination rectangle is [50,50,60,60], a line recorded from (10,10) to (90,90) would appear from (51,51) to (59,59) when the picture is drawn.

The size of any text drawn is also scaled according to the picture frame to the destination rectangle. Using the example above, a text size of 60 would appear as 6 when the picture is drawn. When the horizontal and vertical coordinates are not scaled an equal amount, a question arrises of how to choose an appropirate scaling factor for the font. We just use the vertical change to select the correct sized font.

The font manager handles all the font scaling so must be loaded and intialized before any DrawPicture command is issued.

One of the most common uses of pictures is for printing. The Print Manager uses pictures to record what you want to put on a page. Then it plays back the picture over and over again into a band buffer that is then printed. This way your application only has to record the drawing commands once and the print driver can use the picture for the specific band buffer it needs to deal with.

Another common use for pictures is data interchange between applications. Pictures are one of the most common ways applications pass information back and forth between one another. It is one of the two standard types definded by the scrap manager for the clipboard.

The Picture Calls.

OpenPicture Call Number $B704

OpenPicture allocates memory for recording of quickdraw calls. It returns a handle to the picture that is opened.

```
Stack Before Call
     |  previous contents     |
     |  Space for Handle      | LONG
     |     PicFrame           | POINTER

Stack After Call
     |  previous contents     |
     |    Picture Handle      | HANDLE
     |                        |<-SP
```

OpenPicture also calls HidePen, so no drawing occurs on the screen while the picture is open (unless you call ShowPen just after OpenPicture, or you called ShowPen previously without balancing it by a call to HidePen).

When a picture is open, the current grafPort's PicSave field contains a handle to information related to the picture definition. If you want to temporarily disable the collection of routine calls and picture comments, you can save the current value of this field, set the field to NIL, and later restore the saved value to resume the picture definition.

You cannot call OpenPicture when a picture is already open.

> **Warning**: A grafPort's ClipRgn is initialized to an arbitrarily large region. You should always change the clipRgn to a smaller region before calling OpenPicture to guarentee that when drawing occurs and the ClipRgn is mapped from the PicFrame to the DestRect, it is still valid.

ClosePicture Call Number $B904

ClosePicture tells QuickDraw to stop saving drawing calls and picture comments into the currently opened picture.

No Stack Parameters

You should perform one and only one ClosePicture for every OpenPicture. ClosePicture calls ShowPen, balancing the HidePen call made by OpenPicture.

PicComment Call Number $B804

Stack Before Call

previous contents	
Kind	integer
DataSize	integer
DataHandle	HANDLE to data

Stack After Call

previous contents	
	<-SP

PicComment inserts the specified comment into the currently open picture. The kind parameter identifies the type of comment. DataHandle is a handle to additional data if desired and DataSize is the size of the data in bytes.

If there's no additional data for the commant, DataHandle should be NIL and DataSize should be 0. An application that process the comments must inlcude a procedure to do the processing and store a pointer to it in the data structure pinted to by the GrafProcs field of the GrafPort.

> Note: The standard low-level procedure for processing picture comments simply ignores all comments.

DrawPicture Call Number $BA04

Stack Before Call

```
|   previous contents     |
|      PicHandle          | HANDLE
|      DestREct           | POINTER
```

Stack After Call

```
|   previous contents     |
|                         |<-SP
```

DrawPicture takes the recorded drawing commands and maps them from the picture frame into the destination rectangle (specified here) and draws them.

DrawPicture passes any pictuer comments to a low-level procedure accessed indirectly through the GrafProcs field of the GrafPort (see PicComment above).

> **Warning**: If you call DrawPicture with the intial arbitraily large clip Rgn and the destination rectangle is offset, or larger from the picture frame, you may end up with an empty ClipRgn and no drawing will be done.

KillPicture Call Number $BB04

KillPicture releases all the memory occupied by the given pciture. Use this only when you're completely through with a picture.

Stack Before Call

```
|   previous contents     |
|      PicHandle          | HANDLE
```

Stack After Call

```
|   previous contents     |
|                         |<-SP
```

DrawIcon Call Number $0B12

Draws the specified icon in the specified mode at the specified location clipping tot the current clip and vis regions. This routine will not contribute to a picture nor get printed.

Stack Before Call

previous contents	
IconPtr	POINTER
DisplayMode	WORD
XPosition	integer
YPosition	integer

Stack After Call

previous contents	
	<-SP

The icon data structure is as follows:

IconType	word	Bit 15 is used to indicate whether the icon image has color or black and white information in it. Setting bit 15 to one implies that it is a color icon and cannot be colored otherwise.
IconSize	integer	This is the number of bytes in the icon image and mask.
IconHeight	integer	This is the height of the icon in pixels.
IconWidth	integer	This is the width of the icon in pixels.
IconImage	bytes[IconSize]	This is the icon image. It is IconSize bytes long. Each row of pixels is 1+(IconWidth-1)/2 bytes wide.
IconMask	bytes[IconSize]	This is the mask. It is IconSize bytes long. Each row of pixels is 1+(IconWidth-1)/2 bytes wide.

The display mode word has the following fields.

Bit 0	Set implies the icon is SELECTED.
Bit 1	Set implies the icon is OPEN.
Bit 2	Set implies the icon is OFF LINE.
Bits 3 thru 7	are reserved.
Bits 8 thru 11	are the foreground color to be applied to the black part of black and white icons.
Bits 12 thru 15	are the background color to be applied to the white part of black and white icons.

When the mode word is zero, the icon is copied to the destination through the specified mask. When bit 0 is set, it image is inverted before copying. When bit 1 is set, a light grey pattern is copied instead of the specified image. When bit 2 is set, the light grey pattern is ANDed to the image being copied. All three of these operations can occur at once and the order the testing occurs in is

bit 1 (is it open)
bit 2 (is it off line)

bit 0 (is it selected)

Color is only applied to the black and white icons if bits 8 thru 15 are not all zero.

Color inversion. Colored pixels in an icon are invered as follows:

Black Pixels become white
Any other color becomes black

Documentation Développeurs
Apple Computer France 1987

Document développeur numéro 51

Miscellaneous Tools

type d'upgrade de ce ducument : 5
1 Documentation de première catégorie inchangée
2 Documentation de deuxième catégorie mise à jour
3 Documentation de deuxième catégorie inchangée
4 Mise à jour payante de la documentation de première catégorie
5 Mise à jour gratuite de la documentation de première catégorie
6 Nouveautés payantes non vitales
7 Nouveautés gratuites et vitales

Taille : 41 page(s) environ

Domaine : Tool 03

VERSION : 00:93
DATE : 29.10.86

Cortland Miscellaneous Tools

October 29, 1986

Written by Ray Montagne & Eagle Berns

Revision History.....

March 10, 1986	Ver. 0.81	R. Montagne	Major revisions to the Miscellaneous Tool Set have occured. The Integer Math functions have been removed, and now comprise the INTEGER MATH TOOL SET. The Pascal and Basic I/O functions have also been removed, and are now found in the TEXT TOOL SET. A stack pointer indicator (sp—>) has been added to the parameter lists for clarity. Basic functionality of most tool functions remaining in the MISCELLANEOUS TOOL SET has not changed. However, all of the function numbers have changed. Many of the functional descriptions have been rewritten for clarity. Functions that have changed are: (1) INTERRUPT CONTROL TOOLS (2) FIRMWARE FLAG TOOLS (3) INTERRUPT ENABLE STATUS TOOLS
March 12, 1986	Ver. 0.82	R. Montagne	System Death Manager. Additional information on interrupts source control (Keyboard interrupts). Additional information on the environment when using Firmware Entry. Additional information on installing ROM based tasks into the HeartBeat queue.
April 18, 1986	Ver. 0.83	R. Montagne	ID Manager Type 8. System Death Error Codes. Additional Vectors.
April 23, 1986	Ver. 0.84	R. Montagne	Added vectors for STEP and TRACE. Additional parameters in the GET ADDRESS function. This is the BETA 2.0 Implementation.
April 29, 1986	Ver. 0.85	R. Montagne	Added functions to set and get clamps for absolute devices. Mouse calls will return an error if the card is not switched in rather than call the system death manager.
May 9, 1986	Ver. 0.86	R. Montagne	ID Manager ID assignments. Read ASCII time (state of MSB). Set/Get Vectors.
June 11, 1986	Ver. 0.87	R. Montagne	Added SETUP FILE ID to ID Manager.
June 26, 1986	Ver. 0.88	R. Montagne	No functional change, just added examples.
July 9, 1986	Ver. 0.89	Montagne/Berns	Added SCRAP MANAGER ID tag. Eagle's Examples!! System Death Syntax. System Death messages and Language Card. More descriptive death codes. Stack example in MUNGER was corrected.
July 16, 1986	Ver. 0.90	Montagne	Corrected tool number in death codes.

Aug 20, 1986	Ver. 0.91	Montagne	WriteHexTime example. One Second interrupt handler example. 1/4 Second interrupt handler example. HeartBeat examples.
Oct 20, 1986	Ver. 0.92	Berns	Fix example in Pack/Unpack bytes.
Oct 29, 1986	Ver. 0.93	Montagne	Added a function called SysBeep.

Miscellaneous Tools. So far the tools we have specified fall into broad catagories and each deserve their own tool set. Unfortunately, ther are a number of routines in the firmware that do not fall into any of these categories but still must be accessed from native mode. These routines include:

APPLE][entry points	Mouse support	Clock support
Battery ram support	Interrupt support	ID Tag managment
VBL or HeartBeat managment	System Death managment	

Standard Tool Set Calls.

MTBootInit Function number = $01

This tool call clears the TickCounter and the HeartBeat task link pointer. It also sets the Mouse flag to 'NOT FOUND'. A block of memory with a length of NIL is requested from the memory manager for use by the ID tag manager.

Example:

```
_MTBOOTINIT
```

MTStartUp Function number = $02

This does nothing.

Example:

```
_MTSTARTUP
```

MTShutDown Function number = $03

This does nothing.

Example:

```
_MTSHUTDOWN
```

MTVersion Function number = $04

Input Word Space for result
sp—>

Output Word Version number
sp—>

This tool returns the version number of the Miscellaneous Tool Set.

Example:

```
PEA         $0000       ; SPACE FOR RESULT
_MTBOOTINIT
```

MTReset Function number = $05

This tool call clears the HeartBeat queue link pointer and sets the Mouse flag to 'NOT FOUND'.

Example:

```
_MTRESET
```

MTStatus Function number = $06

Input Word Space for result
sp—>

Output Word Status ($0000=Inactive, $FFFF=Active)
sp—>

This tool returns a status that indicates that the Miscellaneous Tool Set is active.

Example:

```
PEA         $0000       ; SPACE FOR RESULT
_MTSTATUS
```

MTSpare1 Function number = $07

This does nothing.

Example:

```
_MTSPARE1
```

MTSpare2 Function number = $08

This does nothing.

Example:

```
_MTSPARE2
```

Battery Ram Tools. These routines allow the non volatile battery backed up ram to be read or written.

WriteBRam Function number = $09

	Input	LongWord	Buffer Address
sp—>			

The 252 bytes of data at the memory location specified by the **Buffer Address** plus four bytes of checksum data is written to the battery ram.

Example:

```
PUSHLONG   #LABEL        ; BUFFER ADDRESS
_WRITEBRAM
```

ReadBRam Function number = $0A

	Input	LongWord	Buffer Address
sp—>			

The 252 bytes of data plus four bytes of checksum data is read from the battery ram and stored at the memory location specified by the **Buffer Address.**

Example:

```
PUSHLONG   #LABEL        ; BUFFER ADDRESS
_READBRAM
```

WriteBParam Function number = $0B

	Input	Word	Data (low byte only)
	Input	Word	Parameter Reference Number (0-255)
sp—>			

Data is written to the battery ram location specified by the **Parameter Reference Number.**

Example:

```
PEA        $0005         ; DATA IN LOW BYTE
PEA        $0028         ; REF = STARTUP SLOT
_WRITEBPARAM
```

ReadBParam Function number = $0C

	Input	Word	Space for result
	Input	Word	Parameter Reference Number (0-255)
sp—>			
	Output	Word	Data (low byte only)
sp—>			

Example:

```
PEA         $0000       ; SPACE FOR RESULT
PEA         $0028       ; REF = STARTUP SLOT
_READBPARAM
```

Data is read from the battery ram location specified by the **Parameter Reference Number.**

Battery Ram Parameter Reference Numbers:

$00	Port 1 Printer / Modem
$01	Port 1 Line Length
$02	Port 1 Delete line feed after carriage return
$03	Port 1 Add line feed after carriage return
$04	Port 1 Echo
$05	Port 1 Buffer
$06	Port 1 Baud
$07	Port 1 Data / Stop Bits
$08	Port 1 Parity
$09	Port 1 DCD Handshake
$0A	Port 1 DSR Handshake
$0B	Port 1 Xon / Xoff Handshake
$0C	Port 2 Printer / Modem
$0D	Port 2 Line Length
$0E	Port 2 Delete line feed after carriage return
$0F	Port 2 Add line feed after carriage return
$10	Port 2 Echo
$11	Port 2 Buffer
$12	Port 2 Baud
$13	Port 2 Data / Stop Bits
$14	Port 2 Parity
$15	Port 2 DCD Handshake
$16	Port 2 DSR Handshake
$17	Port 2 Xon / Xoff Handshake
$18	Display Color / Monochrome
$19	Display 40 / 80 column
$1A	Display Text Color
$1B	Display Background Color
$1C	Display Border Color
$1D	50 / 60 Hertz
$1E	User Volume
$1F	Bell Volume
$20	System Speed
$21	Slot 1 Internal / External
$22	Slot 2 Internal / External
$23	Slot 3 Internal / External
$24	Slot 4 Internal / External
$25	Slot 5 Internal / External
$26	Slot 6 Internal / External
$27	Slot 7 Internal / External
$28	Startup Slot
$29	Text Display Language
$2A	Keyboard Language
$2B	Keyboard Buffering
$2C	Keyboard Repeat Speed
$2D	Keyboard Repeat Delay
$2E	Double Click Time

$2F	Flash Rate
$30	Shift Caps / Lower Case
$31	Fast Space / Delete Keys
$32	Dual Speed
$33	High Mouse Resolution
$34	Month / Day / Year Format
$35	24 Hour / AM-PM Format
$36	Minimum Ram for RAMDISK
$37	Maximum Ram for RAMDISK
$38-40	Count / Languages
$41-51	Count / Layouts
$52-7F	Reserved
$80	AppleTalk Node Number
$81-A1	Operating system variables
$A2-FB	Reserved
$FC-FF	Checksum

Clock Tools. These routines allow the clock to be set or read. Setting the clock requires that the time be passed as an input paramter in a hex format. Two tools are provided for reading the clocl. One returns time in a hex format, while the other returns time in an ASCII format.

ReadTimeHex Function number = $0D

Input	Word	Space for result	
Input	Word	Space for result	
Input	Word	Space for result	
Input	Word	Space for result	

sp—>

Output	Byte	Day of Week	(0-6 where 0 = Sunday)
Output	Byte	null	
Output	Byte	Month	(0-11 where 0 = January)
Output	Byte	Day	(0-30)
Output	Byte	Current Year minus 1900	
Output	Byte	Hour	(0-23)
Output	Byte	Minute	(0-59)
Output	Byte	Second	(0-59)

sp—>

Returns current time in Hex format.

Example:

```
        PEA     $0000       ; SPACE FOR RESULT
        PEA     $0000       ; SPACE FOR RESULT
        PEA     $0000       ; SPACE FOR RESULT
        PEA     $0000       ; SPACE FOR RESULT
        _READTIMEHEX
```

WriteTimeHex Function number = $0E

Input	Byte	Month	(0-11 where 0 = January)
Input	Byte	Day	(0-30)
Input	Byte	Current Year minus 1900	
Input	Byte	Hour	(0-23)
Input	Byte	Minute	(0-59)
Input	Byte	Second	(0-59)

sp—>

Sets the current time using Hex format.

Example:

```
        PEA     $0104       ; FEBRUARY, 5TH
        PEA     $560A       ; 1986, 10TH HOUR
        PEA     $1900       ; 25TH MINUTE, 0 SEC.
        _WRITETIMEHEX
```

ReadAsciiTime Function number = $0F

Input LongWord ASCII buffer address
sp—>

Reads elapsed time since January 1, 00:00:00 1904, and converts to ASCII time output which is placed in the applications buffer. Note that ASCII time always outputs twenty characters with the MSB of each character set to a one. ASCII time format is defined by the format set up in the battery ram by the control panel. Format versus the battery ram parameter value is shown below:

Date Format	Time Format	ASCII Time Format
0	0	mm/dd/yy HH:MM:SS AM or PM
1	0	dd/mm/yy HH:MM:SS AM or PM
2	0	yy/mm/dd HH:MM:SS AM or PM
0	1	mm/dd/yy HH:MM:SS
1	1	dd/mm/yy HH:MM:SS
2	1	yy/mm/dd HH:MM:SS

Where: HH = Hour
MM = Minute
SS = Second
mm = Month
dd = Day
yy = Year

Example:

```
PUSHLONG    #LABEL          ; BUFFER ADDRESS
_READASCIITIME
```

Vector Initialization Tools. These tools allow the application to set or get the current vector for the interrupt handlers.

SetVector Function number = $10

```
      Input     Word        Vector Reference Number
      Input     LongWord    Address
sp--->
```

Sets the vector address for the interrupt manager or handler specified by the vecotr reference number.

Example:

```
            PEA         $000E       ; REF. = 1/4 SEC. IRQ
            PUSHLONG    #LABEL      ; HANDLER ADDRESS
            _SETVECTOR
```

GetVector Function number = $11

```
      Input     LongWord    Space for result
      Input     Word        Vector Reference Number
sp--->

      Output    LongWord    Address
sp--->
```

Returns with the vector address for the interrupt manager or handler specified by the vector reference number.

Example:

```
            PEA         $0000       ; SPACE FOR RESULT
            PEA         $0000
            PEA         $0015       ; REF. = 1 SEC. IRQ
            _GETVECTOR
```

Vector Reference Numbers:

$0000	Tool Locator #1
$0001	Tool Locator #2
$0002	User's Tool Locator #1
$0003	User's Tool Locator #2
$0004	Interrupt Manager
$0005	COP Manager
$0006	Abort Manager
$0007	System Death Manager
$0008	AppleTalk Interrupt Handler
$0009	Serial Communications Controller Interrupt Handler
$000A	Scan Line Interrupt Handler
$000B	Sound Interrupt Handler
$000C	Vertical Blanking Interrupt Handler
$000D	Mouse Interrupt Handler
$000E	Quarter Second Interrupt Handler
$000F	Keyboard Interrupt Handler
$0010	Front Desk Bus Response Byte Interrupt Handler
$0011	Front Desk Bus SRQ Interrupt Handler
$0012	Desk Accessory Manager
$0013	Flush Buffer Handler
$0014	Keyboard Micro Interrupt Handler
$0015	One Second Interrupt Handler
$0016	External VGC Interrupt Handler
$0017	Other Unspecified Interrupt Handler
$0018	Cursor Update Handler
$0019	Increment Busy Flag (for Scheduler)
$001A	Decrement Busy Flag (for Scheduler)
$001B	Bell Vector (for Sound Tools)
$001C	Break Vector (for Debuggers)
$001D	Trace Vector
$001E	Step Vector
$001F	Reserved Vector
$0020	Reserved Vector
$0021	Reserved Vector
$0022	Reserved Vector
$0023	Reserved Vector
$0024	Reserved Vector
$0025	Reserved Vector
$0026	Reserved Vector
$0027	Reserved Vector
$0028	Control Y Vector
$0029	Reserved Vector
$002A	ProDOS'16 MLI Vector
$002B	OS Vector
$002C	Message Pointer Vector

HeartBeat Tools. These tools allow the application to insert or delete tasks from the HeartBeat queue.

SetHeartBeat Function number = $12

Input LongWord Pointer
sp—>

Installs the task specified by the **pointer** into the HeartBeat queue. The pointer must be set to the address of a task header that precedes the task. The task header area consists of a longword link pointer, count word, and signature word. The link pointer is maintained by the tool, and is set to a value of $00000000 if the task is the last task in the queue. When a task is installed, the link pointer of the previous task is set to point at the task header for the task currently being installed. The count word is set by the application prior to installing the task, and must be maintained by either the task or the application. The count word indicates the number of VBL interrupts that must occur before the associated task is executed. For recurring tasks, the task should reset the count word. For tasks that are run as a software one-shot, the application should reset the count word. The tool will decrement a non zero count word each VBL interrupt. If the decrement results in a count word of zero, the task will be executed. A count word with a value of zero will not be decremented during VBL interrupt, and effectively sets the task inactive until a non zero value is stored to the count word. Tasks are executed in native mode with 8 bit 'm' and 'x'. Task execution should terminate with an 'RTL' instruction. The signature word must be set prior to installing a task, and is used by the tool and the HeartBeat Interrupt Handler to check the integrity of the HeartBeat queue. An example of a HeartBeat task that increments a location in memory everty tenth VBL is shown below:

```
Task1Hdr  Start
          dc      4i'0'        ; Space for Link Pointer
Task1Cnt  dc      i'10'        ; Count word preset to 10
          dc      h'5AA5'      ; Signature Word $A55A
Task1     anop
          rep     #$20         ; 16 bit 'm'
          longa   on
          phk                  ; data bank = program bank
          plb
          lda     #10          ; reset the task count
          sta     Task1Cnt
          sep     #$20         ; 8 bit 'm'
          longa   off
          lda     >TestLoc     ; and increment an address
          inc     a
          sta     >TestLoc
          rtl
```

The following code will install the task shown above.

```
Install   anop
          PUSHLONG  #LABEL     ; BUFFER ADDRESS
          _SETHEARTBEAT        ; INSTALL TASK
```

Note that when a task is installed into the HeartBeat queue, the HeartBeat Interrupt Handler will automatically be installed into the VBL Interrupt Handler vector. Any handler previously installed in the VBL Interrupt Handler vector will be displaced. Installing a task in the HeartBeat queue does not automatically enable VBL interrupts. It is left to the application to enable VBL interrupts. Also, since tasks are linked with simple pointers, the tasks should reside in 'LOCKED' memory. Tasks that make use of system resources should conform to the protocol set down in the SCHEDULER ERS.

It may be desirable to have a ROM based task executing from a peripheral card. In order to install a ROM based task, twelve bytes of ram must be allocated for use by the task header, with the task executing a jump absolute long to the rom based task. An example of this is shown below:

```
Task1Hdr  dc        4i'0'        ; Space for Link Pointer
Task1Cnt  dc        i'10'        ; Count word preset to 10
Task1Sig  dc        h'5AA5'      ; Signature Word $A55A
Task1Jmp  anop
          jmp       >RomTask1    ; jump to ROM based task
```

An example that shows how a program can construct the task header area in RAM for a ROM based task is shown below. Note that this program is run in full native mode (16 bit 'm' and 'x').

```
InstallT1 entry
          lda       #$0001       ; initialize task count
          sta       >Task1Cnt
          lda       #$A55A       ; initialize task signature
          sta       >Task1Sig
          lda       #RomTask1    ; now install 'JMP' to task
          pha
          xba
          and       #$FF00
          ora       #$005C
          sta       >Task1Jmp
          pla
          and       #$FF00
          ora       #^RomTask1
          xba
          sta       >Task1Jmp+2
          PushLong  #Label       ; now install the task
          _SetHeartBeat
```

Errors that may occur when installing a task in the HeartBeat queue include:

$0303	Task already installed in queue
$0304	No signature in task header
$0305	Queue has been damaged-task signature missing during search

DelHeartBeat Function number = $13

Input LongWord Pointer
sp—>

Deletes the task specified by the link address from the HeartBeat Interrupt service queue.

Errors that may occur when deleting a task in the HeartBeat queue include:

$0305 Queue has been damaged-task signature missing during search
$0306 Task was not found in queue

Example:

```
PUSHLONG  #LABEL        ; TASK ADDRESS
_DELHEARTBEAT
```

ClrHeartBeat Function number = $14

Clears the HeartBeat queue root link pointer, affectively removing all tasks from the queue.

Example:

```
_CLRHEARTBEAT
```

System Death Manager. This tool call jumps through the system death vector. At system power-up time, a default system death manager is installed into the system death manager vector. The default system death manager will display either a default system death message followed by an error code, or a user defined system death message followed by an error code. The default system death message will display a sliding Apple below a centered default message as shown below:

FATAL SYSTEM ERROR-> XXXX

If a system death call is made with a user defined message, the user defined message will be displayed starting at the upper left hand corner fo the screen. The user defined message may contain up to 254 characters. The text may be moved down by imbedding carriage return characters in the text. Any desired delimiters between the text string and the error code should be included in the text string.

USER DEFINED MESSAGE OF UP TO 255 CHARACTERS XXXX

SysDeathMgr Function number = $15

	Input	Word	Error code
	Input	LongWord	Pointer
sp—>			

If the longword pointer is set to zero, the default system death message and the error code passed as the tool input are displayed. If pointer is set to point to an ASCII string, the ASCII string will be displayed with the error code. The first byte of the ASCII string should contain a count equal to the number of characters to be displayed. The ASCII string should have the MSB turned off. Note that this tool call will not return! **Death Messages cannot reside in the Language Card address space.**

Example:

```
PEA         $0004       ; YOUR ERROR CODE
PUSHLONG    #LABEL      ; STRING POINTER
_SYSDEATHMGR
```

Reserved System Death Error Codes:

$0001	ProDOS'16 - Unclaimed interrupt
$0004	Divide by zero
$000A	ProDOS'16 - Volume Control Block unusable
$000B	ProDOS'16 - File Control Block unusable
$000C	ProDOS'16 - Block zero allocated illegally
$000D	ProDOS'16 - Interrupt with I/O shadowing off
$0015	Segment Loader error
$0017-24	Can't load a package
$0025	Out of memory
$0026	Segment Loader error
$0027	File map trashed
$0028	Stack overflow error
$0030	Please insert disk (file manager alert)
$0032-53	Memory manager error
$0100	Can't mount system startup volume

System death error codes above $0100 will be tools specific. The high byte of the error code will contain the tool number reporting the error. The low byte of the error code is defined by the tool set reporting the error. No tool will report an error with the low byte set to a value of $00.

DeathCode	Related ToolSet
$01XX	Tool Locator
$02XX	Memory Manager
$03XX	Miscellaneous Tools
$04XX	Quick Draw
$05XX	Desk Manager
$06XX	Event Manager
$07XX	Scheduler
$08XX	Sound Manager
$09XX	Apple Desktop Bus Tools
$0AXX	SANE
$0BXX	Integer Math Tools
$0CXX	Text Tools
$0DXX	Ram Disk
$0EXX	Menu Manager
$0FXX	Window Manager
$10XX	Control Manager
$11XX	Loader
$12XX	Printer 1
$13XX	Printer 2
$14XX	Line Edit
$15XX	Pick Manager
$16XX	Dialog Manager

GET ADDRESS Tools. These tools are provide to allow an application to determine the address of a parameter used by the system firmware.

GetAddr Function number = $16

	Input	LongWord	Space for result
	Input	Word	Reference number
sp—>			
	Output	LongWord	Pointer to parameter
sp—>			

Parameter reference numbers and parameter size are defined below:

Ref. #	Length	Parameter	
$0000	Byte	IRQ Interrupt Flag	(IRQ.INTFLAG)
$0001	Byte	IRQ Data Flag	(IRQ.DATAREG)
$0002	Byte	IRQ Serial Port 1 Flag	(IRQ.SERIAL1)
$0003	Byte	IRQ Serial Port 2 Flag	(IRQ.SERIAL2)
$0004	Byte	IRQ AppleTalk Flag	(IRQ.APLTLKHI)
$0005	LongWord	Tick Counter	(TICKCNT)
$0006	Byte	IRQ Volume	(IRQ.VOLUME)
$0007	Byte	IRQ Active	(IRQ.ACTIVE)
$0008	Byte	IRQ Sound Data	(IRQ.SOUNDDATA)
$0009	20 Bytes	Variables after a 'BRK'	(BRK.VAR)
$000A	12 Bytes	Event Manager Data	(EVMGRDATA)
$000B	Byte	Mouse Location/Flag	(MouseSlot)
$000C	8 Bytes	Mouse Clamps	(MOUSECLAMPS)
$000D	8 Bytes	Absolute device clamps	(ABSCLAMPS)

Note that parameters with reference numbers from $0000 through $0004 should not be used by applications. These parameters are only valid while servicing an interrupt.

Example:

```
        PEA     $0000       ; SPACE FOR RESULT
        PEA     $0000
        PEA     $000C       ; REF. = MOUSE CLAMPS
        _GETADDR
```

Further definition of some parameters is provided below:

IRQ.INTFLAG	D7	1 = Mouse button down
	D6	1 = Mouse button down on last read
	D5	Status of AN3
	D4	1 = 1/4 second interrupted
	D3	1 = VBL interrupted
	D2	1 = Mega// Mouse switch interrupted
	D1	1 = Mega// Mouse movement interrupted
	D0	1 = System IRQ line is asserted
IRQ.DATAREG	D7	1 = Response byte, 0 = Status byte
	D6	1 = Abort
	D5	1 = Desktop manager sequence pressed
	D4	1 = Flush buffer sequence pressed
	D3	1 = SRQ
	D0-2	0 = No FDB data, 0 ≠ number of valid bytes -1
BRK.VAR	Word	A Register
	Word	X Register
	Word	Y Register
	Word	Stack Pointer
	Word	Direct Register
	Byte	Processor Status
	Byte	Data Bank Register
	Byte	Emulation Flag
	Byte	Program Bank Register
	Word	Program Counter
	Byte	State
	Byte	Shadow
	Byte	CYA
	Byte	MSlot
EVMGRDATA	Word	Journaling flag (JournalFlag)
	LongWord	Pointer to journal driver (JournalPtr)
MouseSlot	Byte	Location of the Mouse (MouseSlot) This is a flag used by the MouseTools. If MouseSlot contains a positive value, then it indicates what slot the mouse resides in. If MouseSlot contains a negative value, the Mouse has not been initialized by the Mouse Tools.

MouseClamps	Word	Low X axis mouse clamp
	Word	Low Y axis mouse clamp
	Word	High X axis mouse clamp
	Word	High Y axis mouse clamp

(Note that setting the mouse clamp values directly is not a viable method of setting the mouse clamps. Setting mouse clamps correctly can only be guaranteed using the mouse tools.)

AbsClamps	Word	Low X axis absolute device clamp
	Word	Low Y axis absolute device clamp
	Word	High X axis absolute device clamp
	Word	High Y axis absolute device clamp

(There is no built in firmware to clamp absolute device position within the absolute device clamp bounds. Absolute device drivers must be responsible for clamping position within the clamp bounds.)

Mouse Tools. These tools are provide to interface with the Mouse. These tools will work with both the built in Front Desk Bus Mouse or the Apple][Mouse. Note that the 'InitMouse' call must be executed first. An error will be returned if a dispatch to the mouse is executed with the mouse firmware switched out.

ReadMouse Function number = $17

	Input	Word	Space for result
	Input	Word	Space for result
	Input	Word	Space for result
sp—>			
	Output	Byte	High Byte X Position
	Output	Byte	Low Byte X Position
	Output	Byte	High Byte Y Position
	Output	Byte	Low Byte Y Position
	Output	Byte	Mouse Status
	Output	Byte	Mouse Mode
sp—>			

Returns Mouse position, status and mode.

Example:

```
PEA         $0000       ; SPACE FOR RESULT
PEA         $0000
PEA         $0000
_READMOUSE
```

InitMouse Function number = $18

	Input	Word	Mouse slot $0000 = Search slots for Mouse $0001-7 = Slot Mouse resides in
sp—>			

Initializes the mouse clamp values to $0000 minimum and $03FF maximum. Mouse mode and status are cleared.

Example:

```
PEA         $0000       ; REQUEST SEARCH
_INITMOUSE
```

SetMouse Function number = $19

```
    Input       Word        Mode (in low byte)
sp—>
```

Mode is set to new value as follows:

$00	Turn off Mouse
$01	Set transparent mode
$03	Set movement interrupt mode
$05	Set button interrupt mode
$07	Set button or movement interrupt mode
$08	Turn mouse off, VBL IRQ active
$09	Set transparent mode, VBL IRQ active
$0B	Set movement interrupt mode, VBL IRQ active
$0D	Set button interrupt mode, VBL IRQ active
$0F	Set button or movement interrupt mode, VBL IRQ active

Example:

```
        PEA         $0001       ; TRANSPARENT MODE
        _SETMOUSE
```

HomeMouse Function number = $1A

Positions the Mouse at the minimum clamp position.

Example:

```
        _HOMEMOUSE
```

ClearMouse Function number = $1B

Sets both the X and Y axis position to $0000 if minimum clamps are negative (delta or relative mode), or to the minimum clamp position if the clamps are positive (absolute mode).

Example:

```
        _CLEARMOUSE
```

ClampMouse Function number = $1C

Input	Word	X axis minimum clamp value
Input	Word	X axis maximum clamp value
Input	Word	Y axis minimum clamp value
Input	Word	Y axis maximum clamp value

sp—>

Sets the clamp values to new values, and then sets the Mouse position to the miminum clamp values.

Example:

```
PEA         $0000       ; X MINIMUM
PEA         $03FF       ; X MAXIMUM
PEA         $0000       ; Y MINIMUM
PEA         $03FF       ; Y MAXIMUM
_CLAMPMOUSE
```

GetMouseClamp Function number = $1D

Input	Word	Space for result
Input	Word	Space for result
Input	Word	Space for result
Input	Word	Space for result

sp—>

Output	Word	X axis minimum clamp value
Output	Word	X axis maximum clamp value
Output	Word	Y axis minimum clamp value
Output	Word	Y axis maximum clamp value

sp—>

Returns the current values of the Mouse clamps.

Example:

```
PEA         $0000       ; SPACE FOR RESULT
PEA         $0000
PEA         $0000
PEA         $0000
_GETMOUSECLAMP
```

PosMouse Function number = $1E

```
      Input     Word        X axis position
      Input     Word        Y axis position
sp—>
```

Positions the Mouse to the coordinates specified.

Example:

```
            PEA         $013C       ; X POSITION
            PEA         $028F       ; Y POSITION
            _POSMOUSE
```

ServeMouse Function number = $1F

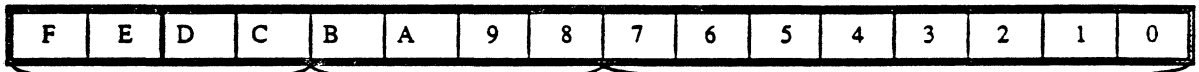
```
      Input     Word        Space for result
sp—>

      Output    Word        Interrupt status (in low byte)
sp—>
```

Returns mouse interrupt status.

Example:

```
            PEA         $0000       ; SPACE FOR RESULT
            _SERVEMOUSE
```

ID Tag Manager. These tools are used to create, delete and inquire status of an ID Tag. The ID Tag is used to mark memory segments as belonging to a specific application or desk accessory. ID tags are made up of three fields encoded in a word parameter. These are the TYPE field, AUX ID field, and MAIN ID field. The type field is encoded in bits 12-14, Aux ID in bits 8-11, and the Main ID in bits 0-7. The AUX ID field is defined by the caller. The Main ID field is generated by the ID Tag manager. The ID Tag will always be assigned with a non zero value in the Main ID field. The Type field has fixed assignments as shown in the table below:



TYPE FIELD.....
0 = MEMORY MANAGER
1 = APPLICATION
2 = CONTROL PROGRAM
3 = ProDOS
4 = TOOL KITS
5 = DESK ACCESSORIES
6 = RUN TIME LIBRARIES
7 = SYSTEM LOADER
8 = FIRMWARE / SYSTEM FUNCTION
9 = TOOL LOCATOR
A = SETUP FILE
B = UNDEFINED
C = UNDEFINED
D = UNDEFINED
E = UNDEFINED
F = UNDEFINED

AUX ID FIELD.....
0 - $F Defined by User

(ID $41XX = Miscellaneous Tools)
($42XX = Scrap Manager)

MAIN ID FIELD.....
0 = Reserved
1-FF Assigned by ID Manager

GetNewID Function number = $20

	Input	Word	Space for result
	Input	Word	ID Tag
sp—>			
	Output	Word	ID Tag
sp—>			

Caller passes a full 16 bit ID tag as input with the TYPE defined as the only relevant parameter. The AUX ID field is specified by the caller, and will not be reassigned by the ID manager. The next available MAIN ID will be concatenated to the TYPE and AUX ID fields, and the resulting ID Tag will be returned to the caller. Note that the TYPE field must be non zero. Note that only 255 ID tags can be assigned for any TYPE ID. If an ID cannot be assigned because all the ID tags for that TYPE have been assigned, then an error will be returned indicating that the ID is not available.

Example:

```
        PEA     $0000       ; SPACE FOR RESULT
        PEA     $5100       ; ITS A DESK ACC.
        _GETNEWID
```

DeleteID Function number = $21

Input Word ID Tag
sp—>

The caller passes the tool a full 16 bit ID tag as input with the TYPE and MAIN ID fields defined as the only relevant parameters. Any ID tags with the same MAIN ID and TYPE are deleted from the current ID tag list. This tool call will not report an error if the tag is not found. It assumes that if its not there, that is what you wanted anyway.

Example:

```
PEA     $5101     ; DELETE DESK ACC TAG
_DELETEID
```

StatusID Function number = $22

Input Word ID Tag
sp—>

The caller passes the tool a full 16 bit ID tag as input with the TYPE and MAIN ID fields defined as the only relevant parameters. If the ID tag is active, no error will be returned. If the ID tag is inactive, an error will be returned indicating that the ID tag is not available.

Example:

```
PEA     $5101     ; DELETE DESK ACC TAG
_STATUSID
```

Interrupt Control Tools. This tool allows certain interrupt sources to be enabled or disabled.

IntSource Function number = $23

	Input	Word	Source Reference Number
sp—>			

This tool call enables or disables the interrupt source specified by the source reference number. Source reference numbers are shown below:

Ref. #	Source and Action
$0000	Enable Keyboard Interrupts
$0001	Disable Keyboard Interrupts
$0002	Enable Vertical Blanking Interrupts
$0003	Disable Vertical Blanking Interrupts
$0004	Enable Quarter Second Interrupts
$0005	Disable Quarter Second Interrupts
$0006	Enable One Second Interrupts
$0007	Disable One Second Interrupts
$0008	THIS DOES NOTHING
$0009	THIS DOES NOTHING
$000A	Enable FDB Data Interrupts
$000B	Disable FDB Data Interrupts
$000C	Enable Scan Line Interrupts
$000D	Disable Scan Line Interrupts
$000E	Enable External VGC Interrupts
$000F	Disable External VGC Interrupts

Example - Installing and enabling a one second interrupt handler:

```
PEA             $0015           ; SET ONE SEC VECTOR
PUSHLONG        #ONEHANDLER     ; POINTER TO HANDLER
_SETVECTOR
BCS             ERROR           ; IF TOOL ERROR OCCURED
PEA             $0006           ; ENABLE ONE SEC IRQ
_INTSOURCE
BCS             ERROR           ; IF TOOL ERROR OCCURED
```

ABOUT KEYBOARD INTERRUPTS.....

When keyboard interrupts are enabled, there is no hardware enable of the keyboard interrupt. The firmware installs a task into the HeartBeat queue and enables VBL interrupts. This causes the HeartBeat interrupt handler to be installed into the VBL interrupt vector. This task will check the status of the keyboard register during each VBL interrupt. If a key is pending, the task will dispatch to the KeyBoard interrupt handler via the keyboard interrupt vector (as installed by the tool 'SETVECTOR'). Since the HeartBeat handler will be installed into the VBL interrupt vector, this precludes the application from installing it's own VBL interrupt handler if keyboard interrupts are to be used. If keyboard interrupts are disabled, the keyboard task is removed from the HeartBeat queue, however the VBL interrupt will not be disabled. If the application wishes to disable keyboard interrupts, and does not wish to have the additional overhead of the VBL interrupts running in the background, the application must disable VBL interrupts also. If no other tasks have been installed into the HeartBeat queue, the additional interrupt overhead is minimal (Interrupt dispatcher and HeartBeat interrupt handler which only increments the tick count before returning).

ABOUT ONE SECOND INTERRUPTS.....

The vector initialization tools are used to install a one second interrupt handler into the one second interrupt vector. Then the interrupt source tool call is used to enable the one second interrupt. The one second interrupt handler will be called by the interrupt manager in 8 bit native mode ('m' = 'x' = 1). The one second interrupt handler must clear the hardware source of the interrupt before executing an 'RTL' to the interrupt manager. Note that an interrupt handler must return to the interrupt manager with the carry cleared if the interrupt source was serviced. An example of a one second interrupt handler that increments a memory location is shown below:

```
ONEHANDLER      START
                LONGA       OFF
                LONGI       OFF
                PHB                         ;SAVE ENVIRONMENT
                PHA
                PHK                         ;SET DATA BANK TO PROGRAM
                PLB
                INC         LOCATION
                LDA         #%00000010      ;CLEAR 1 SEC IRQ SOURCE
                TSB         SC032
                PLA                         ;RESTORE ENVIRONMENT
                PLB
                CLC                         ;INDICATE IRQ WAS SERVICED
                RTL
                END
```

ABOUT ONE QUARTER SECOND INTERRUPTS.....

The vector initialization tools are used to install a one quarter second interrupt handler into the one quarter second interrupt vector. Then the interrupt source tool call is used to enable the one quarter second interrupt. The one quarter second interrupt handler will be called by the interrupt manager in 8 bit native mode ('m' = 'x' = 1). The one quarter second interrupt handler must clear the hardware source of the interrupt before executing an 'RTL' to the interrupt manager. Note that an interrupt handler must return to the interrupt manager with the carry cleared if the interrupt source was serviced. An example of a one second interrupt handler that increments a memory location is shown below:

```
QTRHANDLER      START
                LONGA       OFF
                LONGI       OFF
                PHB                         ; SAVE ENVIRONMENT
                PHA
                PHK                         ; SET DATA BANK TO PROGRAM
                PLB
                INC         LOCATION
                STA         SC047
                PLA                         ; RESTORE ENVIRONMENT
                PLB
                CLC                         ; INDICATE IRQ WAS SERVICED
                RTL
                END
```

Firmware Entry Tools. This tool allows the Apple][emulation mode entry points to be supported from full native mode. This tool will preserve the state of the data bank and direct page registers prior to dispatching to the firmware entry point. During the execution of the firmware task, the data bank and direct page registers are set to a value of zero. The data bank and direct page registers are restored on return from the firmware entry point.

FWentry Function number = $24

Input	Word	Space for result	
Input	Word	Space for result	
Input	Word	Space for result	
Input	Word	Space for result	
Input	Word	Accumulator at entry	(low byte only)
Input	Word	X Register at entry	(low byte only)
Input	Word	Y Register at entry	(low byte only)
Input	Word	Emulation mode entry point (16 bits)	

sp—>

Output	Word	Processor status at exit	(low byte only)
Output	Word	Accumulator at exit	(low byte only)
Output	Word	X register at exit	(low byte only)
Output	Word	Y register at exit	(low byte only)

sp—>

This call dispatches to the specified emulation mode entry point with the registers set to the values passed to the tool as input. On return, the register contents resulting from the entry point dispatch will be passed on the stack. Note that only the least significant byte is relevant on the register input and output.

Example:

```
        PEA     $0000       ; SPACE FOR RESULT
        PEA     $0000       ; SPACE FOR RESULT
        PEA     $0000       ; SPACE FOR RESULT
        PEA     $0000       ; SPACE FOR RESULT
        PEA     $0000       ; A REG
        PEA     $0000       ; X REG
        PEA     $0000       ; Y REG
        PEA     $FDDA       ; ENTRY POINT
        _FWENTRY
        BCS     FWERR       ; BRANCH IF ERROR
        PLY                 ; GET FW REGISTERS
        PLX
        PLA
        PLP
        PLP
```

Tick Count Tool. This tool allows caller to read the current value of the tick counter.

GetTick Function number = $25

	Input	LongWord	Space for result
sp—>			
	Output	LongWord	Current value of Tick Counter
sp—>			

Note that the tick count is only incremented by the heartbeat interrupt handler. This means that the heartbeat interrupt handler must be installed, and VBL interrupts must be enabled in order to get an incrementing tick count. Please see the section on heartbeat tasks.

Example:

```
        PEA     $0000       ; SPACE FOR RESULT
        PEA     $0000       ; SPACE FOR RESULT
        _GETTICK
```

PackBytes and UnPackBytes Tools. PackBytes and UnPackBytes provide for the packing and unpacking of any data, but is ususally used for graphic images.

PackBytes Function number = $26

	Input	Word	Space for result
	Input	LongWord	Pointer to pointer to start of area to be packed
	Input	LongWord	Pointer to a word containing size of the area
	Input	LongWord	Pointer to start of the ouput buffer area
	Input	Word	Size of the output buffer area
sp—>			
	Output	Word	Number of packed bytes generated
sp—>			

Upon completion of the call, the pointer to the area to be packed is moved forward to the next packable byte, and the size of area pointed to by the second input parameter is reduced by the number of bytes traversed. An assembly language example follows:

```
*
* PACKBYTES example: Pack a screen image and write it to file "f"
*
PB      START
        lda     #$7D00          ;size of area to pack
        sta     PicSize
        lda     #$E12000        ;addr of screen image
        sta     PicPtr
        lda     #$^E12000
        sta     PicPtr+2
        lda     #buffer         ;pointer to loca buffer
        sta     BufPtr
        lda     #^Buffer
        sta     BufPtr+2

loop    PUSHWORD  #0            ;Space for result
        PUSHLONG  #PicPtr       ;Pointer to data to pack
        PUSHLONG  #PicSize      ;Pointer to word with size of area
        PUSHLONG  BufPtr        ;Pointer to start of output area
        PUSHWORD  BufSize       ;size of output buffer area
        _PACKBYTES
        pla                     ;get howmuch we did pack this pass
        sta     HowMuch

        CALL    WRITE(f,BufPtr,HowMuch) ;do I/O to write "HowMuch" bytes from "BufPtr" to file "f"

        lda     PicSize         ;see if more to pack;
        bne     loop            ;there is, go back for more
        rts

PicPtr  ds      4               ;set to $e12000 on entry (screen area)
PicSize ds      2               ;size of a picture: set to $7d00 on entry
BufPtr  ds      4               ;set to point to "Buffer" on entry
BufSize dc      i2'$400'        ;local buffer for storing packed stuff
HowMuch ds      2               ;local storage for value from packbytes
Buffer  ds      $400            ;actual buffer

        END
```

An equivalent example in PASCAL follows:

```
                    .
                    .
                    .
Function packbytes  ( VAR   picptr    : POINTER;
                      VAR   picsize   : POINTER;
                      bufptr  : POINTER;
                      bufsize : POINTER;
   : INTEGER; EXTERNAL;
                    .
                    .
                    .
picsize := $7D00;
bufsize := $400;    {note: if large enough, could require but one call}
REPEAT
   howmuch := PackBytes (picptr,picsize,bufptr,bufsize);
    write (f,bufptr,howmuch);
UNTIL picsize=0
```

UnPackBytes Function number = $27

	Input	Word	Space for result
	Input	LongWord	Pointer to the buffer containing packed data
	Input	Word	Buffer size
	Input	LongWord	Pointer to pointer to area to unpack data into
	Input	LongWord	Pointer to word containing the size of the area
sp—>			to contain the unpacked data
	Output	Word	Number of bytes unpacked
sp—>			

Upon completion, the pointer to the unpacked data is positioned one past the last upacked byte and the size fo the area is reduced by the amount unpacked. An assembly language example follows:

```
*
* UNPACKBYTES example: UnPack a file "f" onto the screen
*
PB      START

        stz     Mark            ; mark is the file mark we position to
        lda     #$7D00          ;size of area to unpack into
        sta     PicSize
        lda     #$E12000        ;addr of screen image
        sta     PicPtr
        lda     #$^E12000
        sta     PicPtr+2

loop    CALL    SETFILEMARK(f,Mark)     ;position file "f" to position "Mark"

        CALL    READ(f,BufPtr,BufSize)  ;Read BufSize" bytes" into "BufPtr"

        PUSHWORD    #0              ;Space for result
        PUSHLONG    BufPtr          ;Pointer to start of output area
        PUSHWORD    bufsize         ;size of output buffer area
        PUSHLONG    #PicPtr         ;Pointer to data to pack
        PUSHLONG    #PicSize        ;Pointer to word with size of area

        _UNPACKBYTES

        pla                         ;get how much we did unpack this pass
        clc                         ;add to previous mark pos.
        adc     Mark
        sta     Mark
        lda     picsize             ;see if more to unpack;
        beq     done                ;there isn't, so we're done

        CALL    EOF(f)              ;did we get to end of file (safety check)
        bne     loop                ;no, go back for more
Done    rts

BufPtr  dc      i4'Buffer'          ;pointer to buffer area
bufsize dc      i2'$400'            ;local buffer for storing packed stuff
PicPtr  ds      4                   ;set to $e12000 on entry (screen area)
PicSize ds      2                   ;size of a picture: set to $7d00 on entry
Mark    ds      2                   ;file mark position
Buffer  ds      $400                ;actual buffer

        END
```

An equivalent example in PASCAL follows:

```
                    .
                    .
                    .
Function unpackbytes (          bufptr      : POINTER;
                                bufsize     : POINTER;
                        VAR     picptr      : POINTER;
                        VAR     picsize     : POINTER;
   : INTEGER; EXTERNAL;
                    .
                    .
                    .
mark := 0;          [i.e. start of file}
picsize := $7D00
bufsize := $400;    {note: if large enough, could require but one call}
REPEAT
   setfilemark(mark);
   read(f,bufptr,bufsize);
   howmuch := UnPackBytes (bufptr,bufsize,picptr,picsize);
    mark := mark+howmuch;
UNTIL ((picsize=0) or eof(f) );      [eof test in case of bad data}
```

The packed data is in the form of 1 byte containing a flag in the first 2 bits and a count in the remaining 6 bits, followed by one or more data bytes depending on the flags. Their description is as follows:

00XXXXXX : (XXXXXX : 0 -> 63)	= 1 to 64 bytes follow - unique
01XXXXXX : (XXXXXX : 2,4,5 or 6)	= 3,5,6 or 7 repeats of next byte
10XXXXXX : (XXXXXX : 0 -> 63)	= 1 to 64 repeats of next 4 bytes
11XXXXXX : (XXXXXX : 0 -> 63)	= 1 to 64 repeates of next 1 byte taken as 4 bytes (as in '10' case)

Munger. Munger lets you manipulate bytes in a string of bytes. The basic operation is that of searching a destination string for a target string and if found, replacing it with a replacement string. The end of the destination string, if the string is shortened, is padded with a pad character. If the string is elongated, Characters are truncated off of the end. Special cases to allow various other functions are defined below.

Munger Function number = $28

	Input	Word	Space for result
	Input	LongWord	Pointer (destptr)
	Input	LongWord	Pointer (destlen)
	Input	LongWord	Pointer (targptr)
	Input	Word	Integer (targlen)
	Input	LongWord	Pointer (replptr)
	Input	Word	Integer (repllen)
	Input	LongWord	Pointer (pad)
sp—>			
	Output	Word	Amount of Pad / Truncations
sp—>			

Where input is:

destptr:	Pointer to pointer to the text to be manipulated
destlen:	Pointer to number of bytes to manipulate
targptr:	Pointer to string to be searched for from destptr
targlen:	Number of bytes for targptr
replptr:	Pointer to string to replace when targptr found
repllen:	Number of bytes for replptr
pad:	Character value to pad shortened input with

And output is:

destptr:	Updated to one past end of any replacement
destlen:	Old value reduced by bytes scanned across
pad:	Number of bytes padded (or truncated)
Munger:	Zero if target found, negative if not

Special cases:

If targptr is 0, the substring of length targlen is replaced by the replptr string.

If targlen is 0, replptrs string is inserted at destptr.

If replptr is 0, destptr is updated to past the end of the match of the targptr string.

If repllen is 0, (and replptr is not) the targptr string is deleted rather than replaced (since the replacement string is empty).

There is one case in which munger performs a replacement even if it doesn't find all for the target string. If the destptr string in ints entirety is at the beginning of the targptr string, then the destptr string is totally replaced by the replptr string.

```
*
* MUNGER example : editing a line of text
*
* Changes "robert irwin eagle toranaga marcia houdini berns"
*    into "robert irwin EAGLE toranaga marcia houdini berns"
*
MG      START

        lda         #Name           ;set pointer to name
        sta         DestPtr
        lda         #^Name
        sta         DestPtr+2

        lda         #48             ;get length
        sta         DestLen

        PUSHWORD    #0              ;space for result
        PUSHLONG    DestPtr         ;pointer to textstring to manipulate
        PUSHLONG    #DestLen        ;Pointer to word with number bytes to change
        PUSHLONG    #eagleLC        ;Points to "eagle" (lower case)
        PUSHWORD    #5              ;"eagle" has 5 letters
        PUSHLONG    #eagleUC        ;Pointer to "EAGLE" (upper case)
        PUSHWORD    #5              ;"EAGLE" has 5 letters
        PUSHLONG    #PAD            ;Pad char (don't care for this example)

        _MUNGER

        pla                         ;[THIS WILL BE ZERO, AS WILL PAD]

        rts

DestPtr ds          2               ;on entry, will point to name
DestLen ds          2               ;on entry will be set to "NLen"

PAD     ds          2               ;pad value

eagleLC dc          c'eagle'
eagleUC dc          c'EAGLE'

name    dc          c'robert irwin eagle toranaga marcia houdini berns'
```

An equivalent example in Pascal follows:

```
                .
                .
                .
Function munger     ( VAR    destptr     : POINTER;
                      VAR    destlen     : INTEGER;
                             targptr     : POINTER;
                             targlen     : INTEGER;
                             replptr     : POINTER;
                             repllen     : INTEGER;
                      VAR    PAD         : INTEGER;
  : INTEGER; EXTERNAL;
                .
                .
                .
```

{segment to replace a word in lower case with it's upper case equivalent}

```
name := 'robert irwin eagle toranoga marcia houdini berns';
I:= LEN (name);
i := munger(name, I,'eagle',5,'EAGLE',5,p);
```

{upon completion, *i* is 0, *p* is 0, and *name* is 'robert irwin EAGLE toranaga marcia houdini berns'}

Interrupt Enable State Tool. This function returns with the state of hardware interrupt enable states for interrupt sources that can be controlled by the miscellaneous tool set.

GetIRQenbl Function number = $29

sp—>	Input	Word	Space for result
sp—>	Output	Word	Status of hardware interrupt enables

Status in returned word is defined below:

D8-15	Undefined
D7	1 = Keyboard interrupts enabled
D6	1 = Vertical blanking interrupts enabled
D5	1 = Quarter second interrupts enabled
D4	1 = One second interrupts enabled
D3	Reserved
D2	1 = Front Desk Bus data interrupts enabled
D1	1 = Scan line interrupts enabled
D0	1 = External VGC interrupts enabled

Example:

```
            PEA         $0000          ; SPACE FOR RESULT
            _GETIRQENBL
```

SetAbsClamp Function number = $2A

```
        Input       Word        X axis minimum clamp value
        Input       Word        X axis maximum clamp value
        Input       Word        Y axis minimum clamp value
        Input       Word        Y axis maximum clamp value
sp—>
```

Sets the clamp values for absolute devices to new values.

Example:

```
            PEA         $0000       ; X MINIMUM CLAMP
            PEA         $03FF       ; X MAXIMUM CLAMP
            PEA         $0000       ; Y MINIMUM CLAMP
            PEA         $03FF       ; Y MAXIMUM CLAMP
            _SETABSCLAMP
```

GetAbsClamp Function number = $2B

```
        Input       Word        Space for result
        Input       Word        Space for result
        Input       Word        Space for result
        Input       Word        Space for result
sp—>

        Output      Word        X axis minimum clamp value
        Output      Word        X axis maximum clamp value
        Output      Word        Y axis minimum clamp value
        Output      Word        Y axis maximum clamp value
sp—>
```

Returns the current values of the absolute device clamps.

Example:

```
            PEA         $0000       ; SPACE FOR RESULT
            PEA         $0000
            PEA         $0000
            PEA         $0000
            _GETABSCLAMP
```

System Beep Call. This function calls the Apple][monitor entry point 'BELL1'. The bell routine can be patched out using the SetVector function to patch out the bell vector. Note that any bell routine installed into the bell vector will be called in native mode with 8 bit 'm' and 'x', and must return with the carry flag cleared via an RTL instruction.

SysBeep Function number = $2C

sp—>

Calls the system bell routine.

Example:

```
_SYSBEEP
```

Miscellaneous Tool Set Error Codes

$0000	No Error
$0301	Bad Input Parameter
$0302	No Device for Input Parameter
$0303	Task is already in Heartbeat queue
$0304	No signature in task header was detected during insert or delete
$0305	Damaged queue was detected during insert or delete
$0306	Task was not found during delete
$0307	Firmware task was unsuccessful
$0308	Detected damaged HeartBeat Queue
$0309	Attempted dispatch to a device that is not connected
$030A	Undefined
$030B	ID tag not available

Summary of functions within the Miscellaneous Tool Set

Function Number		Description
$01	1	MTBootInit
$02	2	MTStartUp
$03	3	MTShutDown
$04	4	MTVersion
$05	5	MTReset
$06	6	MTStatus
$07	7	MTSpare1
$08	8	MTSpare2
$09	9	WriteBRam
$0A	10	ReadBRam
$0B	11	WriteBParam
$0C	12	ReadBParam
$0D	13	ReadTimeHex
$0E	14	WriteTimeHex
$0F	15	ReadAsciiTime
$10	16	SetVector
$11	17	GetVector
$12	18	SetHeartBeat
$13	19	DelHeartBeat
$14	20	ClrHeartBeat
$15	21	SysDeathMgr
$16	22	GetAddr
$17	23	ReadMouse
$18	24	InitMouse
$19	25	SetMouse
$1A	26	HomeMouse
$1B	27	ClearMouse
$1C	28	ClampMouse
$1D	29	GetMouseClamp
$1E	30	PosMouse
$1F	31	ServeMouse
$20	32	GetNewID
$21	33	DeleteID
$22	34	StatusID
$23	35	IntSource
$24	36	FWentry
$25	37	GetTick
$26	38	PackBytes
$27	39	UnPackBytes
$28	40	Munger
$29	41	GetIRQenbl
$2A	42	SetAbsClamp
$2B	43	GetAbsClamp
$2C	44	SysBeep

Documentation Développeurs
Apple Computer France 1987

Document développeur numéro 50

Desk Manager

type d'upgrade de ce ducument : 5
1 Documentation de première catégorie inchangée
2 Documentation de deuxième catégorie mise à jour
3 Documentation de deuxième catégorie inchangée
4 Mise à jour payante de la documentation de première catégorie
5 Mise à jour gratuite de la documentation de première catégorie
6 Nouveautés payantes non vitales
7 Nouveautés gratuites et vitales

Taille : 15 page(s) environ

Domaine : Tool 05

VERSION : ERS
DATE : 24.08.86

Desk Manager
External Reference Specification

Steven Glass
John Worthington

August 24, 1986

November 26, 1985	Initial Release.
January 9, 1986	Major Changes to the Data Structures
June 18, 1986	Additional changes reflecting code actually in ROM
July 11, 1986	Changes reflect code implemented for NDAs
August 24, 1986	Structure of CDAs changed slightly. ID section now contains a pointer to a shutdown routine. The INIT routine of an NDA is passed a variable indicating whether this is a startup or shutdown call. The description of the NDA ID routine incorporates the last round of changes to the Menu Manager. SetDAStrPtr & GetDAStrPtr are now correctly documented. Rules for supporting NDAs more solid. Requirements for making DeskStartup and DeskShutdown calls are given. SystemMenu Call no longer supported. Description of FixAppleMenu corrected.

Summary

The Desk Manager provides the user access to desk accessories. A desk accessory is a "mini-application" that can be run at the same time as a Cortland application.

The Desk Manager provides support for two types of desk accessory's: Classic Desk Accessories (CDA) and New Desk Accessories (NDA).

Classic Desk Accessories are desk accessories that are designed to execute in a non-desktop, non-event based environment. Unlike NDAs, a classic accessory gets full control of the machine during what is basically an interrupt state (generated by a keypress). The desk accessory is responsible for saving any of the application's memory that it uses as well as handling all I/O.

New Desk Accessories are Macintosh Style desk accessories that are designed to execute in a desktop, event based environment. NDAs run in a window and get "control" when that window is the topmost window. Just what kind of control a NDA has is described below.

How CDA's are Used

A user activates a CDA from the CDA menu. The CDA menu is displayed by pressing OPEN APPLE-CONTROL-ESCAPE. Two CDA's are built into the system:

Control Panel
Alternate Display Mode

Any others (up to eleven) are loaded from disk. From the CDA menu, a user can select any of the DA's currently in the system. The desk accessory is activated and retains control until it shuts down. When it shuts down, the Desk Manager re-displays the CDA Menu. Only when the user selects Quit from the CDA menu does the original application resume operation.

When can the CDA Menu be displayed?

The Desk manager gets control whenever the user presses OPEN-APPLE CONTROL-ESC. Before it displays the CDA Menu, it checks the system busy flag. If something in the system is busy, the Desk Manager schedules a wake-up with the scheduler. The next time the system flag is free, the scheduler will wake up the Desk Manager which then can display the CDA menu. This guarentees that CDA's have all system resources available to them when they are called.

Execution Environment

Classic Desk Accessories have a single entry (activation) point. When the CDA gets control, the processor is in full native mode (16-bit m and x registers). The desk accessory menu is still displayed on the screen in whatever was mode requested by the user (in the control panel). The CDA must execute the RTL in full native mode for the Desk Manager to work correctly.

When the desk manager displays the CDA menu, it saves the text pages in bank 0 and 1, $E0 and $E1 along with pages 0 and 1 of bank 0 (system direct page and stack). (Only the screen holes used by the Desk Manager are preserved.) These parts of memory are restored by the Desk Manager when the user selects Quit from the CDA menu. Thus a

CDA can feel free to use almost all of this memory as it sees fit. The exception is the stack. Since, the Desk Manager's return address is on the stack (along with other Desk Manager variables), the CDA cannot cut the stack back any farther than it is when it gets control.

A CDA must take care using any other memory in the system that it does not already own. A CDA can use the Memory Manager to obtain additional memory outside "special memory", but it cannot rely on being able to obtain any more of bank 0 and 1.

Form In Memory and On Disk

Classic Desk Accessories have a simple form. They are load files kept on the system disk in the DESK.ACCS subdirectory of the SYSTEM directory and have a file type $B9. The CDA starts with an identification section as follows.

```
StartOfDA   dc i1'NameLength'        ; this combined with the characters that
            dc c'Name of DA'         ; follow make a ProDOS string
            dc i4'StartOfDACode'     ; This is a pointer to the start of the code
            dc i4'ShutDownRoutine'   ; This is a pointer to a shutdown routine
```

The identification structure contains the name of the desk accessory and two pointers. The first pointer is the address of the main entry point to the CDA. The second pointer is the address of a termination routine that is called whenever the DeskShutdown call is made. (Usually by a ProDOS 16 application before it quits but also by ProDOS 16 itself whenever the OS is switched from 16 to 8 or 8 to 16.

How NDAs are Used

New Desk Accessories are loaded by the operating system at boot time. An application that wants to make NDA's available to the user does not have to do a lot of work. If the Application uses TaskMaster, it need only make three calls:

DeskStartup	to initialize the Desk Manager
FixAppleMenu	to put the list of NDAs in the Apple Menu
DeskShutdown	to shut down the Desk Manager

TaskMaster will handle opening NDAs in response to menu selections, calling SystemTask and SystemClick when appropriate. Calling SystemEdit when a selection is made from the Edit Menu, and closing a desk accessory in response to the Close item of the File Menu.

Applications that do not use TaskMaster must do the following to support new desk accessories.

call DeskStartup	To initialize the Desk Manager.
call FixAppleMenu	To put the list of NDAs in the Apple Menu
call OpenNDA	When the user selects an NDA from the Apple Menu
call SystemTask	Frequently (at least every time through the event loop).
call SystemClick	When a MouseDown event occurs in a system window.
call SystemEdit	When a desk accessory is active and the user selects undo, cut, copy, paste or clear from the edit menu.
close an NDA	When the user selects close from the file menu. You can use CloseNDA or CloseNDAbyWinPtr to do this.
DeskShutdown	To shut down the Desk Manager

Execution Environment

NDAs have four entry points: open, close, action and init. For each of these entry points the processor is in Full Native Mode. There is no direct page available so the NDA must obtain it from the stack. The open routine returns a long word on the stack. The action routine is passed information in all three registers.

A NDA can assume that the following tools are loaded and initialized:

QuickDraw
Event Manager
Window Manager
Menu Manager
Control Manager
Scrap Manager
LineEdit
Dialog Manager

An NDA can also assume that the PrintManager is available but not necessarily loaded.

The NDA is responsible for saving and restoring important globals like the current graf port (other important globals will be added to this list as we think of them).

Form In Memory and On Disk

New Desk Accessories have a different form than CDAs. They are still load files kept on the system disk in the DESK.ACCS subdirectory of the SYSTEM directory but they have a file type $B8 and The NDA starts with an identification section as follows.

```
StartOfDA   dc i4'PtrToOpen'        ; Pointer to the open routine
            dc i4'PtrToClose'       ; Pointer to the close routine
            dc i4'PtrToAction'      ; Pointer to the action routine
            dc i4'PtrToInit'        ; Pointer to the init routine
            dc i2'Period'           ; How often the NDA gets run codes
            dc i2'EventMask'        ; Describes what events it wants
            dc c' LMenuLine \H**'   ; The text which describes the menu item
```

The open routine must return a pointer to its window on the stack. When it calls the open routine, the desk accessory manager puts 4 bytes of zero on the stack before it pushes the RTL address.

The close routine has no inputs and no outputs. It should however be able to work even if it is called when the desk accessory is not open.

The action routine is passed the action code in the a register. The possible action codes are:

Event	1	The pointer to the event is pased in the x & y registers. The only events that can be passed to a DA are ButtonDown, ButtonUp, KeyDown, AutoKeyDown, Update and Activate. Update and Activate events for a desk accessory are always passed on. The first three

		are passed on only if the EventMask indicates they should be passed on.
Run	2	The time period specified has passed.
Cursor	3	This is passed to a desk accessory if it is the front window each time SystemTask is called. The purpose is to allow the desk accessory to change the cursor when it is over the NDA's window.
Menu	4	This is passed to a desk accessory if an item from a system menu is selected. The MenuId is passed in the X register, the MenuItemId is passed in the Y register.
Undo	5	This is passed to a desk accessory if the application
Cut	6	determines that the user has selected one of these edit
Copy	7	commands from Edit menu. The action call should
Paste	8	return a boolean in the A-register indicating whether
Clear	9	or not the the command was handled.

The InitRoutine is a routine that is called every time DeskStartup or DeskShutdown is called. The desk manager passes a variable in the a-register to indicate which call is being made (startup or shutdown). A zero indicates that this is a shutdown call; a nonzero number indicates this is a startup call.

The Period field describes how often the DA should be called with the "Run" action code. A period of 1 is every 60th of a second. A period of 2 is every 30th of a second. A period of 60 is every second. A period of $FFFF is never. A period of 0 is as often as possible. The action routine is called with the "run" code from SystemTask. The application should be calling SystemTask every time through its event loop.

The MenuLine is a line of text that will be passed to the Menu Manager to appear in the Apple Menu. The line must start with two waste characters since the MenuManager puts something here. The line must also have a back slash in it (\) and an H followed by two place holder characters. These characters will be replaced with the menu item ID for the desk accessory when FixAppleMenu is called.

Desk Manager Routines

The desk manager has a number of calls but very few that a user would make. These are described below.

Initialization Calls

DeskBootInit — Internal routine called at boot time to initialize the DA manager.

```
Stack Before Call
|  previous contents  |
|                     |<-SP

Stack After Call
|  previous contents  |
|                     |<-SP
```

DeskStartup — Call made by an application before it makes any other desk manager calls.

```
Stack Before Call
|  previous contents  |
|                     |<-SP

Stack After Call
|  previous contents  |
|                     |<-SP
```

An application should only make this call if it is supporting new desk accessories. This means that all the conditions for supporting new desk accessories have been met: required tools are loaded; tools are initialized; menu bar is in place and has edit menu. When the application makes this call the Desk Manager initializes all the NDAs that are installed.

DeskShutdown — Call made by an application before shutdown if it has made any desk manager calls.

```
Stack Before Call
|  previous contents  |
|                     |<-SP

Stack After Call
|  previous contents  |
|                     |<-SP
```

This call is made by applications which make the DeskStartup call and by the ProDOS 16 when it switches from one OS to another. The desk manager makes a call to every desk accessory installed in the system to let it no that this call has been made. New desk accessories are passed an initialization call with the a register set to zero. Classic desk accessories are called via their shutdown entry point.

Every application which supports new desk accessories must make this call before shutting down. It must also be careful to make the call before it shuts down any of the tools a new desk accessory may require.

DeskVersion — Returns the version number of the Desk Manager.

Stack Before Call

previous contents	
space for version	
	<-SP

Stack After Call

previous contents	
version number	
	<-SP

DeskReset — Resets the Desk Manager.

Stack Before Call

previous contents	
	<-SP

Stack After Call

previous contents	
	<-SP

This call should not be made by an application. It is made by the Reset handler built into ROM. It is only called when a user presses CONTROL-RESET.

DeskStatus — Returns whether or not the Desk Manager's startup call has been issued.

Stack Before Call

previous contents	
space for boolean	
	<-SP

Stack After Call

previous contents	
boolean result	
	<-SP

State Save Calls

The state saving calls are internal routines used by the desk manager to preserve the machine state. Careless use of any of these calls could prevent an application being interrupted from running when the current interrupt is over.

SaveScrn — SaveScreen will save the 80-column text screens in bank 00, 01, E0 and E1. This new image of the screen will be used for subsequent calls to RestScreen.

Stack Before Call

previous contents	
	<-SP

```
Stack After Call
     |  previous contents      |
     |                         |<-SP
```

An important thing to note is that only the screen holes used by the Desk Manager are preserved.

RestScrn — RestoreScreen will restore the screen area saved by the Desk Manager.

```
Stack Before Call
     |  previous contents      |
     |                         |<-SP

Stack After Call
     |  previous contents      |
     |                         |<-SP
```

An important thing to note is that only the screen holes used by the Desk Manager are preserved.

SaveAll — Saves all the variables that the Desk Manager preserves when the CDA menu is activated.

```
Stack Before Call
     |  previous contents      |
     |                         |<-SP

Stack After Call
     |  previous contents      |
     |                         |<-SP
```

RestAll — Restores all the variables that the Desk Manager preserves when the CDA menu is activated.

```
Stack Before Call
     |  previous contents      |
     |                         |<-SP

Stack After Call
     |  previous contents      |
     |                         |<-SP
```

HouseKeeping

InstallNDA — Installs the new desk accessory in the system.

Stack Before Call

previous contents	
handle to ID	handle pointing to ID structure of DA
	<-SP

Stack After Call

previous contents	
	<-SP

InstallCDA — Installs the classic desk accessory in the system

Stack Before Call

previous contents	
handle to ID	handle pointing to ID structure of DA
	<-SP

Stack After Call

previous contents	
	<-SP

Classic Desk Accessory Routines

ChooseCDA — Activates the Desk Manager and displays the CDA menu.

Stack Before Call

previous contents	
	<-SP

Stack After Call

previous contents	
	<-SP

ChooseCDA causes the Desk Manager to display the CDA Menu as if the user key stroke interupt has occured. I'm not sure there is any valid reason for a program to make this call.

SetDAStrPtr — Lets a program change the the Classic Desk Accessories built into ROM.

Stack Before Call

previous contents	
Handle to Alt Disp DA	**Handle** to new Alternate display desk accessory
Pointer to New Strings	**POINTER** to table of strings

```
    |                       |<-SP

Stack After Call
    |  previous contents    |
    |                       |<-SP
```

This routine is used to localize the desk accessories in ROM. It allows the built-in desk accessories to have different names.

The alternate display mode desk accessory should contain the following:

```
AltDispDA anop
          dc    i1'StrEnd-StrStart'    ; length of string
StrStart  dc    c'Alternate Display Mode' ; any name you want (within reason)
StrEnd    dc    i4'Open'
          dc    i4'ShutDown'

Open      setmode8                     ; 8 bit m and x
          phb                          ; save data bank register
          lda   #$00                   ; set db reg to $00
          pha
          plb
          jsl   $E100A4                ; Vectored ROM routine to call
          plb                          ; restore data bank reg
          setmode16                    ; 16 bit m and x
ShutDown rtl
```

The table of strings must be in the following form:

```
StringTable   dc    i4'titlestr'     ; title line
              dc    i4'contrlstr'    ; control panel
              dc    i4'quitstr'      ; quit
              dc    i4'selectstr'    ; select string
```

The strings currently used are:

```
titlestr  dc        h'5A A0 41 A0'
          dc        c'Desk Accessories ' ; note space after str
          dc        18h'20'
          dc        h'5F 00'
```

This string must be exactly 39 characters long. If the title ("Desk Accessories" in the above example) changes size then the number 18 in the next line should be changed as appropriate. This number tells the number of inverse spaces to display. It should also be noted that the strings used by the desk manager for titles are C style (zero terminated) strings. Thus there is no length byte to change.

```
cntrlstr dc c'Control Panel'
         dc         h'00'
```

This string has no length requirement other than it must be less than 34 characters.

```
quitstr dc  c'Quit'
        dc          h'00'
```

This string has no length requirement other than it must be less than 34 characters.

```
selectstr dc h'5A'
        dc          c' Select: '      ; note spaces before and after str
        dc          h'4A A0 4B'
        dc          17h'A0'
        dc          c'Open: '         ; note space after str
        dc          h'4D A0 A0 5F 00'
```

This string must be exactly 39 characters long. If the length of the words "Select" or "Open:" changes, the easiest way to modify things is by changing the number 17 in the 4th line above. This will alter the number of spaces drawn between "Select" and "Open".

GetDAStrPtr — Returns the pointer to the table of strings described in the SetDAStrPtr routine.

```
Stack Before Call
    |   previous contents    |
    |    space for result     | LONG
    |      DA id num          |
    |                         |<-SP

Stack After Call
    |   previous contents     |
    | pointer table of strings| POINTER
    |                         |<-SP
```

New Desk Accessory Routines

OpenNDA — Opens the specified DA by ID number.

```
Stack Before Call
    |   previous contents     |
    |   Space for RefNum      | word
    |       ID Num            | ID number returned from Menu Manager (word)
    |                         |<-SP

Stack After Call
    |   previous contents     |
    |       RefNum            |
    |                         |<-SP
```

This call is made when an application discovers that the user has selected an NDA from the Apple Menu. The ID Num passed is the same ID returned by the Menu Manager and set up by the FixAppleMenu call.

CloseNDA Closes the specified DA.

```
Stack Before Call
     |  previous contents     |
     |      RefNum            |
     |                        |<-SP

Stack After Call
     |  previous contents     |
     |                        |<-SP
```

The RefNum is the RefNum returned by the open call. This call is very similar to the call on the Macintosh. Like the Macintosh, it is unlikely to be used by an application since NDAs are closed when the mouse goes down in the CloseBox and SystemClick handles this.

CloseNDAbyWinPtr Closes the NDA whose window pointer is equal to the one that is passed.

```
Stack Before Call
     |  previous contents     |
     |      WinPtr            | POINTER
     |                        |<-SP

Stack After Call
     |  previous contents     |
     |                        |<-SP
```

This call is handy when trying to close a desk accessory because the user chose close from the file menu. When the user chooses close, the application uses the FrontWindow call to see what window is to be closed. If the front window is not an application window, the application can pass the pointer to CloseNDAbyWinPtr.

Possible Errors

NotSysWindow This is returned when the window pointer is not the pointer to a window owned by a NDA.

CloseAllNDAs Closes all open NDAs.

No stack parameters.

FixAppleMenu Adds the names of the NDAs to the specified menu.

```
Stack Before Call
     |  previous contents     |
     |      MenuID            | ID of menu to add DA's to
     |                        |<-SP

Stack After Call
     |  previous contents     |
     |                        |<-SP
```

This call is used to put the names of the currently installed NDAs in a menu (usually the AppleMenu). They are appended to the menu with the first NDA given ID one, the second NDA given ID two and so on.

GetNumNDAs — Returns the number of NDAs currently installed.

```
Stack Before Call
     |   previous contents     |
     |   Space for integer     | word
     |                         |<-SP

Stack After Call
     |   previous contents     |
     |   Number of NDAs        | integer
     |                         |<-SP
```

SystemClick — Called when application detects mouse down in a system window.

```
Stack Before Call
     |   previous contents     |
     |   Ptr to EvtRecord      |
     |      Window Ptr         |
     |   FindWindow Result     |
     |                         |<-SP

Stack After Call
     |   previous contents     |
     |                         |<-SP
```

This call is slightly different than the equivalent Macintosh call. One additional input is passed on the stack. This additional input is the result of the FindWindow call (that is where in the system window the mouse when down). This is different because the Desk Manager has no way to find out this information unless it is passed by the application.

Note: If the application is using TaskMaster, it never needs to make this call. TaskMaster does the work for it.

SystemEdit — Passes standard menu edits to system windows.

```
Stack Before Call
     |   previous contents     |
     |    Space for result     |
     |       EditType          |  Word
     |                         |<-SP

Stack After Call
     |   previous contents     |
     |    Processed Flag       |  BOOLEAN
     |                         |<-SP
```

The valid Edit Types are

1 Undo
2 Cut
3 Copy
4 Paste
5 Clear

The processed flag returns true if the top window is a system window and false otherwise.

SystemTask — Called periodically by an application to support Desk Accessories doing periodic actions.

```
Stack Before Call
     |  previous contents     |
     |                        |<-SP

Stack After Call
     |  previous contents     |
     |                        |<-SP
```

For eack open desk accessory, SystemTask causes the accessory to perform the periodic action defined for it, if any such action was defined and if the proper time period has passed since the action was last performed. For example, a clock accessory can be defined such that the second hand is to move once every second; the periodic action for the accessory will be to move the second hand to the next position, and SystemTask will alert the accessory every second to perform that action.

Note: If the application is using TaskMaster, it never needs to make this call. TaskMaster does the work for it.

SystemEvent — This is the entry point the Event Manger uses to the Desk Manager.

```
Stack Before Call
     |  previous contents     |
     |  space for boolean     | word
     |    Event What          | word
     |   Event Message        | long
     |    Event When          | long
     |    Event Where         | point (long)
     |    Event Mods          | word
     |                        |<-SP

Stack After Call
     |  previous contents     |
     |       boolean          |
     |                        |<-SP
```

System Event returns true if the event is processed by a DA and false if it is to be sent on to the application. The CDA activation keystroke is processed in this way.

Note: An application would never make this call.

Summary of Calls

Required Tool Calls

DeskBootInit
DeskStartup
DeskShutdown
DeskVersion
DeskReset
DeskStatus

State Saving Calls

SaveScrn
RestScrn
SaveAll
RestAll

HouseKeeping Calls

InstallCDA
InstallNDA

Classic Desk Accessory Calls

ChooseCDA
SetDAStrPtr
GetDAStrPtr

New Desk Accessory Calls

OpenNDA
CloseNDA
CloseNDAbyWinPtr
CloseAllNDAs
FixAppleMenu
GetNumNDAs
SystemClick
SystemEdit
SystemTask
System Event

Documentation Développeurs
Apple Computer France 1987

Document développeur numéro 56

Window Manager

type d'upgrade de ce ducument : 5
1 Documentation de première catégorie inchangée
2 Documentation de deuxième catégorie mise à jour
3 Documentation de deuxième catégorie inchangée
4 Mise à jour payante de la documentation de première catégorie
5 Mise à jour gratuite de la documentation de première catégorie
6 Nouveautés payantes non vitales
7 Nouveautés gratuites et vitales

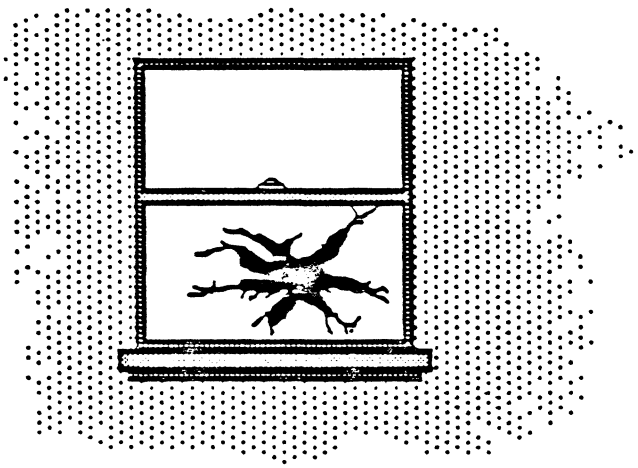
Taille : 80 page(s) environ

Domaine : Tool 14

VERSION : 1.0
DATE : 25.09.86

Cortland Window Manager

Dan Oliver



This ERS corresponds to the Beta release of the Window Manager, version 1.0. There will be no further changes to the Window Manager that will compromise applications written for this version. However, document errors will be changed to conform to actual code. Over the next few months I will be fixing bugs and releasing appendixes that detail Window Manager functions. If you have areas you would like clarified please let me know and I will try to publish an appendix. Send comments to:

Apple Computer, Inc.
20525 Mariani Ave, MS: 22X
Cupertino, CA 95014

ATTN: Dan Oliver

September 25, 1986

01/30/86 Initial release

06/05/86 Revised release. Note the removal of direct access to window records by applications and additional calls to compensate. The inputs to **NewWindow** has become more Mac like, and **DisposeWindow** is folded into **CloseWindow**.

06/10/86 Updates to page 4, and page 6 in the appendix.

06/14/86 Updates to pages 15-20.

06/18/86 Name changes to **BootWmgr**, **InitWindows**, **TermWindows**, and **WmgrVersion**. Addition of **WindReset** and **WindStatus**, although not completed. Additional parameter, user ID, passed to **WindStartup** (formerly **InitWindows**). **TaskMaster** uses an extended event record.

07/15/86 Expanded **SetFrameColor**. **WNewRes** doesn't redraw the screen anymore. Changes to color table in "WINDOW FRAME COLORS AND PATTERNS". New input to **NewWindow**. New calls; **GetCOrgin**, **SetCOrigin**, **GetDataSize**, **SetDataSize**, **GetMaxGrow**, **SetMaxGrow**, **GetScroll**, **SetScroll**, **GetPage**, **SetPage**, **GetCDraw**, **SetCDraw**, **GetInfoDraw**, **SetInfoDraw**, **StartDrawing**. New sections DRAW CONTENT ROUTINE and DRAW INFORMATION BAR ROUTINE. New input parameters to **MoveWindow**. Defaults added to **DragWindow**. Parameters expressed as "POINT" are now broken down into two WORDs.

07/16/86 Replacement for pages 6-8 in the appendix which labels **NewWindow** parameter list. Insert pages 15.a-15.c between pages 15 and 16. These pages define DRAW CONTENT ROUTINE and DRAW INFORMATION BAR ROUTINE as promised in the last release.

08/13/86 Two bits added to wframe field of window record (see **NewWindow**). **SetOrgnMask** call added for scrollable windows that use color dithering in 640 mode. Parameter length field added to parameter list passed to **NewWindow**. **SetWMgrIcons** call added along with a WINDOW MANAGER ICON FONT section. **TaskMaster** returns window pointer in TaskData field rather than the message field.

08/27/86 Version 81.9. **DragRect** removed (implemented in Control Manager). Document correction to **SendBehind**. Changed names of **GetInfoText** to **GetInfoRefCon**, **SetInfoText** to **SetInfoRefCon**. Added wInfoHeight to input parameter to **NewWindow**, you can now select the height of window information bars. Added functions; **GetRectInfo**, **StartInfoDrawing**, **EndInfoDrawing**. Parameter change to **Refresh**. Added F_ALERT to wframe. TaskMask field in the TaskRec is expanded from a WORD to a LONG and another bit is defined. ERS corrections to **GetCOrigin**, **SetPage**, **PinRect** and **CheckUpdate**.

09/02/86 **PinRect** theRect is a pointer to a RECT, not a RECT.

Tool Number: 14

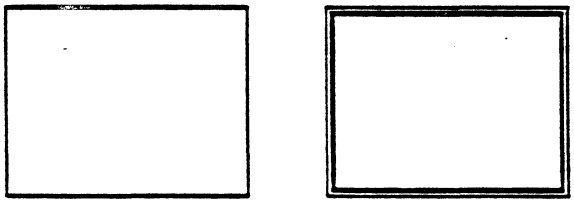
Tools needed installed: Quick Draw
Memory Manager
Event Manager

Stack requirement: 512 bytes, when also using the Menu and Control Managers.

ABOUT THE WINDOW MANAGER

The Window Manager is a tool for dealing with windows on the Cortland screen. The screen represents a working surface or **desktop**; graphic objects appear on the desktop and can be manipulated with a mouse. A **window** is an object on the desktop that presents information, such as a document or a message. Windows can be any size or shape, and there can be one or many of them, depending an the application.

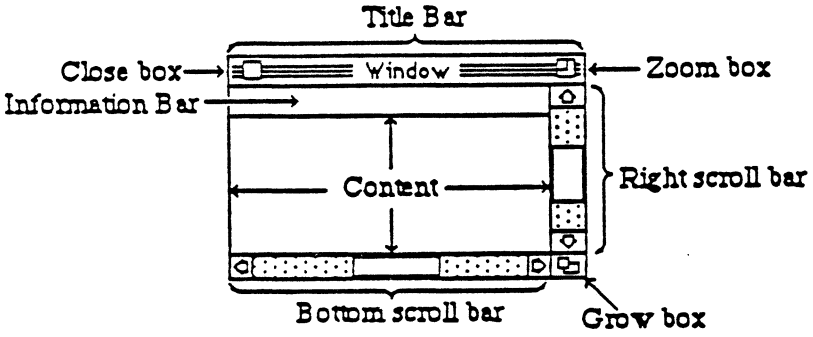
There are two kinds of predefined window frames, **document** and **alert.**



Document Alert

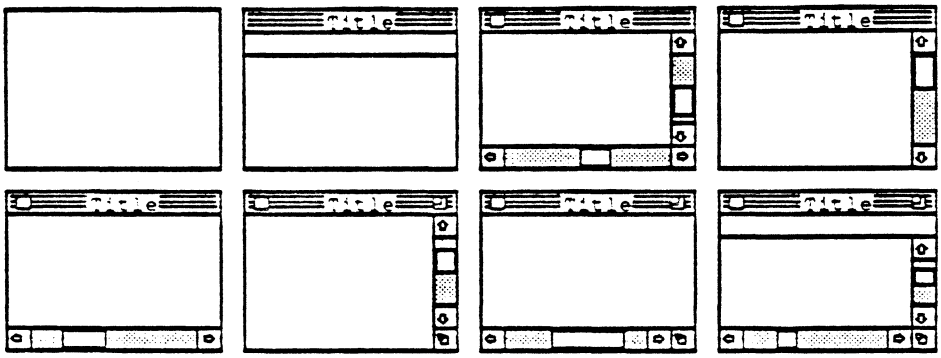
The alert window is used by the Dialog Manager and is explained in that ERS.

Inside the document window can be **standard window controls,** which are; title bar, close box, zoom box, right scroll bar, bottom scroll bar, grow box, and information bar. The title bar displays the window's title, can hold the close and zoom boxes, and can be a drag region for moving the window. The close box is selected by the user to remove the window from the screen. The zoom box is selected by the user to make the window its maximum size and to return it to its previous size and position. The right scroll bar is used to scroll vertically through the data in the window. The bottom scroll bar is used to scroll horizontally through the data in the window. The grow box is dragged by the user to change the size of the window. The information bar is a place an application can display some information that won't be effected by the scroll bars.



A document window may have any or all of the standard window controls. The only restriction is that if there is a close or zoom box there must also be a title bar. Common sense would dictate that there only be a zoom box if there is a grow box, although this is not a requirement.

No standard controls may be added to a alert window. Here are some possible document window combinations:



Your application can easily use standard window types, or create your own window types (see DEFINING YOUR OWN WINDOWS). Some windows may be created indirectly for you when you use other parts of the Toolbox; an example is the window the Dialog Manager creates to display an alert. Windows created either directly or indirectly by an application are collectively called **application windows**. There's also a class of windows called **system windows**; these are the windows in which desk accessories are displayed.

The Window Manager's main function is to keep track of overlapping windows. You can draw in any window without running over onto windows in front of it. You can move windows to different places on the screen, change their **plane** (front-to-back order), or change their size, all without concern for how the various windows overlap. The Window Manager keeps track of any newly exposed areas and provides a convenient mechanism for you to ensure that they are properly redrawn.

Finally, you can easily set up your application so mouse actions cause these standard responses inside a document window, or similar responses inside other windows:

- Clicking anywhere in an inactive window makes it the active window by bring it to the front and highlighting it.

- Clicking inside the close box of the active window closes the window. Depending on the application, this may mean that the window disappears altogether, or a representation of the window (such as an icon) may be left on the desktop.

- Dragging anywhere inside the title bar of a window (except in the close or zoom boxes, if any) pulls an outline of the window across the screen, and releasing the mouse button moves the window to the new location. If the window isn't the active window, it becomes the active window unless the Command key was also held down. A window can never be moved completely off the screen; by convention, it can't be moved such that the visible area of the title bar is less than four pixels square.

- Dragging inside the size box of the active window changes the size of the window.

WINDOW REGIONS

Every window has the following two regions:

- The **content region**: the area that your application draw in

- The **frame region**: the outline of the entire window plus any standard window controls.

Together, the content and frame regions makeup the **structure region.**

The content region is bounded by the rectangle you specify when you create the window (that is, the portRect of the window's grafPort) The content region is where your application presents information to the user.

A window may also have any of the regions listed below within the window frame.

- A **go-away region** , a close box in the active window. Clicking in this region closes the window.

- A **drag region,** the title bar. Dragging in this region pulls an outline of the window across the screen, moves the window to a new location, and makes it the active window (if it isn't already) unless the Command key was held down.

- A **grow region,** the grow box. Dragging in this region pulls the lower right corner of an outline of the window across the screen with the window's origin fixed, resizes the window, and makes it the active window (if it isn't already) unless the Command key was held down.

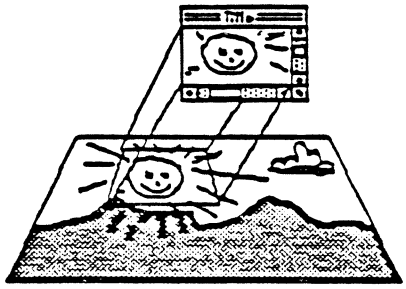
- A **zoom region,** the zoom box in the active window. Clicking in this region toggles between the current position and size to a maximum size, and back again.

Clicking in any region of an inactive window simply makes it the active window.

Note: The results of clicking and dragging that are discussed here don't happen automatically, unless you are calling TaskMaster; you have to make the right Window Manager calls to cause them to happen.

CONTENT REGION AND WORK AREA

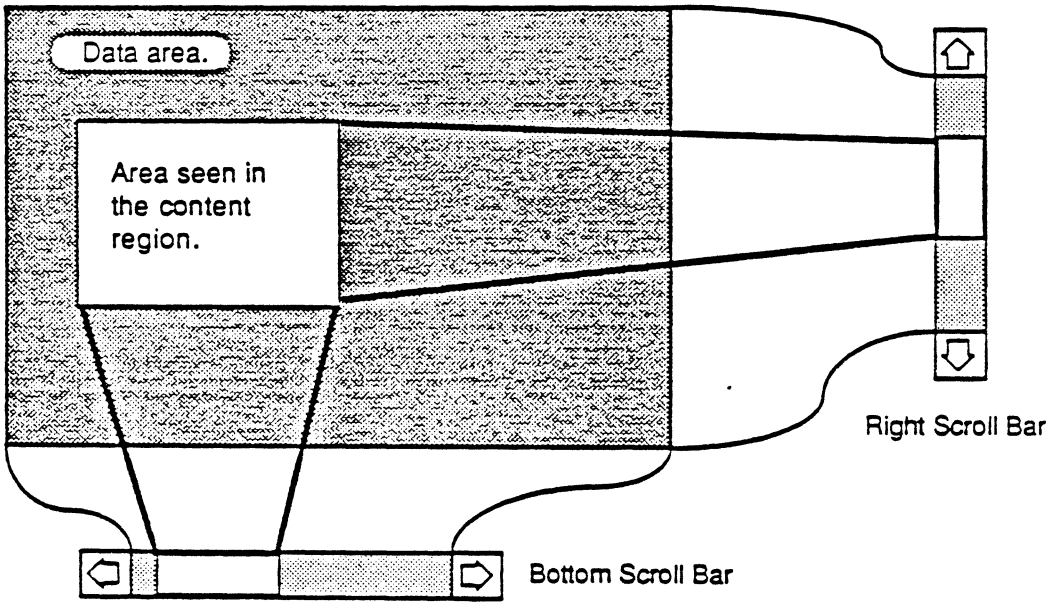
What is the purpose of windows any way? Windows are used to present more information than the hardware (screen) can display at one time, and do it in a standard way. The name window is used because the user sees through the window into a larger area. The power of windows is their ability to give the user a standard device for accessing large amounts of data. Windows act like a microfiche viewer. What is seen on the viewer is like what is seen in the window's content region. And the window's **data area** is what the microfiche is to the viewer. Through the content region the user can see part of the data area, unless the content region is large enough to view the entire data area. Scroll bars are used to scroll the data area through the content region. The grow box and zoom box are used to display more, or less, of the data area at one time. When the window is moved, the data area is moved with it, so the view in the content remains the same.



WINDOW SCROLL BARS

Window scroll bars are the devices used for scrolling the data area through the content region and showing the relationship between the data area and content region. The Control Manager must be installed in order to use scroll bars in windows. Scroll bars are handled by the Control Manager but this document will go over how standard window scroll bars act relating to windows.

The scroll bar is like a reduced cross section of the work area. The scroll thumb is the same ratio to the page region as the content region is to the data area.

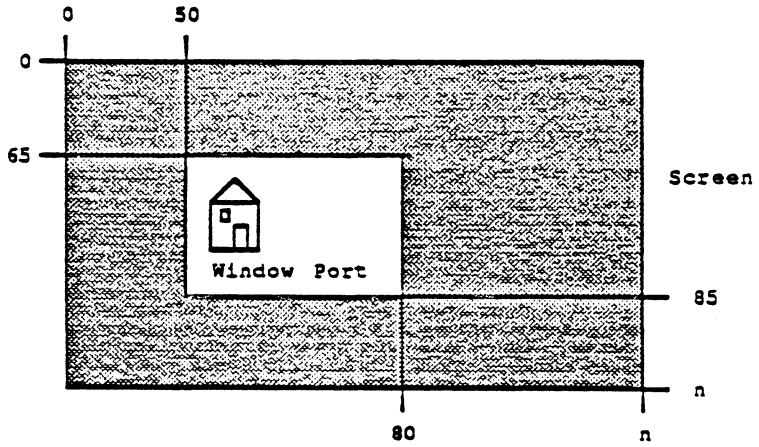


Origin Movement

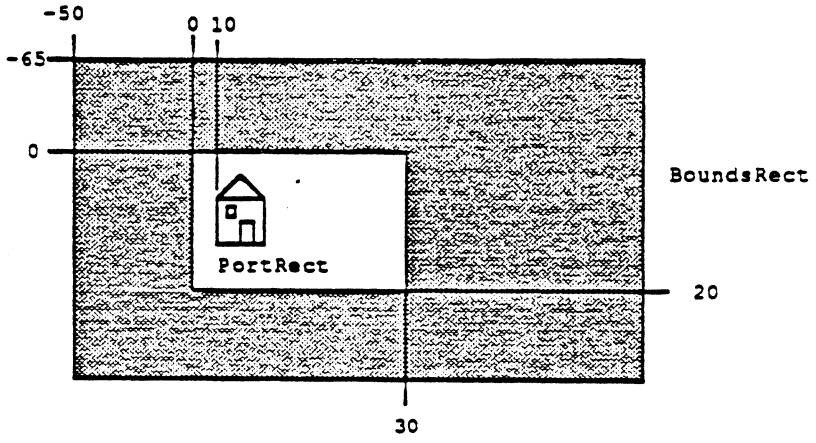
This section goes into detail on how the origin of a window can change and what the effects are. You should already be familiar with QuickDraw's explanation of ports and boundRect.

The origin of a window is what allows data to be scrolled and drawing to occur in the proper place after a scroll. PortRects, BoundRects and origins are not intuitive concepts so the following diagrams will walk through what might happen with an origin and point out some places of possible misunderstanding.

We will start with a case that everyone should understand with little explanation. The gray area is a screen with the pixel in its upper left corner being 0,0 (coordinates are shown here as y,x). The window port appears on the screen at 65,50 to 85,80. These points are called global coordinates. To draw the house the x coordinate of the left side would be 60, that is 10 pixels inside the window port.

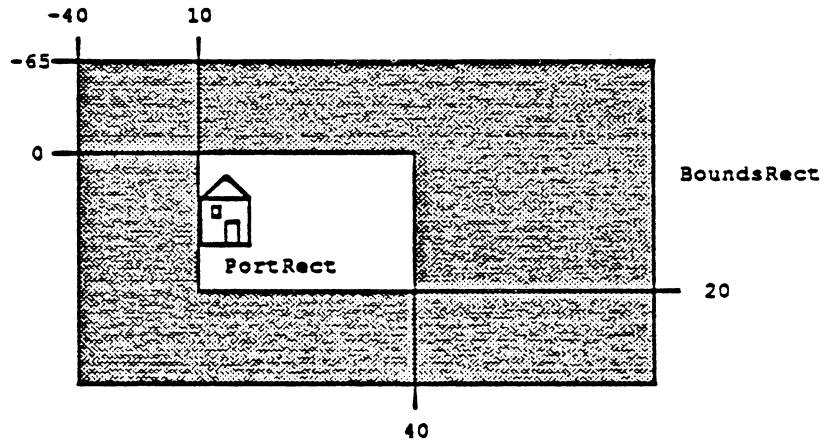


However, a window port is almost a screen in that it has its own coordinate system, called the local coordinate system. The following diagram shows what really happens when a window is created. Although the window is still at 65,50 to 85,80 on the screen the local coordinates of the window are 0,0 to 20,30. Notice that its height and width are the same. Also notice that I'm now referring to the window as PortRect and the screen as BoundsRect. These are terms used by QuickDraw. At this point you don't really have to understand BoundsRects and PortRects other than some simple relationships. First I will only refer to the horizontal axis, although the same thing happen with vertical axis. To draw the left wall of this house you would pass the x coordinate of 10 to Quick draw to draw a single vertical line. QuickDraw would then subtract the x origin of the BoundsRect to determine where on the screen to actually draw the line. So, 10 less -50 is 60. 60 is the global coordinate, 10 was the local. You always work with local coordinates. The reason applications work in local coordinates is so where ever the window is moved, its coordinate system is the same. The Window Manager will change the BoundsRect when the window is moved. If the window is moved one pixel to the left the BoundsRect would become -65,-49. When the coordinate of 10 is passed to QuickDraw it would compute the global coordinate of 59 (10 less -49). So, the house would be drawn in other place on the screen, but the same place in the window, and the application doesn't have to make any changes.



In the previous examples all had origins of 0,0 and require almost no understanding on the part of the programmer to write an application. However, one the powerful features of windows is their ability to scroll to show more data than the screen allows. In the next example the window has not moved, but the user has scrolled the picture using the bottom scroll bar. When the user moved the thumb on the scroll bar the rectangle from 0,10 to 20,30 was scrolled (moved) to 0,0 to 20,20. Then the origin of the window was changed to 0,10 the exposed rectangle on the right side was redrawn in an update event.

After the scroll the application would pass the coordinate of 10 to draw the left side of the house and QuickDraw would compute 10 less -40 to get the global coordinate of 50. Things would not be to bad at this point except for a problem the Window Manager has. The Window Manager needs every window to have an origin of 0,0 in order for it to move, grow and overlap windows. This is an inefficiency in the Window Manager which will hopefully be fixed in the far future. For now, whenever the Window Manager is called, the origin of every window must be 0,0.



Here's the plan. Whenever TaskMaster calls your update routine its will switch to the window's port and set its origin, in this example it would be **SetOrigin**(0,10). Then you can draw in the window's local coordinates. When you are finished drawing and return back to TaskMaster it will perform a **SetOrigin**(0,0). Here's where the weirdness creeps in. Changing the origin does not change the screen, however any drawing outside of your update routine without setting the origin would have undesirable results. Drawing your house with the origin still at 0,0 would produce two houses, one 10 pixels to the right of the other. To draw outside of your update routine you need to first set the origin either yourself or through **StartDrawing** and then perform a **SetOrigin**(0,0) to put it back.

In short, when drawing outside of your update routine you must perform a **StartDrawing** before drawing and a **SetOrigin**(0,0) when you are finished drawing. This is also true for when you are performing a hit test in your content region, the event position must be converted to local coordinates.

USING THE WINDOW MANAGER

To use the Window Manager, you must have previously initialized QuickDraw and the Event Manager. The first Window Manager routine to call is the initialization routine, **WindStartup**.

Where appropriate in your program, use **NewWindow** to create any windows you need.

Now you have a choice to make. There are two ways to handle user input in relation to windows. You can poll the user via **GetNextEvent**, and decide what to do with events, or poll via **TaskMaster**, which will handle most events dealing with standard user interfaces (see USING TASKMASTER).

If you are not using **TaskMaster**, you must poll for events by calling **GetNextEvent** in the Event Manager. For button down events, call **FindWindow**, to see if the button was pressed inside a window. The following are results from **FindWindow** and the standard actions to take:

wInMenuBar	Button passed somewhere outside of the desktop. If you have not subtracted any area from the desktop, there is a good chance it was pressed in the system menu bar. Call **MenuSelect** in the Menu Manager.
wInDrag	Button pressed in a window's drag region, in may not be the active window however. Call **DragWindow**.
wInContent	Button pressed in window's content region. Call **SelectWindow** if the window is not the active window. Otherwise, handle the event according to your application.
wInGoAway	Button pressed in active window's close region. Call **TrackGoAway**. If **TrackClose** returns TRUE, call **CloseWindow**, or **HideWindow**, perhaps after saving whatever the user was working on inside the window.
wInZoom	Button pressed in active window's zoom region. Call **TrackZoom**. If **TrackZoom** returns TRUE, call **ZoomWindow**.
wInGrow	Button pressed in active window's grow region. Call **GrowWindow**.

USING TASKMASTER

TaskMaster is a procedure that can handle many standard functions. TaskMaster is called instead of GetNextEvent and the first thing TaskMaster does, is call GetNextEvent. If there isn't an event ready, TaskMaster will return zero. If an event is ready, TaskMaster will look at the event and try to handle it. If the event can not be handled at all, the event code is returned and the application can handle the event just like returning from GetNextEvent. If TaskMaster can handle the event, it will call standard functions to try and complete the task. For example, if the user presses the mouse button in an active window's zoom region, TaskMaster will detect it and call TrackZoom, then call ZoomWindow if the user actually selects the zoom region, and return no event. However, sometimes TaskMaster can handle an event only up to a point. If the user presses the mouse in the active window's content region, TaskMaster will detect it, but won't be able to go any further, so it returns wInContent, which tells the application the mouse button is down the active window's content region.

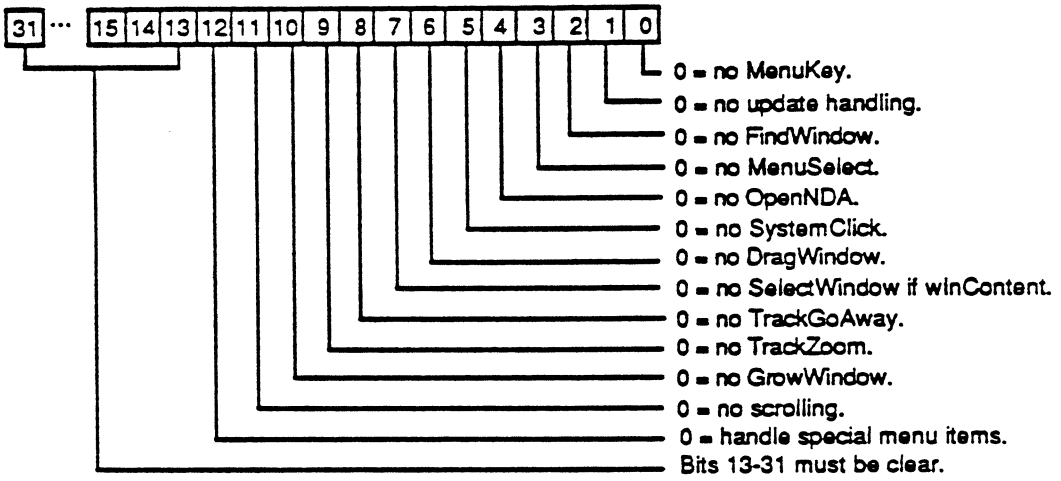
TaskMaster is provided for two reasons. The first is to make it easier for a programmer to get an application up and running as quickly as possible, and still take advantage of all the standard user interfaces. Although TaskMaster was tailored for beginning programmers, advanced programmers may find it useful when writing small, inhouse, applications. The second reason for TaskMaster is aimed at applications that are sold and will hopefully be around for years, if not weeks, to come. In the future, applications might have to be modified to take advantage of some new, as yet unknown, feature. If an application is using TaskMaster, it may be possible to make the modification to TaskMaster, without adversely affecting past applications, so your application will be using the new feature without any modification on your part.

TaskMaster is one of the steps taken to remove the user interface duties from the application, as most operating system have done in the past. TaskMaster should be usable by even the most advanced applications, although some alternate algorithms may have to be used in order to get the desired results.

When calling TaskMaster you pass a pointer to a TaskMaster record, TaskRec. The beginning of the record is the same as an event record. When TaskMaster calls GetNextEvent, it will pass the pointer given, so the event record part of TaskRec is set by GetNextEvent. The structure of TaskRec is:

what	WORD	Event record portion, unchanged from GetNextEvent.
message	LONG	
when	LONG	
where	LONG	
modifiers	WORD	
TaskData	LONG	Extended portion for TaskMaster.
TaskMask	LONG	

The TaskMask is used by your application to tell **TaskMaster** about functions you would not like it to perform. To perform everything **TaskMaster** is capable of TaskMask should be $00001FFF. Bits are clear in TaskMask to disable features. See **TaskMaster** function call for an outline of **TaskMaster** features and places TaskMask would cause a break. TaskMask is defined as:



It is important that bits 13-31 be clear. In fact, **TaskMaster** will return an error if they are not. The bits are for future features which will continue to run with predated applications because bits 13-31 will mask off the new, unknown to current applications, features.

Window type return codes from **TaskMaster** are:

inUpdate - The window, who's pointer is stored in the TaskData field of TaskRec, needs to be redrawn. Call **BeginUpdate**, draw the visRgn or the entire content region, and call **EndUpdate**.

wInMenuBar - The user made a selection from the system menu bar and the ID of the item selected is stored in the low-order WORD TaskData, the menu ID is in high-order WORD of TaskData. Handle the menu selection and unhighlight the menu's title when you have completed the requested action.

wInContent - This means the mouse button has been pressed inside the content region of the active window. The window's pointer is in the TaskData field of TaskRec. Because the event happen inside an area your application controls, it is up to you to handle this event in the manner you chose.

wInGoAway - User has selected the window's go-away region. Do what's needed and call **CloseWindow** with the window pointer in the TaskData field of TaskRec.

WINDOW MANAGER ICON FONT

The standard document window definition uses a font to draw the close and zoom boxes, and their highlighted states, in a window's title. If you would like to use different icons you can replace the default font. To replace the icon font, or just get the handle to the current font, call **SetWMgrIcons**. The format of the font is as follows:

Character 0	Close box.
Character 1	Highlighted close and zoom boxes.
Character 2	Zoom box.

WINDOW RECORDS

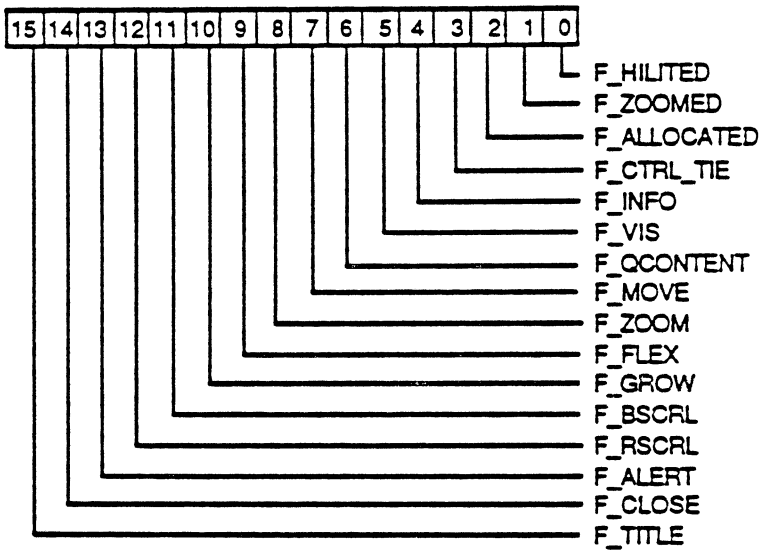
The Window Manager keeps all the information it requires for its operations on a particular window in a **window record**. The record contains the window's grafPort, title pointer, position, size of work area, a reserved long for the application, and other flags the Window Manager needs to manage windows. The complete window record is accessed directly only by the Window Manager. Application access to record information is restricted to calls through the Window Manager and directly to the first part of the window record.

Not allowing direct access to the entire window record has good and bad sides. Access to window information will be slower if calls to the Window Manager have to be made. However, the delay would have to be measured in milliseconds, and the delay never seen on the screen. On the plus side, future Window Managers would not be tied to an older, possibly inadequate, record structure. The chances of improving the current Window Manager, and maintaining compatibility across future hardware, is greatly improved by allowing records to change.

Many Window Manager calls require a window pointer that is returned from **NewWindow**. That pointer is the pointer to the window's grafPort.

The part of the window record that is defined is:

wnext	LONG	Pointer to the next window in the window list.
wport	BYTE[186]	Window's port. Returned window pointers return a pointer to here.
wstrucRgn	LONG	Handle of window's entire region, the frame plus content.
wcontRgn	LONG	Handle of window's content region.
wupdateRgn	LONG	Handle of region that is the part of the content that needs redrawing.
wcontrol	LONG	Handle of application's first control in content region.
wFrameCtrl	LONG	Handle of frame's first control.
wframe	WORD	Bit vector that describes window.



F_HILITED	1 = frame is highlighted, 0 = unhighlighted.
F_ZOOMED	1 = currently zoomed, 0 = not zoomed.
F_ALLOCATED	1 = record was allocated, 0 = record was provided by application.
F_CTRL_TIE	1 = control's state is independent, 0 = inactive window has inactive controls.
F_INFO	1 = information bar, 0 = no information bar.
F_VIS	1 = currently visible, 0 = window is invisible.
F_MOVE	1 = title bar is a drag region, 0 = no drag region.
F_ZOOM	1 = zoom box on title bar, 0 = no zoom box. (Zoom box must have title bar.)
F_FLEX	1 = **GrowWindow** and **ZoomWindow** won't change the origin.
F_GROW	1 = grow box, 0 = no grow box. (Grow box must have at least one scroll bar.)
F_BSCRL	1 = window frame horizontal scroll bar, 0 = no horizontal scroll bar.
F_RSCRL	1 = window frame vertical scroll bar, 0 = no vertical scroll bar.
F_ALERT	1 = alert type frame (don't set grow, close, infobar, title bar, or scrolls).
F_CLOSE	1 = close box, 0 = no close box. (Close box must have title bar.)
F_TITLE	1 = title bar, 0 = no title bar.

(The remainder of the record is undefined)

WINDOWS AND GRAFPORTS

It's easy for applications to use windows: To the application, a window is a grafPort that it can draw into with QuickDraw routines. When you create a window, you specify a rectangle that becomes the portRect of the grafPort in which the window contents will be drawn. The bit map for this grafPort, its pen pattern, and other characteristics are the same as the default values set by QuickDraw. These characteristics will apply whenever the application draws in the window, and they can easily be changed with QuickDraw routines.

There is, however, more to a window than just the grafPort that the application draws in. The other part of a window is called the **window frame**, since it usually surrounds the rest of the window. For drawing window frames, the Window Manager creates a grafPort that has the entire screen as its portRect; this grafPort is called the **Window Manager port**.

WINDOW FRAME COLORS AND PATTERNS

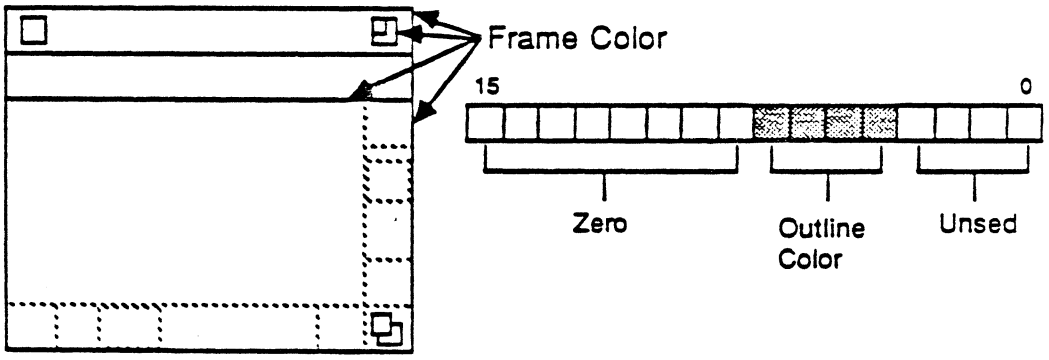
In addition the to the standard window types and controls, the color of the window and controls can be selected. Colors are selected from a color table which you either pass when creating a window, or a default table. The color table for a document window is:

FrameColor	WORD	Color of window frame.
TitleColor	WORD	Color of inactive bar, inactive title, and active title.
TBarColor	WORD	Color and pattern of active title bar.
GrowColor	WORD	Color of grow box.
InfoColor	WORD	Color information bar background.

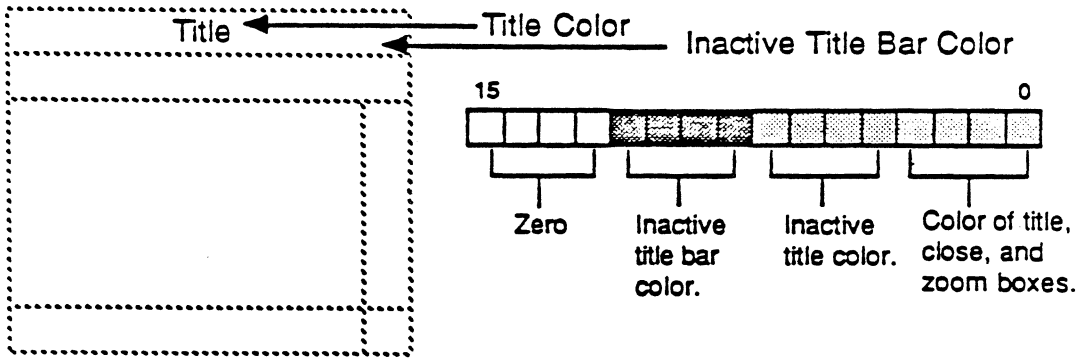
Use **SetFrameColor** to set the color table a window should use, and **GetFrameColor** to get a pointer to the window's current color table.

The following diagrams show how these colors are used.

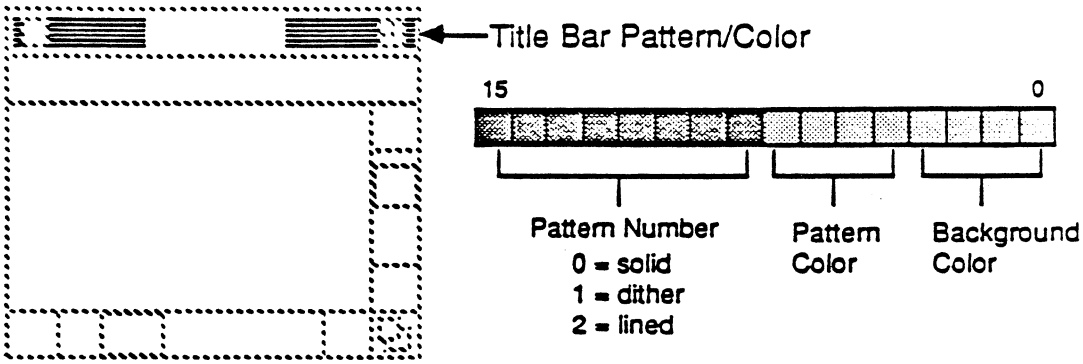
FrameColor:



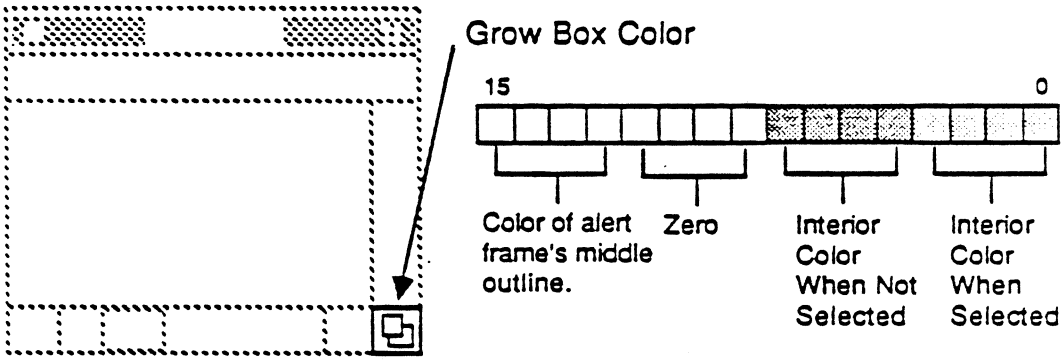
TitleColor:



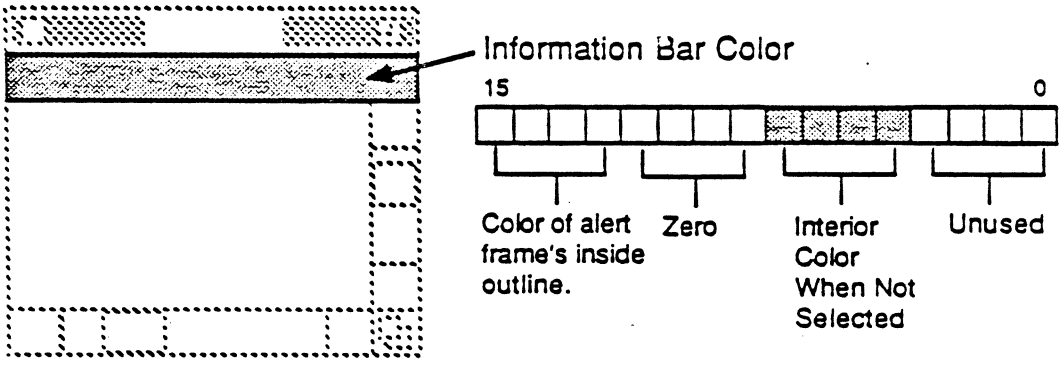
TBarColor:



GrowColor:



InfoColor:



HOW A WINDOW IS DRAWN

When a window is drawn or redrawn, the following two-step process usually takes places: the window frame is drawn, then the window contents.

To perform the first step of this process, the Window Manager manipulates regions of the Window Manager port as necessary to ensure that only what should be drawn is drawn. It then calls the window definition function with a request that the window frame be drawn. The window definition function is either within the Window Manager or in the application for custom windows (see DEFINING YOUR OWN WINDOWS).

For the second step the Window Manager generates an **update event** to get the application to draw the window contents. It does this by accumulating, in the update region, the areas of the window's content region that need updating. The Event Manager periodically calls **CheckUpdate** to see if there's any window whose update region is not empty; if it finds one, it reports (via the **GetNextEvent** function) that an update event has occurred, and passes along the window pointer in the event message. Update events will be issued to the front most window first and the bottom most last. The application should respond as follows:

1. Call **BeginUpdate**. This procedure temporarily replaces the visRgn of the window's grafPort with the intersection of the visRgn and the update region. It then clears the update region for that window.

2. Draw the window contents.

3. Call **EndUpdate** to restore the actual visRgn.

DRAW CONTENT ROUTINE

If wContDefProc is non-zero, the value will be considered the address of a routine in your application that will draw the window's content region. WContDefProc must be set if you want to use window frame scroll bars. TaskMaster will scroll the content and call wContDefProc to update the uncovered area when the use performs a scrolling action. WContDefProc could be considered a control action procedure.

WContDefProc might be useful even if you are not using window frame scroll bars. TaskMaster will be able to handle you update events if wContDefProc is set. TaskMaster will call BeginUpdate, wContDefProc and EndUpdate. (TaskMaster does take some short cuts in calling BeginUpdate and EndUpdate because its part of the Window Manager.) Along these lines, if you are using window frame scroll bars, and therefore TaskMaster, you will not get update events, because they are handled by TaskMaster.

There are no inputs or outputs to your draw content routine. Simple draw what is needed in the content and perform a RTL to exit. Remember that the content will have already been erased using the window port's background pattern, and the visRgn is set to the area needing to be redrawn.

Warning: Do not change ports or perform a SetOrigin call while in your draw content routine.

DRAW INFORMATION BAR ROUTINE

If bit 4 in wFrame of the NewWindow parameter list is set, the window will have an information bar. An information bar is the width of window and a height pass in the parameter list to NewWindow. It will appear just above the content region.

The Window Manager draws the empty information bar, but it is up to the application to draw any information inside. Your application can do this by storing the address of an information bar draw routine in wInfoDefProc of the NewWindow parameter list (also set bit 4 of wFrame). When the standard window frame defProc draws the empty information bar, it will also call wInfoDefProc. The inputs to your routine will be:

InfoBar:LONG	Pointer to information bar enclosing RECT.
InfoData:LONG	wInfoRefCon value from NewWindow parameter list.
theWindow:LONG	Pointer to window's port.

An information bar draw routine that just prints a string might look like this:

```
InfoDefProc  START
;
theWindow    equ   6
InfoData     equ   theWindow+4
InfoBar      equ   InfoData+4
;
;
             phd                        Save the current direct page.
;
             tsc
             tcd                        Switch to direct page in stack.
;
;
; -- Position the pen at the text starting point ----------------------------
;
             ldy   #left_side           (Where left_side equals 2.)
             lda   [InfoBar],y          Get the left side of the information bar,
             clc
             adc   #20                  plus a tab over,
             pha                        to get a starting x position. (Pass to _MoveTo.)
;
             ldy   #top_side            (Where top_side equals 0.)
             lda   [InfoBar],y          Get the top side of the information bar,
             clc
             adc   #10                  plus enough to vertically center the text,
             pha                        to get a starting y position. (Pass to _MoveTo.)
;
             _MoveTo                    Move the pen to the starting point.
;
;
; -- Print the text on the information bar ----------------------------------
;
             pea   infoStrgl-16         Pass high word of string.
             pea   infoStrg             Pass low word of string.
             _DrawString                Print the string.
;
```

```
;
; --- All done, now clean-up stack and return to Window Manager ----------
;
            ply                     Get original direct page back.
;
            lda   2,s               Move return down over input parameters.
            sta   <14               Works because stack and direct page are equal.
            lda   0,s
            sta   <12
;
            tsc                     Now move stack pointer over input parameters.
            clc
            adc   #12               Number of bytes of input parameters.
            tcs                     New stack.
;
            tya                     Restore original direct page.
            tcd
;
            rtl                     Back to Window Manager.
;
            END
```

I have taken some liberties here, such as taking for granted the color and writing mode of the pen when the text is written. When entered, the current port is the Window Manager's. It is permissible to change the pen location, color, and writing mode without saving the original port state. However, that's as much as you should do without first saving the port state, and then restoring it on exit.

Warning: Do not change the current port's clip or vis regions, unless you save and restore the original.

Another liberty taken is when the text is centered vertically. You should make QuickDraw calls to find font height, find the InfoBar height, and then actually center the text. You should always use InfoBar as offsets into the information bar interior, they could be different from time to time.

And of course 'infoStrg' would have to be defined.

MAKING A WINDOW ACTIVE: ACTIVATE EVENTS

A number of Window Manager routines change the state of a window from inactive to active or from active to inactive. For each such change, the Window Manager generates an **activate event**, passing along the window pointer in the event message. The activeFlag bit in the modifiers field of the event record is set if the window has become active, or cleared if it has become inactive.

When the Event Manager finds out from the Window Manager that an activate event has been generated, it passes the event on to the application (via the **GetNextEvent** function). Activate events have the highest priority of any type of event.

Usually when one window becomes active another becomes inactive, and vice versa, so activate events are most commonly generated in pairs. When this happens, the Window Manager generates first the event for the window becoming inactive, and then the event for the window becoming active. Sometimes only a single activate event is generated, such as when there's only one window in the window list, or when the active window is permanently disposed of (since it no longer exists).

Activate events for dialog and alert windows are handled by the Dialog Manager. In response to activate or inactivate events for windows created directly by your application, you might take actions such as the following:

- Inactivate controls in inactive window, and activate controls in active windows.
- In a window that contains text being edited, remove the highlighting or blinking cursor from the text when the window becomes inactive and restore it when the window becomes active.
- Enable or disable a menu or certain menu items as appropriate to match what the user can do when windows become active or inactive.

DEFINING YOUR OWN WINDOWS

You may want to define your own type of window - maybe a round or hexagonal window, or even a window shaped like an apple. QuickDraw and the Window Manager make it possible for you to do this.

To define your own type of window, you write a routine that can will duplicate some Window Manager functions. When the Window Manager needs to do something it will call your routine and not its own. The address of the routine is passed to CreateWindow. The inputs to your routine will be:

```
varCode:WORD - operation needed to be performed.
theWindow:LONG - pointer to the window's port.
Param:LONG - flag used by some messages.
```

Output will be:

```
outCome:LONG - returned flag.
```

Offsets into the stack are:

Param	= 4
theWindow	= Param+4
varCode	= theWindow+4
outCome	= varCode+2

Your routine must strip off the three input parameters and return via RTL. So, the shell of your defProc routine might be:

```
MyWindow  START
          lda   12,s                Get varCode.
          asl   a
          tax
          lda   >actions,x
          pha
          rts                       Go to action handler.

actions   dc    i2'draw_wind-1'     Routine to draw the window's frame.
          dc    i2'test_hit-1'      Routine that find a window region at a given point.
          dc    i2'calc_rgns-1'     Compute the window's structRgn and contRgn.
          dc    i2'init_wind-1'     Do additional initialization.
          dc    i2'kill_wind-1'     Do additional disposal.
          END

draw_wind START
          :
     Draw window frame.
          :
          jmp   exit
          END

test_hit  START
          :
     Find what area of the window the point in Param is located.
          :
          jmp   exit
          END

calc_rgns START
          :
     Compute the window's structRgn and contRgn.
          :
          jmp   exit
          END

init_wind START
          :
     Perform additional initialization.
          :
          jmp   exit
          END

kill_wind START
          :
     Perform additional disposal.
          :
          jmp   exit
          END
```

```
exit    START
;
        lda  2,s                Move return address.
        sta  12,s
        lda  1,s
        sta  11,s
;
        tsc                     Strip off input parameters.
        sec
        sbc  #10
        tcs
;
        rtl                     Return to Window Manager.
        END
```

varCode will be:

wDraw	= 0	Draw window frame.
wHit	= 1	Tell what region mouse button was pressed in.
wCalcRgns	= 2	Calculate wstrucRgn and wcontRgn.
wNew	= 3	Do any additional window initialization.
wDispose	= 4	Take any additional disposal actions.

The following sections tell you what is expected is response to the varCode.

wDraw - Draw Window Frame

Param:

wDrawFrame	= 0	Draw the window's entire frame.
wInGoAway	= 1	Draw go-away region.
wInZoom	= 2	Draw zoom region.
Bit 31	= 1	Highlight.
	= 0	Unhighlighted.

Your routine should draw in the current grafPort, which will be the Window Manager port. The Window Manager will request this operation only if the window is visible.

Param	
$00000000	The entire window frame should be drawn as an inactive window.
$80000000	The entire window frame should be drawn as an active window.
$00000001	The go-away region should be drawn as unhighlighted.
$80000001	The go-away region should be drawn as highlighted.
$00000002	The zoom region should be drawn as unhighlighted.
$80000002	The zoom region should be drawn as highlighted.

wHit - Find What Region a Point Is In

Param equals the point to check. The vertical coordinate is in the low-order WORD and the horizontal coordinate in the high-order WORD. The Window Manager will request this operation only if the window is visible. Your routine should determine where the point is in your window and then return:

wNoHit	= 0	Not on the window at all.
wInContent	= 19	In window's content region.
wInDrag	= 20	In window's drag (title bar) region.
wInGrow	= 21	In window's grow (size box) region.
wInGoAway	= 22	In window's go-away (close box) region.
wInZoom	= 23	In window's zoom (zoom box) region.
wInInfo	= 24	In window's information bar.
wInFrame	= 27	In window, but not any of the above areas.

Usually, wNoHit means the given point isn't anywhere within the window, but this is not necessarily so.

wCalcRgns - Calculate Window's Regions

Your routine should calculate the window's entire region and its content region based on the current grafPort's portRect. The Window Manager will request this operation only if the window is visible. When you calculate regions for your window, do not alter the clipRgn or visRgn of the window's grafPort. The Window Manager and QuickDraw take care of this for you. Altering the clipRgn or visRgn may result in damage to other windows.

wNew - Initialization

After initializing fields as appropriate when creating a new window, the Window Manager sends the message wNew to your routine. This gives your routine a chance to perform any initialization it may require. For example, because the structure of the window record is not documented you made want to allocate your own record structure, initialize it, and store its pointer via SetWRefCon.

wDispose - Remove Window

The Window Manager's CloseWindow and DisposeWindow procedures send this message so your routine can carry out any additional actions required when disposing of the window. The routine might, for example, release space that was allocated by the initialize routine.

wGrow - Draw the Outline of the Window

Param is a pointer to a RECT (rectangle). Your routine should draw an outline image of your window that would fit the given rectangle. The Window Manager requests this operation repeatedly as the user drags inside the grow region. Your routine should use the grafPort's current pen pattern and pen mode, which are set up so one call will draw the outline and next will erase it (XOR mode).

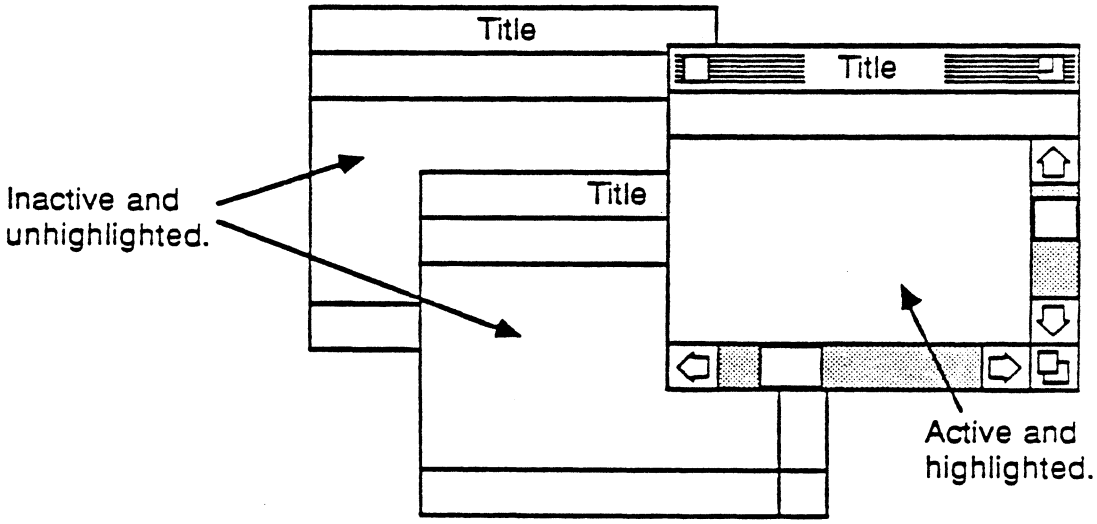
Definitions

highlighted — The frame of the window is drawn in full detail. Generally the front most window is the only highlighted window on the screen. However, any window could be highlighted, even all the windows.

unhighlighted — Opposite of highlighted. Generally all the windows behind the front most window are unhighlighted. However, even the front most window can be unhighlighted.

active — In this document it only means the front most window on the screen. But generally it also means it is highlighted. This should also be the window your application acts on when the user types, gives commands, or whatever is appropriate to the application.

inactive — Any windows behind the front most (active) window. Generally these window will be unhighlighted.



window list or list — An internal linked list of all the window records created by NewWindow and not removed by a CloseWindow or DisposeWindow call. The first visible window in this list is the active window.

top window or top — The first window in the window list. However, the top window is not the active window unless it is visible.

bottom window or bottom — The last window in the window list.

Cortland Window Manager
Appendix A
Window Calls

WINDOW MANAGER ROUTINES

Initialization and Termination

WindBootInit

input: None.

output: None.

Called only by SetTSPtr.

WindStartup

input: userID:WORD - user's ID that Window Manager can use.

output: None

WindStartup initializes the Window Manager. Calls the Event Manager for zero page to use, clears the window list, and sets the default desktop pattern and color. It creates the Window Manager port; you can get a pointer to this port with the GetWMgrPort procedure. The desktop is the entire screen until the Menu Manager, if used, subtracts any area needed for a system menu bar. Call this procedure once before all other Window Manager routines. WindStartup does not draw the desktop, see Refresh.

WindShutDown

input: None.

output: None.

Closes all windows and frees any memory allocated by the Window Manager.

WindVersion

input: None.

output: wVersion:WORD - Window Manager's version number.

WindReset

input: None.

output: None.

WindStatus

input: None.

output: status:WORD -

Status is TRUE if the Window Manager is initialized, else FALSE.

WNewRes

input: None.

output: None.

Called after the screen resolution has been changed. The Window Manager will close its port and open a new one (in the new resolution). Then the screen is not redrawn by the Window Manager in the new resolution. Call **Refresh** when all resolution changes are done, such as desktop pattern and window colors.

Desktop

inputs: Operation:WORD - operation to perform/
Param:LONG - parameter needed for operation.

output: RetParam:LONG - possible return parameter.

Possible Operation numbers:

FromDesk	= 0	Subtract region from desktop region.
ToDesk	= 1	Add region to desktop region.
GetDesktop	= 2	Return handle of desktop region.
SetDesktop	= 3	Set handle of dektop region.
GetDeskPat	= 4	Return current desktop pattern.
SetDeskPat	= 5	Set new desktop pattern.
GetVisDesktop	= 6	Return desktop, less any windows.

Expected inputs and outputs:

Operation	= FromDesk
Param	= Handle of region to be subtracted from desktop region.
RetParam	= Not used (not even necessary to push room on stack).

Operation	= ToDesk
Param	= Handle of region to be added to desktop region.
RetParam	= Not used (not even necessary to push room on stack).

Operation	= GetDesktop
Param	= Not used.
RetParam	= Handle of desktop region.

Operation	= SetDesktop
Param	= Handle of new desktop region.
RetParam	= Same as Param.

Operation = GetDeskPat
Param = Not used.
RetParam = Current desktop pattern where:

$00xxxxxx Where xxxxxx is the address of your routine that will be called to draw the desktop. There are no inputs or outputs,the current port will be the Window Manager's, and the clipping region will be set to the area needing to be drawn. Your routine should exit via a RTL.

Warning: The current direct page and data bank is not defined on entry to your routine. If you need to reference you own direct page you will have to save the original and switch to yours. The same is true with the data bank, except here you can use long addressing. When you exit your routine the direct page and data bank must be the same as it was on entry.

$80xxxxxx Where xxxxxx is the address of the pattern to be used for the desktop.

$4000xxxx The default desktop pattern where xxxx is:

- 00xx = solid desktop pattern.
- 01xx = dithered desktop pattern.
- 02xx = horizontal stripped desktop pattern.
- xxNx = N is the pattern's foreground color.
- xxxN = N is the pattern's background color.

Operation = SetDeskPat
Param = New desktop pattern (see GetDeskPat for definition).
RetParam = Not used (not even necessary to push room on stack).
Desktop is redrawn with new pattern.

Operation = GetVisDesktop
Param = Handle of region that will be set to the visible desktop.
RetParam = Not used (not even necessary to push room on stack).
The desktop region is copied into the given region, and all visible windows are subtracted from it.

NewWindow

input: paramList:LONG - pointer to a parameter list.

output: theWindow:LONG - pointer to window port, zero if error.

Possible errors:

1 = incorrect parameter list length.
2 = unable to locate memory for window record.

NewWindow creates a window as specified by its parameters, adds it to the window list, and returns a pointer to the new window's port. It allocates space for the structure and content regions of the window and asks the window definition function to calculate those regions.

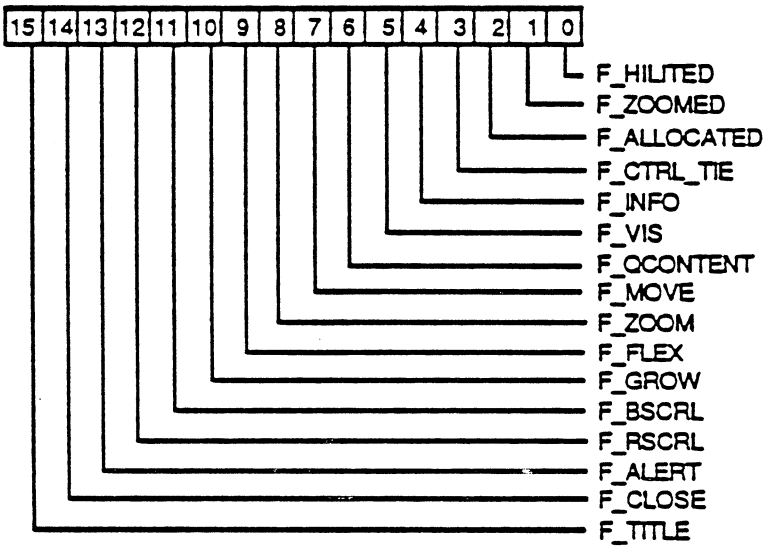
The parameter list is:

param_length	WORD	Number of bytes in parameter table.
wFrame	WORD	Bit vector that descibes the window.
wTitle	LONG	Pointer to window's title.
wRefCon	LONG	Reserved for application's use only.
wZoom	RECT	Size and position of content when zoomed.
wColor	LONG	Pointer to window's color table.
wYOrigin	WORD	Content's vertical origin.
wXOrigin	WORD	Content's horizontal origin.
wDataH	WORD	Height of entire document.
wDataW	WORD	Width of entire document.
wMaxH	WORD	Maximum height of content allowed by GrowWindow.
wMaxW	WORD	Maximum width of content allowed by GrowWindow.
wScrollVer	WORD	Number of pixels to scroll content vertically for arrows.
wScrollHor	WORD	Number of pixels to scroll content horizontally for arrows.
wPageVer	WORD	Number of pixels to scroll content vertically for page.
wPageHor	WORD	Number of pixels to scroll content horizontally for page.
wInfoRefCon	LONG	Value passed to information bar draw routine.
wInfoHeight	WORD	Height of information bar.
wFrameDefProc	LONG	Address of standard window definition procedure.
wInfoDefProc	LONG	Address of routine that draw's the information bar interior.
wContDefProc	LONG	Address of routine that draw's the content region interior.
wPosition	RECT	Window's starting position and size.
wPlane	LONG	Window's starting plane.
wStorage	LONG	Address of memory to use for window record.

Each parameter is covered in more detail below.

param_length — Total number of bytes in parameter table, including param_length. Use labels in code to come up with the value (that's why I don't give it here). The value is used mainly for error checking. Most errors with NewWindow occur because of typing errors when creating the parameter list. The problem can be compounded further by the assembler or complier skipping field because of typing errors but not generating an error.

wFrame — Window frame type:



F_HILITED — 1 = window is highlited, 0 = not highlighted. This flag will be set by NewWindow, so whatever you pass will be ignored.

F_ZOOMED — 1 = window is currently in a zoomed state, 0 = window is not zoomed. This flag is not used is F_ZOOM is zero.

F_ALLOCATED — 1 = window record was allocated by the NewWindow, 0 = window record was not allocated by the NewWindow. If this flag is set when CloseWindow is called, the window record will be freed.

F_CTRL_TIE — 1 = the state of the window's controls is not tied to the window's state. 0 = when the window is inactive (unhighlighted), its controls are also considered inactive without regard for the active state of the control.

F_INFO — 1 = window has an information bar as part of the window's frame, 0 = no information bar.

F_VIS — 1 = window is visible, 0 = window is invisible.

F_QCONTENT — 1 = if there is a button down event inside an inactive window's content, the window will be selected and a wInContent message will be returned by TaskMaster. This feature is use if you would like to act on any button down in the content, even if it was also used to activate the window.

If this bit is zero **TaskMaster** will act in the same way, execpt it will return an inNull message. This feature is use if you would like to button down in an inactive content to activate the window and then not use the same button down event again.

F_MOVE — 1 = window can be dragged by its title bar, 0 = the window's title bar is not considered a drag region and can not, therefore, be moved.

F_ZOOM — 1 = window has a zoom box in its title bar, 0 = window does not have a zoom box in its title bar. The window must have a title bar in order to have a zoom box.

F_FLEX — 1 = data height and width is flexible, which means that **GrowWindow** and **ZoomWindow** will not change the window's origin as needed. See Origin Movement for more information.

F_GROW — 1 = window has a grow box, 0 = window does not have a grow box.

F_BSCRL — 1 = window has a bottom (horizontal) scroll bar as part of the window frame, 0 = no bottom scroll bar.

F_RSCRL — 1 = window has a right (vertical) scroll bar as part of the window frame, 0 = no right scroll bar.

F_ALERT — 1 = alert type window frame. This is used by the Dialog Manager to draw an alert window. F_COSE, F_TITLE, F_RSCRL, F_BSCRL, F_GROW, F_FLEX, F_ZOOM and F_INFO should all be set to zero.

F_CLOSE — 1 = window has a close box in its title bar, 0 = window does not have a close box in its title bar. The window must have a title bar in order to have a close box.

F_TITLE — 1 = window has a title bar as part of the window's frame, 0 = no title bar.

Warning: If F_GROW is set, F_BSCRL or F_RSCRL must also be set. That is to say, to have a window frame grow box, you must have at least one window frame scroll bar.

wTitle	Pointer to title of window. If window does not have a title bar this value can be zero. The first byte in the string should be the length of the string, followed byte the ASCII characters of the title.
wRefCon	Application defined reference value. This value is reserved for the application's use only, and can be any value desired.

wZoom	Rectangle of the content region when the window is zoomed. If the bottom side of the rectangle is zero, a default RECT will be used. The default will be set so that the window will use the entire screen.
wColor	Pointer to window's color table. This is the color table used to draw the window's frame. Zero to use the default color table.
wYOrigin	Vertical offset of content region. This value is the vertical value passed to SetOrgin when TaskMaster is used to draw inside the content region. It is also used to compute the right (or vertical) scroll bar. Zero if not using window frame scroll bars.
wXOrigin	Horizontal offset of content region. This value is the horizontal value passed to SetOrgin when TaskMaster is used to draw inside the content region. It is also used to compute the bottom (or horizontal) scroll bar. Zero if not using window frame scroll bars.
wDataH	Height of entire data area. Used to compute the right scroll bar. Zero if not using window frame scroll bars.
wDataW	Width of entire data area. Used to compute the bottom scroll bar. Zero if not using window frame scroll bars.
wMaxH	Maximum content height allowed when growing the window. This value is passed to GrowWindow when called by TaskMaster. If set to zero, a default value will be used, so that the window will take up the height of the desktop. Zero if not using window frame grow box.
wMaxW	Maximum content width allowed when growing the window. This value is passed to GrowWindow when called by TaskMaster. If set to zero, a default value will be used, so that the window will take up the width of the desktop. Zero if not using window frame grow box.
wScrollVer	Number of pixels to scroll the content region when the up or down arrows are selected in the right scroll bar. Used only if the scroll bar is part of the frame and TaskMaster is used. Zero if not using window frame scroll bars.
wScrollHor	Number of pixels to scroll the content region when the left or right arrows are selected in the bottom scroll bar. Used only if the scroll bar is part of the frame and TaskMaster is used. Zero if not using window frame scroll bars.
wPageVer	Number of pixels to scroll the content region when the up or down page regions are selected in the right scroll bar. Used only if the scroll bar is part of the frame and TaskMaster is used. Zero will default to whatever the content region's height is at the time less 10.
wPageHor	Number of pixels to scroll the content region when the left or right page regions are selected in the bottom scroll bar. Used only if the scroll bar is part of the frame and TaskMaster is used. Zero will default to whatever the content region's width is at the time less 10.

wInfoRefCon — Value passed to Information Bar draw routine. The value can be anything the application would like, such as a pointer to a string to be printed in the information bar. Zero if not using window frame information bar.

wInfoHeight — Height of the information bar, if bit #4 of wFrame is set.

wFrameDefProc — Pointer to window's definition procedure. Zero for a standard document window.

wInfoDefProc — Address of routine that will be called to draw in the information bar. Zero if not using window frame information bar.

wContDefProc — Address of routine that will be called to draw the window's content region. If you are using window frame scroll bars this value must be set. If you are not using window frame scroll bars this value can be zero. However, if you are not using window frame scroll bars, but you would like **TaskMaster** to handle update events, set this value. The routine will be called when the content region needs to be drawn. On entry, the current port will be the window's, the visible region region will be set to the update area, and the origin set. There are no input or output parameters. Exit the routine via RTL.

wPosition — A rectangle given in global coordinates, determines the window's size and location, and becomes the portRect of the window's grafPort; note, however, that the portRect is in local coordinates. **NewWindow** sets the top left corner of the portRect to (0,0). For the standard types of windows, this RECT defines the content region of the window.

wPlane — Pointer to window port this window should appear behind. Zero for bottom most, $FFFFFFFF for top most.

wStorage — Address of memory to use for window's record. If set to zero, the record will be allocated. Because window records are not completely defined, the size needed for a window record is unknow. Therefore, you must allow 325 bytes for a window record. Actually, it is best to have the record allocated by the Window Manager. Being able to use your own memory for a window record is provided for in case you need to put up a window to say there is no memory left, and therefore the Window Manager could not allocate one.

Note: The bit map, pen pattern, and other characteristics of the window's grafPort are the same as the default values set by the **OpenPort** procedure in QuickDraw. (**NewWindow** actually calls **OpenPort** to initialize the window's grafPort.) Note, however, that the coordinates of the grafPort's portBits.bounds and visRgn are changed along with its portRect.

NewWindow also sets the window class in the window record to indicate that the window was created directly by the application.

CloseWindow

input: theWindow:LONG - pointer to window's port.

output: None.

CloseWindow removes the given window from the screen and deletes it from the window list. It releases the memory occupied by all data structures associated with the window, including the memory taken up by the window record if it was allocated by NewWindow. Call this procedure when you're done with a window.

Any update events for the window are discarded. If the window was the frontmost window and there was another window behind it, the latter window is highlighted and an appropriate activate event is generated.

Warning: If you allocated memory yourself and stored a handle to it in the refCon field, CloseWindow won't know about it—you must release the memory before calling CloseWindow.

Window Record and Global Access

GetWMgrPort

input: None.

output: wPort:LONG - pointer to Window Manager's port.

SetWMgrIcons

input: NewFont:LONG - handle of new icon font to use, negative to not replace font.

output: OldFont:LONG - handle of icon font before replacement, if any.

See WINDOW MANAGER ICON FONT for more information about the font.

SetWRefCon

inputs: refCon:LONG - reserved LONG for application's use
theWindow:LONG - pointer to window's port.

output: None.

SetWRefCon is used to set a LONG value that is inside the window record and is reserved for the application's use.

GetWRefCon

input: theWindow:LONG - pointer to window's port.

output: refCon:LONG - reserved LONG for application's use

GetWRefCon is used to retrieve a LONG value from a window's record that was passed to either **NewWindow** or **SetWRefCon** by the application sometime before this call.

SetWTitle

inputs: title:LONG - pointer to string for new title.
theWindow:LONG - pointer to window's port.

output: None.

Updates window's record with new title pointer

GetWTitle

input: theWindow:LONG - pointer to window's port.

output: title:LONG - pointer to string of window's title.

SetFrameColor

inputs: newColor:LONG - pointer to 8 word pattern/color table, zero for system.
theWindow:LONG - pointer to window's port, zero to set default.

output: None.

See WINDOW FRAME COLORS AND PATTERNS for a definition of the color table. Does not redraw the window. Do a HideWindow and ShowWindow before and after this call to redraw the window in its new colors.

If newColor is zero, the pointer to the system default color table will be used. If theWindow is zero, the default window color table will be set. To understand defaults, system, and all this, it necessary to understand how the Window Manager finds a color table to use for drawing. First, a field in the window record is checked for a pointer to a color table. The field is zero after allocated by **NewWindow**, and remains zero until a **SetFrameColor**. If a pointer is found in the window's record, that is the table used. If a zero is found, the default table is used. Now comes the tricky part, the default table starts out as the system table, but can be changed by **SetFrameColor** when theWindow is zero.

GetFrameColor

inputs: newColor:LONG - pointer to 8 word table that will be set with the color table.
theWindow:LONG - pointer to window's port.

output: None.

See WINDOW FRAME COLORS AND PATTERNS for a definition of the color table.

FrontWindow

input: None.

output: theWindow:LONG - pointer to the active window's port.

FrontWindow returns a pointer to first visible window in the window list (that is, the active window). If there are no visible windows, it returns zero.

GetNextWindow

input: theWindow:LONG - pointer to window's port.

output: NextWindow:LONG - pointer to next window's port in list, zero is last.

GetNextWindow returns a pointer the next window after theWindow in the window list, or zero if theWindow is the last window in the window list.

GetWKind

input: theWindow:LONG - pointer to window's port.

output: WindowKind:WORD - TRUE if system window, FALSE if application window.

GetWKind returns the kind of window theWindow is.

GetWFrame

input: theWindow:LONG - pointer to window's port.

output: wFlag:WORD - bit vector of window's frame type.

GetWFrame returns the same type of bit vector passed to NewWindow. See NewWindow for the definition of the bits of wFlag.

SetWFrame

input: wFlag:WORD - bit vector of window's frame type.
theWindow:LONG - pointer to window's port.

output: None.

SetWFrame sets the same type of bit vector passed to NewWindow. See NewWindow for the definition of the bits of wFlag. The window frame is not redrawn.

GetStructRgn

input: theWindow:LONG - pointer to window's port.

output: WStructRgn:LONG - handle of window's structure region.

See WINDOW REGIONS for a definition of what the structure region is.

GetContRgn

input: theWindow:LONG - pointer to window's port.

output: WContRgn:LONG - handle of window's content region.

See WINDOW REGIONS for a definition of what the content region is.

GetUpdateRgn

input: theWindow:LONG - pointer to window's port.

output: WUpdateRgn:LONG - handle of window's structure region.

See **BeginUpdate** for an explaination of how the update region is used.

GetDefProc

input: theWindow:LONG - pointer to window's port.

output: WDefProc:LONG - pointer theWindow's definition procedure.

GetDefProc returns the address of the routine that is called to draw, hit test, and otherwise define, a window's frame and behavior.

SetDefProc

input: WDefProc:LONG - pointer theWindow's definition procedure.
theWindow:LONG - pointer to window's port.

output: None.

SetDefProc sets the address of the routine that is called to draw, hit test, and otherwise define, a window's frame and behavior. See DEFINING YOUR OWN WINDOWS for an explaination of what a definition procedure does.

GetWControls

input: theWindow:LONG - pointer to window's port.

output: ControlList:LONG - address of first control in window's control list, zero = none.

GetWControl returns the address of the first control in the window's control list. The window's control list is the list of controls created by the application with calls to NewControl in the Control Manager. The window's control list is separate from the window frame's control list, explained in GetFControl.

GetFullRect

input: theWindow:LONG - pointer to window's port.

output: wFullSize:LONG - pointer to RECT to be used as content's zoomed size.

If the zoom flag is set in the frame flag, see GetWFrame, then wFullSize will equal theWindow's last size and position. Otherwise, wFullSize will equal the size and position of theWindow's content region (port) the next time the window is zoomed via a call to ZoomWindow.

SetFullRect

input: wFullSize:LONG - pointer to RECT to be used as content's zoomed size.
theWindow:LONG - pointer to window's port.

output: None.

If the zoom flag is set in the frame flag, see GetWFrame, then wFullSize will equal theWindow's last size and position. Otherwise, wFullSize will equal the size and position of theWindow's content region (port) the next time the window is zoomed via a call to ZoomWindow.

GetSysWFlag

input: theWindow:LONG - pointer to window.

output: sysFlag:WORD - TRUE if system window, FALSE if application window.

SetSysWindow

input: theWindow:LONG - pointer to window.

output: None.

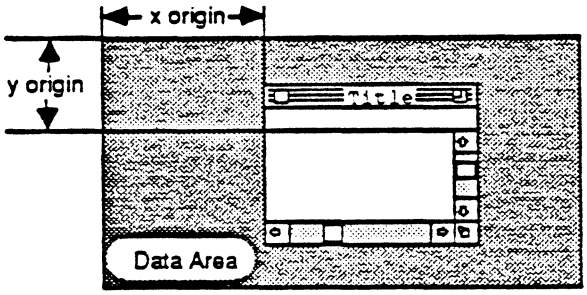
TheWindow is marked as a system window.

GetCOrigin

inputs: theWindow:LONG - pointer to window's port.

output: LONG - low WORD = y origin, high WORD = x origin.

These values are used by TaskMaster for setting the origin of the window's port when handling an update event. The values are also used to compute scroll bars in the window frame.



SetCOrigin

inputs: xOrigin:WORD - content region's horizontal offset into the data area.
yOrigin:WORD - content region's vertical offset into the data area.
theWindow:LONG - pointer to window's port.

output: None.

See GetCOrigin for a description of origins. Setting these values will not generate any update event, although the entire content will probably needed to be redrawn.

SetOrgnMask

inputs: originMask:WORD - mask used to put horizontal origin on a grid.
theWindow:LONG - pointer to window's port.

output: None.

SetOrgnMask is useful when you are using a scrollable window in 640 mode with dithered colors. The video hardware of the Cortland is such that different pixel position get their color from different color tables. By using the effect it is possible to produce many more colors than the two bits per pixels might suggest. However, the pixels are then horizontally position dependent to keep the same color. Scrolling windows can change the color by putting the pixels in the wrong horizontal position. That's where SetOrgnMask comes in. OriginMask will be ANDed by TaskMaster with any new horizontal origin that is created to force the origin to certain boundaries. The default is $FFFF, single pixel.

StartDrawing

input: theWindow:LONG - pointer to window's port.

output: None.

StartDrawing can be used for drawing in a window's content region outside of update events. **StartDrawing** will make the window the current port, and set its origin. After the call, any drawing, outside of update events, will occur inside theWindow's content and in the proper coordinate system.

Note: **StartDrawing** is only of use with standard document window's with frame scroll bars. Otherwise, only a **SetPort** would be needed to make the proper port current.

GetDataSize

inputs: theWindow:LONG - pointer to window's port.

output: dataSize:LONG - low WORD is the height, high WORD is the width.

The height and width of the data area is returned. The data area is the total amount of data that can be viewed in a window, either through resizing or scrolling.

SetDataSize

inputs: dataWidth:WORD - width of data area.
dataHeight:WORD - height of data area.
theWindow:LONG - pointer to window's port.

output: None.

See **GetDataSize**. Setting these values will not change the scroll bars or generate update events.

GetMaxGrow

inputs: theWindow:LONG - pointer to window's port.

output: maxGrow:LONG - low WORD is the max height, high WORD is the max width.

These values are pasted to **GrowWindow** by **TaskMaster**. The content region will not be allowed to be sized to exceed these values.

SetMaxGrow

inputs: maxWidth:WORD - maximum content width allowed when resizing.
maxHeight:WORD - maximum content height allowed when resizing.
theWindow:LONG - pointer to window's port.

output: None.

See GetMaxGrow.

GetScroll

inputs: theWindow:LONG - pointer to window's port.

output: scroll:LONG - low WORD is the vertical amount, high WORD the horizontal.

Returns the number of pixels that **TaskMaster** will scroll the content region when the user selects the arrows on window frame scroll bars.

SetScroll

inputs: hScroll:WORD - number of pixels to scroll horizontally.
vScroll:WORD - number of pixels to scroll vertically.
theWindow:LONG - pointer to window's port.

output: None.

See GetScroll.

GetPage

inputs: theWindow:LONG - pointer to window's port.

output: page:LONG - low WORD is the vertical amount, high WORD the horizontal.

Returns the number of pixels that TaskMaster will scroll the content region when the user selects the page regions on window frame scroll bars.

SetPage

inputs: hPage:WORD - number of pixels to page horizontally.
vPage:WORD - number of pixels to page vertically.
theWindow:LONG - pointer to window's port.

output: None.

See GetPage.

GetCDraw

inputs: theWindow:LONG - pointer to window's port.

output: contDraw:LONG - address of routine that is called to draw the content region.

TaskMaster will call this routine when it gets an update event for that window. See CONTENT DRAW ROUTINE for more infomation about the draw routine.

SetCDraw

inputs: contDraw:LONG - address of routine to draw content region.
theWindow:LONG - pointer to window's port.

output: None.

See GetCDraw.

Information Bar Routines

GetInfoDraw

inputs: theWindow:LONG - pointer to window's port.

output: InfoDraw:LONG - address of TheWindow's infomation bar definition procedure.

The standard window definition procedure will call TheWindow's information bar definition procedure whenever the window's frame needs to be drawn, if the window has an infomation bar. See INFORMATION BAR DEFINITION PROCEDURE for more infomation.

SetInfoDraw

inputs: InfoDraw:LONG - address of TheWindow's infomation bar definition procedure.
theWindow:LONG - pointer to window's port.

output: None.

See GetInfoDraw.

GetInfoRefCon

input: theWindow:LONG - pointer to window's port.

output: InfoRefCon:LONG - value associated to the information bar.

GetInfoRefCon will return the value that is passed to TheWindow's information bar definition procedure.

SetInfoRefCon

input: InfoRefCon:LONG - value passed to the information bar defintion procedure.
theWindow:LONG - pointer to window's port.

output: None.

SetInfoRefCon will set the value that is passed to TheWindow's information bar definition procedure. See INFORMATION BAR DEFINITION PROCEDURE.

GetRectInfo

input: InfoRect:LONG - pointer to destination RECT where the RECT will be stored.
theWindow:LONG - pointer to window's port.

output: None.

InfoRect will be set to the coordinates of the information bar rectangle (excluding the outline frame). If there is no information bar in the window the coordinates of InfoRect will all be zero. The coordinate system will be local to the window's frame, that is, 0,0 will be the upper left corner of the window. The coordinates can be used to set the position of objects that will be drawn in the information bar.

StartInfoDrawing

input: InfoRect:LONG - pointer to destination RECT where the RECT will be stored.
theWindow:LONG - pointer to window's port.

output: None.

Call **StartInfoDrawing** anytime you need to draw or hit test outside of your information bar definition procedure.

InfoRect will be set to the coordinates of the information bar rectangle (excluding the outline frame). If there is no information bar in the window the coordinates of InfoRect will all be zero. The coordinate system will be local to the window's frame, that is, 0,0 will be the upper left corner of the window and the current port will be the Window Manager's.

It is ok to set the clip region after a **StartInfoDrawing** and before **EndInfoDrawing** calls, this is not true from within the information bar definition procedure.

Warning: You must call **EndInfoDrawing** when you have completed interaction with the information bar and before you make any other calls to the Window Manager.

EndInfoDrawing

input: None.

output: None.

Call **EndInfoDrawing** after **StartInfoDrawing** and before any other calls to the Window Manager. **EndInfoDrawing** will put the Window Manager back into a global cooridnate system.

Warning: Calling any Window Manager function between a **StartInfoDrawing** and **EndInfoDrawing** may result in window system failure.

Window Shuffling

SelectWindow

input: theWindow:LONG - pointer to window's port.

output: None.

SelectWindow makes theWindow the active window as follows: It unhighlights the previously active window, brings theWindow in front of all other windows, highlights theWindow, and generates the appropriate activate events. Call this procedure if you are not using **TaskMaster** and there's a mouse-down event in the content region of an inactive window.

HideWindow

input: theWindow:LONG - pointer to window's port.

ouput: None.

HideWindow makes theWindow invisible. If theWindow is the frontmost window and there's a window behind it, **HideWindow** also unhighlights theWindow, brings the window behind it to the front, highlights that window, and generates appropriate activate events. If theWindow is already invisible, **HideWindow** has no no effect.

ShowWindow

inputs: theWindow:LONG - pointer to window's port.

output: None.

Makes theWindow visible and draws it if it was invisible. It does not change the front-to-back ordering of the windows. Remember that if you previously hid the frontmost window with **HideWindow**, **HideWindow** will have brought the window behind it to the front; so if you then do a **ShowWindow** of the window you hid, it will no longer be frontmost. If theWindow is already visible, **ShowWindow** has no effect.

ShowHide

input: showFlag:WORD - TRUE to show, FALSE to hide.
theWindow:LONG - pointer to the window's port.
ouput: None.

If showFlag is TRUE, **ShowHide** makes theWindow visible if it's not already visible and has no efect if it is already visible. If showFlag is FALSE, **ShowHide** makes theWindow invisible if it's not already invisible and has no effect if it is already invisible. Unlike **HideWindow** and **ShowWindow**, **ShowHide** never changes the highlighting or front-to-back ordering of windows or generates activate events.

Warning: Use this procedure carefully, and only in special circumstances where you need more control than allowed by **ShowWindow** and **HideWindow**. You could end up with an active window that isn't highlighted.

BringToFront

input: theWindow:LONG - pointer to window's port.

output: None.

BringToFront brings theWindow to the front of all other windows and redraws the windows as necessary, but does not do any highlighting or unhighlighting. Normally you won't have to call this procedure, since you should call **SelectWindow** to make a window active, and SelectWindow takes care of bringing the window to the front. If you do call **BringToFront**, however, remember to call **HiliteWindow** to make the necessary highlighting changes.

SendBehind

inputs: behindWindow:LONG - pointer to window port, -1 to top or -2 to bottom.
theWindow:LONG - pointer to window's port.

output: None.

SendBehind sends theWindow behind behindWindow, redrawing any exposed windows. If behindWindow is -2 ($FFFFFFFE), it sends theWindow behind all other windows. If behindWindow is -1 ($FFFFFFFF), it puts theWindow in front of all other windows (same as SelectWindow). If theWindow is the active window, it unhighlights theWindow, highlights the new active window, and generates the appropriate activate events.

Window Drawing

HiliteWindow

input: fHilite:WORD - TRUE to highlight window frame, FALSE to unhighlight.
theWindow:LONG - pointer to window's port.

output: None.

If fHilite is TRUE, this procedure highlights theWindow. If fHilite is FALSE, **HiliteWindow** unhighlights theWindow. The exact way a window is highlighted and unhighlighted depends on its window definition procedure.

Normally you won't have to call this procedure, since you should call **SelectWindow** to make a window active, and **SelectWindow** takes care of the necessary highlighting changes. Highlighting a window that isn't the active window should never be done.

Refresh

input: ClobberedRect:LONG - pointer to RECT needing redraw, zero for entire screen.

output: None.

Redraws the entire desktop and all the windows inside of ClobberedRect, or entire screen if the pointer is zero. Useful when the entire screen was clobblered by some application specific, non-Window Manager, operation.

User Interaction

FindWindow

inputs: whichWindow:LONG - address of where to store pointer of window.
pointX - x coordinate to check (global).
pointY - y coordinate to check (global).

outputs: Location:WORD:

wNoHit	= $0000	Not on the window at all.
wInDesk	= $0010	On the desktop area.
wInMenuBar	= $0011	On the system menu bar.
wInContent	= $0013	In window's content region.
wInDrag	= $0014	In window's drag (title bar) region.
wInGrow	= $0015	In window's grow (size box) region.
wInGoAway	= $0016	In window's go-away (close box) region.
wInZoom	= $0017	In window's zoom (zoom box) region.
wInInfo	= $0018	In window's information bar.
wInFrame	= $001B	In window, but not any of the above areas.
wInSysWindow	= $8xxx	In a system window, lower part is one of the above.

When a mouse-down event occurs, the application should, if not using TaskMaster, call **FindWindow** with pointY,pointX equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). **FindWindow** tells which part of which window, if any, the mouse button was pressed in. If it was pressed in a window, the pointer to the window is stored in your variable at the address passed to **FindWindow** in theWindow parameter.

DragWindow

inputs: grid:WORD - drag resolution, zero for default.
startX - starting x coordinate of cursor (global).
startY - starting y coordinate of cursor (global).
grace:WORD - grace buffer around Bounds.
BoundsRect:LONG - pointer to RECT to use as cursor boundary, zero for default.
theWindow:LONG - pointer to window's port.

output: None.

When there is a mouse-down event in the drag region of theWindow, and TaskMaster is not being used, the application should call **DragWindow** with startY,startX equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). **DragWindow** pulls a dotted outline of theWindow around, following the movements of the mouse until the button is released. When the mouse button is released, **DragWindow** call **MoveWindow** to move theWindow to the location to which it was dragged. The window will be dragged and moved in its current plane.

grid — Allowed horizontal resolution movement. If grid is one, the window can be positioned at any horizontal position. If grid is two, the window can only be moved a multiple of 2 pixels horizontally. If grid is four, the window can only be moved a multiple of 4 pixels horizontally. The only allowed values are; 1, 2, 4, 8, 16, 32, 64, 128... The grid parameter is provided to speed up window moves by eliminating the need for bit shifting, if the grid value is the correct value. If grid is passed as zero, a default value will be used. The defaults are; 4 for 320 mode and 8 for 640 mode.

startY,startX — The point where the mouse button was pressed, in global coordinates, as stored in the where field of the event record. This point is used with the tracked cursor position to compute the movement delta.

grace — Grace is the distance, in pixels, that you will allow the cursor to move away from BoundsRect before the dragged outline should be snapped back to its starting position. **TaskMaster** uses 8 for this value. The BoundsRect is expanded by the value of grace to compute the slopRect passed to **DragRect**. See **DragRect** for more information.

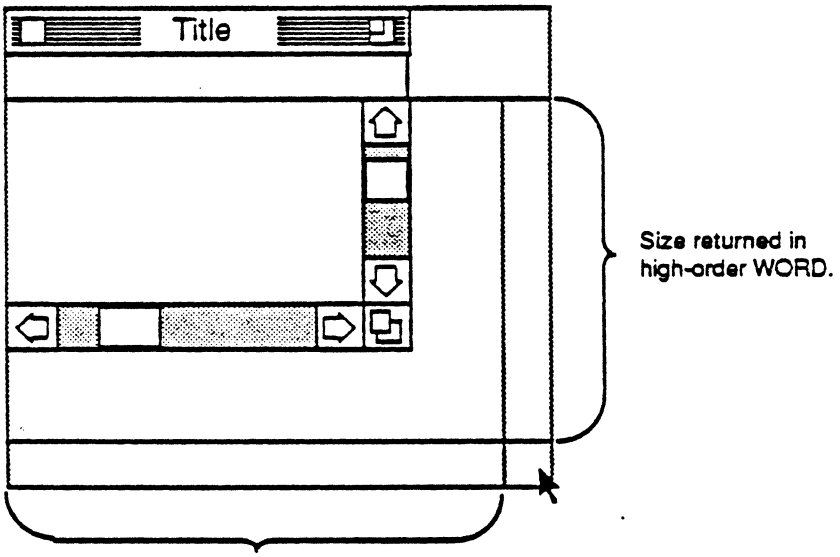
BoundsRect — Pointer to a RECT, in global coordinates, that is passed to **DragRect** as the limitRect parameter. See **DragRect** for more information. If zero is passed for the pointer, the bounds of the desktop, less 4 all around, will be used.

GrowWindow

inputs: minWidth:WORD - minimum width of content region to allow.
minHeight:WORD - minimum height of content region to allow.
startX - starting x coordinate of cursor (global).
startY - starting y coordinate of cursor (global).
theWindow:LONG - pointer to window's port.

output: newSize:LONG - high WORD = new height, low WORD = new width.

When there's a mouse-down event in the grow region of theWindow, the application should call **GrowWindow** with startY,startX equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). **GrowWindow** pulls a grow image of the window around, following the movements of the mouse until the button is released. The grow image for a document window is a dotted outline of the entire window and also the lines delimiting the title bar, size box, and scroll bar areas. The diagram below illustrates this for a document window containing both scroll bars. In general, the grow image is defined in the window definition function and is whatever is appropriate to show that the window's size will change.



The application should subsequently call **SizeWindow** to change the portRect of the window's grafPort to the new one outlined by the grow image. The sizeRect parameter specifies limits, in pixels, on the vertical and horizontal measurements of what will be the new portRect. SizeRect.top is the minimum vertical measurement, sizeRect.left is the minimum horizontal measurement, sizeRect.bottom is the maximum vertical measurement, and sizeRect.right is the maximum horizontal measurement.

GrowWindow returns the actual size for the new portRect as outlined by the grow image when the mouse button is released. The high-order WORD of the LONG is the vertical

measurement in pixels and the low-order WORD is the horizontal measurement. A return value of zero indicates that the size is the same as that of the current portRect.

TrackGoAway

inputs: startX - starting x coordinate of cursor (global).
startY - starting y coordinate of cursor (global).
theWindow:LONG - pointer to window's port.

output: GoAway:WORD - TRUE if go away selected when button released, else FALSE.

When there's a mouse-down event in the go-away region of theWindow, and the application is not using TaskMaster, the application should call **TrackGoAway** with thePT equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). **TrackGoAway** keeps control until the mouse button is released, highlighting the go-away region as long as the mouse location remains inside it, and unhighlighting it when the mouse moves outside it. The exact way a window's go-away region is highlighted depends on its window definition procedure. If the mouse button is released inside the go-away region, **TrackGoAway** unhighlights the go-away region and returns TRUE (the application should then eventually perform a CloseWindow). If the mouse button is released outside the go-away region, **TrackGoAway** returns FALSE (in which case the application should do nothing).

TrackZoom

inputs: startX - starting x coordinate of cursor (global).
startY - starting y coordinate of cursor (global).
theWindow:LONG - pointer to window's port.

output: Zoom:WORD - TRUE if zoom region was selected, else FALSE.

When there's a mouse-down event in the zoom region of theWindow, and the application is not using TaskMaster, the application should call **TrackZoom** with thePT equal to the point where the mouse button was pressed (in global coordinates, as stored in the where field of the event record). **TrackZoom** keeps control until the mouse button is released, highlighting the zoom region as long as the mouse location remains inside it, and unhighlighting it when the mouse moves outside it. The exact way a window's zoom region is highlighted depends on its window definition procedure. If the mouse button is released inside the zoom region, **TrackZoom** unhighlights the zoom region and returns TRUE (the application should then eventually perform a **ZoomWindow**). If the mouse button is released outside the zoom region, **TrackZoom** returns FALSE (in which case the application should do nothing).

TaskMaster

input: EventMask:WORD - used to call GetNextEvent.
TaskRec:LONG - pointer to an extended event record to use.

output: TaskCode:WORD - task code, zero equal no further task to perform.

Possible error: 3 = bits 13-31 are not clear in TaskMask field of TaskRec.

See Constants for TaskCode definitions.

See USING TASKMASTER for more information.

TaskMaster uses TaskRec and EventMask to pass to GetNextEvent. An outline of TaskMaster follows:

Call SystemTask for possible desk accessories.
Call GetNextEvent with a TaskRec and EventMask.
If GetNextEvent returns 'no event' TaskMaster will exit and return inNull.
The message field of the TaskRec is duplicated into the TaskData field.

If event code is key down event:
    If TaskMask bit #0 = 0:
        TaskMaster exits and returns inKey.
    Call MenuKey for the system menu bar with the key from TaskRec.
    If MenuKey returns 'no selection made' TaskMaster exits and returns inKey.
    If TaskMask bit #4 = 0:
        TaskMaster exits and returns wInMenuBar.
    If the item selected has an ID number greater than 255:
        TaskMaster exits and returns wInMenuBar.
    Else the item belongs to a desk accessory and:
        Call OpenNDA to open the desk accessory selected.
        Call HiliteMenu to unhighlight the selected menu.
        TaskMaster exits and returns inNull.

If event code is update event:
    If TaskMask bit #1 = 0:
        TaskMaster exits and returns inUpdate.
    If the window with the update has an update draw routine (see NewWindow):
        Switch to window's port.
        Window's origin is set according to the origin values in its record.
        The window's update draw routine is called (routine in application).
        Window's origin is returned to zero,zero.
        The previous port is restored.
        TaskMaster exits and returns inNull.
    Else TaskMaster is unable to process the event:
        TaskMaster exits and returns inUpdate.

If event code is button down event:
    If TaskMask bit #2 = 0:
        TaskMaster exits and returns inButtDwn.
    Call FindWindow which place any found window pointer in the TaskData field.

    If FindWindow returns wInMenuBar:
        If TaskMask bit #3 = 0:
            Low-order WORD of TaskData field in TaskRec = zero.
            TaskMaster exits and returns wInMenuBar.
        Call MenuSelect.
        If MenuSelect returns 'no selection made':
            TaskMaster exits and returns inNull.
        Else if the item ID > 255 OR TaskMask bit #12 = 0:
            Low-order WORD of TaskData field in TaskRec = selected item's ID.
            High-order WORD of TaskData field in TaskRec = selected menu's ID.
            TaskMaster exits and returns wInMenuBar.
        Else if the item ID < 250
            if TaskMask bit #4 = 0
                Low-order WORD of TaskData field in TaskRec = selected item's ID.
                High-order WORD of TaskData field in TaskRec = selected menu's ID.
                TaskMaster exits and returns wInDeskItem.
            Else
                Call OpenNDA to open the desk accessory selected.
                Call HiliteMenu to unhighlight the selected menu.
                TaskMaster exits and returns inNull.
        Else (item is between 250 and 255)
            If TaskMask bit #12 = 0 or top window is an application window.
                Low-order WORD of TaskData field in TaskRec = selected item's ID.
                High-order WORD of TaskData field in TaskRec = selected menu's ID.
                TaskMaster exits and returns wInSpecial.
            Else if item ID = 255 (Close item).
                Call CloseNDAbyWinPtr with top window pointer (system window).
                TaskMaster exits and returns inNull.
            Else (item must be an Undo, Cut, Copy, Paste or Clear).
                Call SystemEdit.
                If SystemEdit returns FALSE.
                    Low-order WORD of TaskData = selected item's ID.
                    High-order WORD of TaskData = selected menu's ID.
                    TaskMaster exits and returns wInSpecial.
                Else
                    TaskMaster exits and returns inNull.

    Else if FindWindow returns a value that is negative:
        If TaskMask bit #5 = 0:
            TaskData = window pointer returned from FindWindow.
            TaskMaster exits and returns value returned by FindWindow.
        FindWindow found something in a system window.
        Call SystemClick with the window and result from FindWindow.
        NOTE: This is as far as system windows can go in TaskMaster.
        TaskMaster exits and returns inNull.

Else if **FindWindow** returns wInDrag:
    If TaskMask bit #6 = 0:
        TaskData = window pointer returned from **FindWindow**.
        TaskMaster exits and returns wInDrag.
    If the command key is not down and the window is not active:
        Call **SelectWindow** to make the window active.
    Call **DragWindow**.
    TaskMaster exits and returns inNull.

Else if **FindWindow** returns wInContent:
    If TaskMask bit #7 = 0:
        TaskData = window pointer returned from **FindWindow**.
        TaskMaster exits and returns wInContent.
    If the window is not active:
        Call **SelectWindow** to make the window active.
        TaskMaster exits and returns inNull.
    Else:
        TaskData = window pointer returned from FindWindow.
        TaskMaster exits and returns wInContent.

Else if **FindWindow** returns wInGoAway:
    If TaskMask bit #8 = 0:
        TaskData = window pointer returned from **FindWindow**.
        TaskMaster exits and returns wInGoAway.
    Call **TrackGoAway**.
    If **TrackGoAway** returns TRUE:
        TaskData = window pointer returned from **FindWindow**.
        TaskMaster exits and returns wInGoAway.
    TaskMaster exits and returns inNull.

Else if **FindWindow** returns wInZoom:
    If TaskMask bit #9 = 0:
        TaskData = window pointer returned from **FindWindow**.
        TaskMaster exits and returns wInZoom.
    Call **TrackZoom**.
    If **TrackZoom** returns TRUE:
        Call **ZoomWindow**.
    TaskMaster exits and returns inNull.

Else if **FindWindow** returns wInGrow:
    If TaskMask bit #10 = 0:
        TaskData = window pointer returned from **FindWindow**.
        TaskMaster exits and returns wInGrow.
    Call **GrowWindow**.
    Call **SizeWindow** with results from **GrowWindow**.
    TaskMaster exits and returns inNull.

Else if FindWindow returns wInFrame:
    If TaskMask bit #11 = 0:
        TaskData = window pointer returned from FindWindow.
        TaskMaster exits and returns wInFrame.
    If the window is not active:
        Call SelectWindow to make the window active.
        TaskMaster exits and returns inNull.
    Else if button down event occurred in a window frame scroll bars:
        TaskMaster does some unorthodox window and port manipulation.
        Calls TrackControl with an action procedure within TaskMaster.
        NOTE: The window owner of frame scroll bar is the Window Manager's.
        The action procedure in TrackMaster performs scrolling and updates.
        TaskMaster exits and returns inNull.
    Else:
        TaskMaster exits and returns wInFrame.

Else:
    TaskData = window pointer returned from FindWindow.
    TaskMaster exits and returns value returned from FindWindow.

Else:
    TaskMaster exits and returns event code.

Window Sizing and Positioning

MoveWindow

inputs: newX - new x coordinate of content region's upper left corner (global).
newY - new y coordinate of content region's upper left corner (global).
theWindow:LONG - pointer to window's port.

output: None.

MoveWindow moves theWindow to another part of the screen, without affecting its size. The top left corner of the window's portRect is moved to the screen point newY,newX. The local coordinates of the window's top left corner remain the same.

SizeWindow

inputs: newWidth:WORD - new width of window.
newHeight:WORD - new height of window.
theWindow:LONG - pointer to window's port.

output: None.

SizeWindow enlarges or shrinks the portRect of theWindow's grafPort to the width and height specified by newWidth and newHeight, or does nothing if newWidth and newHeight are zero. The window's position on the screen does not change. The new window frame is drawn; if the width of a document window changes, the title is again centered in the title bar, or is truncated if it no longer fits.

ZoomWindow

input: theWindow:LONG - pointer to window's port.

output: None.

ZoomWindow will flip the size and position of theWindow between its current size and position, to its maximum size, passed to **NewWindow**. If called again, before theWindow is moved or resized, theWindow will be resize and positioned to the size and position before the last **ZoomWindow** was performed. When a **SizeWindow** or **MoveWindow** is performed, while a window is zoomed, the last size becomes the new size and position.

Update Region Maintenance

InvalRect

input: badRect:LONG - pointer to RECT to be added to the update region.

output: None.

InvalRect accumulates the given rectangle into the update region of the window whose grafPort is the current port. This tells the Window Manager that the rectangle has changed and must be updated. The rectangle is given in local coordinates and is clipped to the window's content region.

For example, this procedure is useful when you're calling **SizeWindow** for a document window that contains a size box or scroll bars that are not inside the window's frame. Suppose you're going to call **SizeWindow** with fUpdate=TRUE. If the window is enlarged, you'll want not only the newly created part of the content region to be updated, but also the two rectangular areas containing the (former) size box and scroll bars; before calling **SizeWindow**, you can call **InvalRect** twice to accumulate those areas into the update region. In case the window is made smaller, you'll want the new size box and scroll bar areas to be updated, and so can similarly call **InvalRect** for those areas after calling **SizeWindow**. As another example, suppose your application scrolls up text in a document window and wants to show new text added at the bottom of the window. You can cause the added text to be redrawn by accumulating that area into the update region with **InvalRect**.

InvalRgn

input: badRgn:LONG - handle of region to be added to the update region.

output: None.

InvalRgn is the same as **InvalRect** but for a region that has changed rather than a rectangle.

ValidRect

input: goodRect:LONG - pointer to a RECT to be removed from the update region.

output: None.

ValidRect removes goodRect from the update region of the window whose grafPort is the current port. This tells the Window Manager that the application has already drawn the rectangle and to cancel any updates accumulated for that area. The rectangle is clipped to the window's content region and is given in local coordinates. Using **ValidRect** results in better performance and less redundant redrawing in the window.

For example, suppose you've called **SizeWindow** with fUpdate=TRUE for a document window that contains a size box or scroll bars not part of the window frame. Depending on the dimensions of the newly sized window, the new size box and scroll bar areas may or may not have been accumulated into the window's update region. After calling **SizeWindow**, you can redraw the size box or scroll bars immediately and then call **ValidRect** for the areas they occupy in case they were in fact accumulated into the update region; this will avoid redundant drawing.

ValidRgn

input: goodRgn:LONG - handle of a region to be subtracted from the update region.

output: None.

ValidRgn is the same as **ValidRect** but for a region that has been drawn rather than a rectangle.

BeginUpdate

input: theWindow:LONG - pointer to window's port.

output: None.

Call **BeginUpdate** when an update event occurs for theWindow. **BeginUpdate** replaces the visRgn of the window's grafPort with the intersection of the visRgn and the update region and then sets the window's update region to an empty region. You would then usually draw the entire content region, though it suffices to draw only the visRgn; in either case, only the parts of the window that require updating will actually be drawn on the screen. Every call to **BeginUpdate** must be balanced by a call to **EndUpdate**. (See "HOW A WINDOW IS DRAWN".) **BeginUpdate** calls can be nested (that is **BeginUpdate** may be called several times, for several different windows, before **EndUpdate** is called for each window).

EndUpdate

input: theWindow:LONG - pointer to window's port.

output: None.

Call **EndUpdate** to restore the normal visRgn of theWindow's grafPort, which was changed by **BeginUpdate** as described above.

Miscellaneous Routines

PinRect

inputs: theXPt:WORD - the x coordinate of the point to be pinned.
theYPt:WORD - the y coordinate of the point to be pinned.
theRect:LONG - pointer to RECT that is the boundary of the given point.

output: pinnedPt:LONG - point inside theRect nearest to thePt.

PinRect "pins" thePt inside theRect: If thePt is inside theRect, thePt is returned; otherwise, the point associated with the nearest pixel within theRect is returned. (The high-order WORD of the pinnedPt is the vertical coordinate; the low-order WORD is the horizontal coordinate.) More precisely, for theRect (left,top) (right,bottom) and thePt (h,v), PinRect does the following:

- If h < left, it returns left.
- If v < top, it returns top.
- If h > right, it returns right–1.
- If v > bottom, it returns bottom–1.

Note: The 1 is subtracted when thePt is below or to the right of theRect so that a pixel drawn at that point will lie within theRect.

CheckUpdate

input: theEvent:LONG - pointer to an event record.

output: Flag:WORD - TRUE if update event found, else FALSE.

CheckUpdate is called by the Event Manager. From the top to the bottom in the window list, it looks for a visible window that needs updating (that is, whose update region is not empty). If a window with something in its update region is found, an update event for that window is stored in theEevnt and returns TRUE. If it doesn't find such a window, it returns FALSE.

Constants

Bit masks for wframe field of window record and **NewWindow** parameter list:

```
F_HILITED       $0001       Window is highlighted.
F_ZOOMED        $0002       Window is zoomed.
F_ALLOCATED     $0004       Window record was allocated.
F_CTRL_TIE      $0008       Window state tied to controls.
F_INFO          $0010       Window has an information bar.
F_VIS           $0020       Window is visible.
F_MOVE          $0080       Window is movable.
F_ZOOM          $0100       Window is zoomable.
F_GROW          $0400       Window has grow box.
F_BSCROLL       $0800       Window has horizontal scroll bar.
F_RSCROLL       $1000       Window has vertical scroll bar.
F_CLOSE         $4000       Window has a close box.
F_TITLE         $8000       Window has a title bar.
```

Number of bytes in a window record:

```
WIND_SIZE       325         Size of WindRec.
```

Command messages sent to window defintion procedures:

```
wDraw           0           Draw window frame command.
wHit            1           Hit test command.
wCalcRgns       2           Compute regions command.
wNew            3           Initialization command.
wDispose        4           Dispose command.
```

Return vales from **TaskMaster**, **FindWindow**, and definition procedures wHit command:

```
wNoHit          0
wInDesk         16
wInMenuBar      17
wInSysWindow    18
wInContent      19
wInDrag         20
wInGrow         21
wInGoAway       22
wInZoom         23
wInInfo         24
wInSpecial      25
wInDeskItem     26
wInFrame        27
```

Value passed to **SendBehind**:

BOTTOM_MOST	0	To make window bottom.
TOP_MOST	-1	To make window top.
TO_BOTTOM	-2	To send window to bottom.

Data Types

TaskRec (passed to **TaskMaster**):

what	Integer	Same as event record.
message	LongInt	Same as event record.
when	LongInt	Same as event record.
where	LongInt	Same as event record.
modifiers	Integer	Same as event record.
TaskData	LongInt	TaskMaster return value.
TaskMask	Integer	TaskMaster feature mask.

Defined part of window record:

wnext	Pointer	Pointer to next window Record.
wport	Port	Window's port.
wstrucRgn	Handle	Region of frame plus content.
wcontRgn	Handle	Content region.
wupdateRgn	Handle	Update region.
wcontrol	Handle	Window's control list.
wFrameCtrl	Handle	Window frame's control list.
wframe	Integer	Bit flags.

Number of bytes in a window record:

WIND_SIZE	325	Size of WindRec.

Window frame color table:

FrameColor	Integer	Color of window frame.
TitleColor	Integer	Color of title and bar.
TBarColor	Integer	Color/pattern of title bar.
GrowColor	Integer	Color of grow box.
InfoColor	Integer	Color of information bar.

Parameter list passed to NewWindow:

```
param_length    Integer
wFrame          Integer
wTitle          Pointer
wRefCon         LongInt
wZoom           RECT
wColor          Pointer
wYOrigin        Integer
wXOrigin        Integer
wDataH          Integer
wDataW          Integer
wMaxH           Integer
wMaxW           Integer
wScrollVer      Integer
wScrollHor      Integer
wPageVer        Integer
wPageHor        Integer
wInfoRefCon     LongInt
wFrameDefProc   Pointer
wInfoDefProc    Pointer
wContDefProc    Pointer
wPosition       RECT
wPlane          LongInt
wStorage        Pointer
```

Error Codes

ParamLenErr	1	NewWindow	First word of parameter list is the wrong size.
AllocateErr	2	NewWindow	Unable to allocate window record.
TaskMaskErr	3	TaskMaster	Bits 12-15 are not clear in TaskMask field of TaskRec.

INDEX

Documentation Développeurs
Apple Computer France 1987

Document développeur numéro 55

Menu Manager

type d'upgrade de ce ducument : 5
1 Documentation de première catégorie inchangée
2 Documentation de deuxième catégorie mise à jour
3 Documentation de deuxième catégorie inchangée
4 Mise à jour payante de la documentation de première catégorie
5 Mise à jour gratuite de la documentation de première catégorie
6 Nouveautés payantes non vitales
7 Nouveautés gratuites et vitales

Taille : 50 page(s) environ

Domaine : Tool 15

VERSION : 1.0
DATE : 25.09.86

Cortland Menu Manager

Dan Oliver

This ERS corresponds to the Beta release of the Menu Manager, version 1.0. There will be no further changes to the Menu Manager that will compromise applications written for this version. However, document errors will be changed to conform to actual code. Over the next few months I will be fixing bugs and releasing appendixes that detail Menu Manager functions. If you have areas you would like clarified please let me know and I will try to publish an appendix. Send comments to:

Apple Computer, Inc.
20525 Mariani Ave, MS: 22X
Cupertino, CA 95014

ATTN: Dan Oliver

02/18/86 Initial release.

05/08/86 Many parameter changes to accommodate menu speedups. Menu record changes and an alternative method of defining menus, see MENU STRINGS. Additional Mac type calls to help portability. Many unnecessary features that slowed things down have gone away. New calls **CheckItem, SetItemMark, GetItemMark, EnableItem, DisableItem, NewMenu, DisposeMenu, SetMenuID, SetItemID, SetSysBar, GetSysBar,** and **InitPalette.**

06/18/86 Fall Down menus removed. **InitMenus, BootMmgr, MmgrReset, MmgrVersion** names changed. Additional parameter, user ID, passed to **MenuStartup** (formerly **InitMenus**). Direct access to menu record no longer supported. Custom menus being rethought, and not currently complete. **GetMenuPtr** and **GetItemPtr** removed. **InsertMenu** and **InsertItem** are being redesigned, and are not complete. Now using standard error return code, although it is always 'No error'.

07/15/86 Changed the term Menu String to menu/item line list. **NewMenu** now allocates only one menu at a time. Special characters in menu/item lines changes; X is now color replace highlighting, ID numbers must be included. **InsertMenu, InsertItem, DeleteMenu, DeleteItem** are complete. Custom menus are defined.

07/16/86 Replacements for pages 8 and 9, I wasn't using proper ID numbers in my examples.

08/13/86 Removed standard color menus. One character has been added to the front of menu and item lines. Added **GetMHandle** and **GetMenuMgrPort** calls. Changed inputs to **SetMenuFlag**. Menu/Item strings may terminate with a zero in addition to a return. Change to menu records.

08/27/86 Added more information about **CalcMenuSize.** Corrected **GetItemFlag** in ERS.

This ERS describes the Menu Manager, the part of the Cortland Toolbox that allows you to create sets of menus, and allows the user to choose from the commands in those menus.

Tool Number: 15

Tools needed installed: Quick Draw
Memory Manager
Event Manager

Stack requirement: 512 bytes.

About the Menu Manager

The Menu Manager supports the use of menus which can be part of the Cortland user interface. Menus allow users to examine all choices available to them at any time without being forced to choose one of them, and without having to remember command words or special keys. The Cortland user simply positions the cursor in the menu bar and presses the mouse button over a menu title. The application then calls the Menu Manager, which highlights the selected title and "pulls down" the menu below it. As long as the mouse button is held down, the menu is displayed. Dragging through the menu causes each of the menu items (commands) in it to be highlighted in turn. If the mouse button is released over an item, that item is "chosen". The item blinks briefly to confirm the choice, and the menu disappears.

When the user chooses an item, the Menu Manager tells the application which item was chosen, and the application performs a corresponding action. When the application completes the action, it removes the highlighting from the menu title, indicating to the user that the operation is complete.

If the user moves the cursor out of the menu with the mouse button held down, the menu remains visible, though no menu items are highlighted. If the mouse button is released outside the menu, no choice is made: The menu just disappears and the application takes no action. The user can always look at a menu without causing any changes in the document or on the screen.

Menu Bars

A menu bar is an outlined rectangle that holds the titles of all the menus associated with the bar. A menu may be enabled or temporarily disabled. A disabled menu can still be pulled down, but its title and all the items in it are dimmed and not selectable.

Keep in mind that if your program is likely to be translated into other languages, the menu titles may take up more space. If you're having trouble fitting your menus into the menu bar, you should review your menu organization and menu titles.

The System Menu Bar

There can be one special type of menu bar which is called the System Menu Bar. There can only be one system menu bar on the screen at one time. The system menu bar always appears at the top of the Cortland screen; nothing but the cursor ever appears in front of it. In applications that support desk accessories, the first menu should be the desk accessory menu (the menu whose title is a colored apple symbol). The desk accessory menu contains the names of all available desk accessories. When the user chooses a desk accessory from the menu, the title of a menu belonging to the desk accessory may appear in the menu bar, for as long as the accessory is active, or the entire menu bar may be replaced by menus belonging to the desk accessory.

Color number 1 is reserved for drawing the Apple logo as the title for the desk accessory menu. Therefore, color number 1 should not be used as the normal, hilite, or outline color. The color can be used for menus, items, nonsystem menu bars, and the rest of the screen.

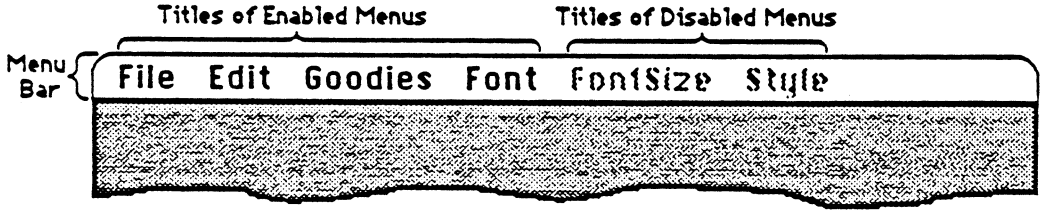
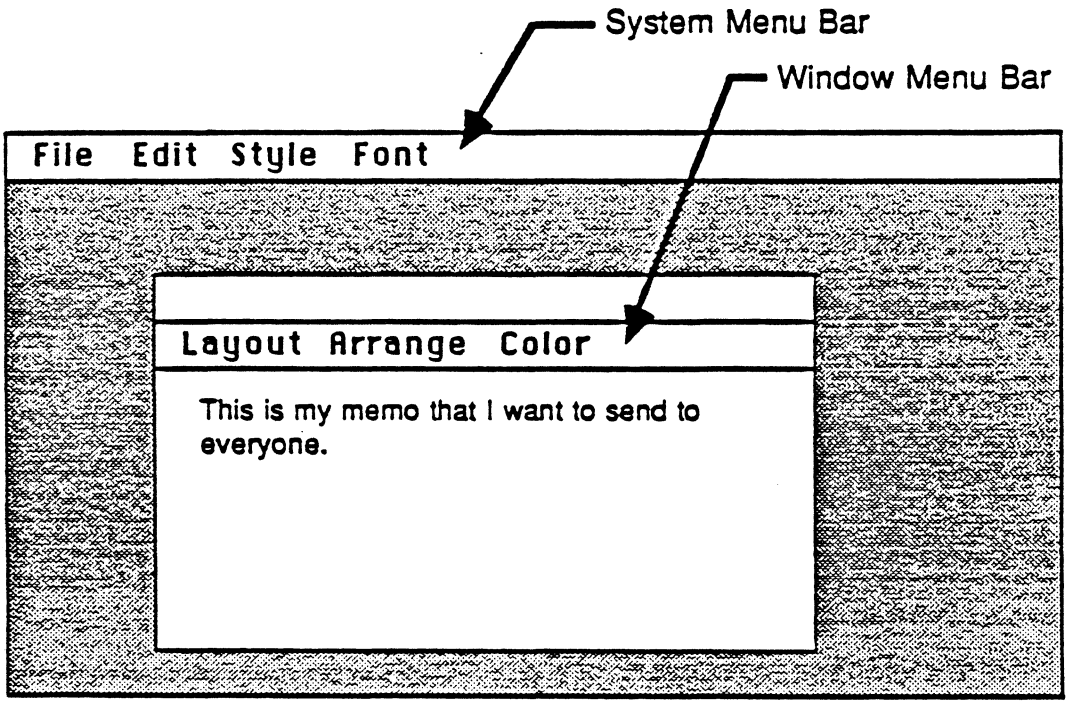


Figure 1. The System Menu Bar

Window Menu Bars

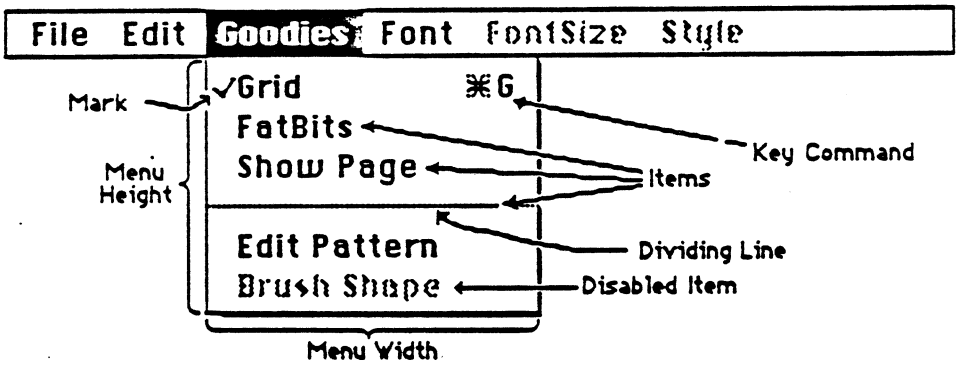
In addition to the System Menu Bar your application can have various window menu bars. These can appear anywhere in windows. Window menu bars are provided to give you more flexibility and to address the limited resolution in 320 mode. Window menu bars should be used moderately, if at all. Window menu bars perform in the same manner as the System Menu Bar.



Appearance of Menus

A standard menu consists of a number of menu items listed vertically inside a shadowed rectangle. A menu item may be the text of a command or just a line dividing groups of choices (see Figure 2). Menus always appear in front of everything else, except the cursor. The menu in figure 2 is a menu with 6 items including one dividing line.

Figure 2. A Standard Menu.



Each item can have a few visual variations from the standard appearance:

- A mark (any character) may appear on the left side of the item, to denote the status of the item or of the mode it controls. See **SetItemMark**, **GetItemMark**, and **CheckItem**.
- A apple symbol on the right side of the item, to show that the item may be invoked from the keyboard (that is, it has a keyboard equivalent), followed by a character. Pressing indicated character while holding down the Command key invokes the item just as if it had been chosen from the menu. See **MenuKey** .
- Each item's text may have its own text style. See **SetItemStyle** and **GetItemStyle**.
- A dimmed appearance, to indicate that the item is disabled, and can't be chosen (dividing lines should always be disabled). See **DisableItem** and **EnableItem**.
- Any menu may be drawn directly by the application and might contain anything (see DEFINING YOUR OWN MENUS).

If the standard menu doesn't suit your needs—for example, if you want more graphics, or perhaps a nonlinear text arrangement—you can define a custom menu that, although visibly different to the user, responds to your application's Menu Manager calls just like a standard menu (see DEFINING YOUR OWN MENUS).

Keyboard Equivalents for Commands

Your program can set up a keyboard equivalent for any of its menu commands so the command can be invoked from the keyboard with the Command key (apple key). You can assign one or two keyboard equivalents per item. One equivalent is the primary, and is displayed to the right of the item. The other equivalent is the alternate and is not displayed. The alternate equivalent should be the lower case equal to the primary equivalent (which should be upper case). See **MenuKey** for a discussion on how items are searched.

> **Note:** For consistency between applications, you should specify an upper case letter as the promary keyboard equivalent.

Using the Menu Manager

To use the Menu Manager, you must have previously initialized QuickDraw. For user interaction you must use the Event Manager. If you are going to be using the Window Manager, it must be initialized before the Menu Manager.

Initialize Menu Manager.

1.)
```
lda     MyID            ID number returned by Memory Manager startup.
pha                     Pass ID to MenuStartup.
lda     MenuZPage       Zero for Menu Manager use, whick can be allocated.
pha                     Pass Zero Page that Menu Manager can use.
_MenuStartUp            Will create and draw an empty menu bar.
                        No output.
```

You now have a system menu bar that contains no menus. It is the current menu bar. The Menu Manager can handle several menu bars by changing the 'current menu bar' (see **SetSysBar**). But if you are only using one menu, most cases, don't even worry about it, the system menu bar will always be current.

Your program then must define menus and items by providing a list of menu and item lines to NewMenu for each menu (see MENU LINES AND ITEM LINES) and InsertMenu to add them to the system menu bar. FixMenuBar may be of use in setting default sizes.

2.)
```
pha                     Space for returned handle.
pha
pea     Menu1|-16       Pass pointer to menu/item lines.
pea     Menu1
_NewMenu                Allocate a menu record and initialize it.
pla                     Get returned menu handle
sta     menuHandle      and save it.
pla
sta     menuHandle+2
ora     menuHandle      Check for bad handle.
beq     error           Unable to allocate handle or bad menu/item lines.

lda     menuHandle+2    Pass handle of menu to insert (just allocated).
pha
lda     menuHandle
pha
pea     0               Insert menu as first menu flag.
_InsertMenu             Insert menu in data structures, not drawn.
```

In the above example, the menu/item line data might look like:

```
DATA:
Menu1       dc      c'>>Title\N1',i1'0'     Menu title.
            dc      c'--Item 1\N256',i1'0'  Item text.
            dc      c'.'                    Termination character.
```

After all the menus you would like have been created and inserted you must set the sizes of the menu bar and menus (NewMenu does not set the menu's size).

3.)
```
pha                     Space for returned bar height.
_FixMenuBar             Initialize menu bar/menus sizes.
pla                     Get the height of the menu bar.  Generally of no use.
```

After created and initialized the new menu bar should be drawn. If you would like, change the color of the menu bar and menus with SetBarColors before drawing the menu.

4.)
```
_DrawMenuBar            Draws the menu bar and the menu titles.
```

The menu bar is up and initialization is complete.

Now you are ready to accept input from the user and you have a choice. The first choice is to use Task Master. Task Master is part of the Window Manager and is defined in that document. As you will see section A describes how to interact with menu not using Task Master and is more to understand and debug than section B that uses Task Master.

```
TaskRec     anop
what        ds      2           First part is an event record, defined in Event Manager.
message     ds      4
when        ds      4
where       ds      4
modifiers   ds      2
TaskData    ds      4           Place for returned values.
TaskMask    ds      4           Used by Window Manager calls, not really needed here.
```

Section A, not using Task Master:

```
poll        pha                 Space for returned value.
            pea     $002C       Accept key and mouse down events (at least these events).
            pea     TaskRec|-16 Pass pointer to event record.
            pea     TaskRec
            _GetNextEvent
            pla                 Get return flag.
            beq     poll        Was there an event?

            lda     what        Get event code.
            cmp     #3          Key press event?
            bne     ck_button
            cmp     #5          Auto-key event?
            bne     ck_button

            pea     TaskRec|-16 Pass TaskRec which contains the key pressed in 'message'.
            pea     TaskRec
            pea     0           Use current menu bar flag.
            pea     0
            _MenuKey
            bra     ck_Menu     Execute common menu selection routine.

ck_button   cmp     #1          Mouse down event?
            bne     poll        If not key or button, continue to poll.
```

You should first determine, but it's absolutely necessary, if the button was pressed in the menu bar. If you are using the Window Manager, call **FindWindow**. It will return a value that states in point is in the system menu bar. If you are not using the Window Manager it is application specific as to how to determine this. Using the height of the menu bar, returned from **FixMenuBar**, could be of use.

Continue if you find the point is in the menu bar.

```
            pea     TaskRec|-16     Pass TaskRec which contains the key pressed in 'message'.
            pea     TaskRec
            pea     0               Use current menu bar flag.
            pea     0
            _MenuSelect             Menu select will wait for a button up before returning.

ck_Menu     lda     TaskData        Get ID of item selected.
            beq     poll            Was an item selected.  If not, return to polling.

            cmp     #256            Item IDs 1-255 are reserved for desk accessory items.
            bcc     call_deskmgr    Do what is necessary for desk accessories.

            and     #$00FF          Short cut for table, if all item IDs are between 256 and
511.
            asl     a
            tax
            jmp     (itemProcs,x)   Jump to item handler.

itemProcs   dc      i'item1'        Address for item handler (we only have one item).

item1       (perform whatever action the item dictates)

            pea     0               Pass FALSE to unhighlight the menu title.
            lda     TaskData+2      Get menu's ID, returned from MenuKey or MenuSelect.
            pha                     Pass the menu's ID number.
            _HiliteMenu             Unhighlight the menu's title, after item action.

            jmp     poll            Return to polling.
```

Section B, using TaskMaster:

```
poll        pha                     Space for returned value.
            pea     $002C           Accept key and mouse down events (at least these events).
            pea     TaskRec|-16     Pass pointer to Task record.
            pea     TaskRec
            _TaskMaster             Task Master will call GetNextEvent for you.
            pla                     Get return flag.
            beq     poll            Was there an event?

            cmp     #18             Was there a menu event?
            bne     poll            If not, continue to poll (or check other events).

            lda     TaskData        Get ID of item selected.
            and     #$00FF          Short cut for table, if all item IDs must be 256 to 511.
            asl     a
            tax
            jmp     (itemProcs,x)   Jump to item handler.

itemProcs   dc      i'item1'        Address for item handler (we only have one item).

item1       (perform whatever action the item dictates)

            pea     0               Pass FALSE to unhighlight the menu title.
            lda     TaskData+2      Get menu's ID, returned from MenuKey or MenuSelect.
            pha                     Pass the menu's ID number.
            _HiliteMenu             Unhighlight the menu's title, after item task action.

            jmp     poll            Return to polling.
```

Note: The Menu Manager will try to automatically save and restore the screen behind the menu, or tell the Window Manager to update the screen. However, if you are not using the Window Manager and the Menu Manager can not allocate a buffer large enough to save the screen behind the menu, your application will have to update the screen area after a menu has been pulled down. See CheckRedraw.

If your menu bar, or items in a menu, are going to change while on the screen you can use **SetMenuTitle**, **InsertMenu**, **DeleteMenu**, **SetItem**, **InsertItem**, and **DeleteItem** to rearrange the menus and items.

There are several miscellaneous Menu Manager routines that may be of use to applications. **CalcMenuSize** calculates the dimensions of a menu and is called by **FixMenuBar**. **CountMItems** counts the number of items in a menu. **FlashMenuBar** inverts the menu bar, or just a menu title. **SetItemBlink** controls the number of times a menu item blinks when it's chosen.

Menu Lines and Item Lines

Menus may be created by passing a pointer to a list of **menu** and **item lines** to **NewMenu** which will parse them and allocate enough memory for necessary records, and initialize those records. The list can be edited using a word processor, thus allowing users to easily customize their own menus. An example of a list is:

```
>>Title 1\N1
--Item string 1\N256
--Item string 2\N257
--Item string 3\N258
.
```

This is a simple list of one menu line and 3 item lines. The first character on a line denotes the start of a menu or an item in a menu. Each line is terminated by a return (decimal 13) or a null byte (0). The character to denote a title is whatever the very first character is in the first line. The character to denote items is the first character on subsequent lines that is different from the title character. And lastly, a character different from the item character, and after the title, denotes the end of the list. In the example, the '>' character is the title charcter, '-' is the item character, and '.' is the terminating character. However, any characters may be used, as long as the title and item characters are different, and the termination character is different from the item character. So, the title and termination character may be the same.

The second character on each line (other than the termination line) is a place holder for the length of the string. **NewMenu** will replace the second character with the string's length. Therefore, after a **NewMenu** call the string data will have been altered.

> Note: The menu/item string must stay in their original memory. **NewMenu** sets pointers in the menu record to the address of the strings.

In the example you'll notice a backslash, '\', followed by a 'N' and a number. The backslash denotes the end of a title's text and the beginning of special characters. The 'N' is a special character that precedes an ID number. A decimal, unsigned, ASCII ID number immediately follows. Every menu title and item must have an ID number, even dividing lines. The ID number for each menu title should be different from every other menu on a menu bar. The ID of an item should be different from every other item on a menu bar. Items that are dividing lines, and always disabled, can have the same ID number.

Special characters are:

\	Beginning of special characters.
*	Followed by a primary, then an alternate character to be used as a keyboard equivalents. Use a space for no alternate character.
B	Bold the text.
C	Followed by a character to be used to mark the item.
D	To dim (disable).
H	Hexidecmal, nonASCII, ID number follows, low byte/high byte.
I	Italize the text.
N	Decimal ASCII ID number follows, any length, between 1 and 65535.
U	Underscore the text.
V	Places a dividing line under the item without using a separate item.
X	Use color replace, and not XOR, highlighting.

All the special characters pertain to items. Special characters *, B, C, I, U, and V do not pertain to menu titles.

An example of a menu and item lines using mulitple special characters and different title, item, and terminating characters:

$$Title 1\N1	Title character is '$', ID = 1, can be same as an item's.
--Item string 1\N256*Xx	Item character is a dash, ID = 256, key eqivalents X and x.
--Item string 2\BC√UN257	Item character is a dash, bold, checked, underscored, ID = 257.
--Item string 3\IN258	Item character is a dash, text will appear italized, ID = 258.
$	Terminating character, can be the same as the title character.

Some more special stuff. Using just the @ symbol in a title will give you the Apple logo. An example of an Apple logo menu title:

$$@\N1X	Apple logo title, ID = 1, color replace highlighting.

Note: The X special character (color replace highlighting) should always be used with the Apple logo.

Note: To get the Apple logo the @ must follow the character denoting a menu title, and then be followed by a special character begin mark or an end of line mark (return). Do not place a space before or after the @, like you should for other menu titles.

There is no way to include a '\' in a title's string. It will always be seen as the beginning of special characters.

A single dash, '-', for an item's text will denote a dividing line. Special characters apply to dividing lines. Dividing lines should always be marked as dimmed, 'D'. It would look like:

>>-\N256D	If the '>' character denotes an item line.

ID Number Assignment

ID numbers are not assigned automatically, it must be assigned in the menu/item line list. Item ID numbers are allocated accordingly:

$0000	0	Internal use, generally means front, or first item in menu.
$0001 - $00F9	1-249	Reserved for desk accessory items.
$00FA	250	**Undo** edit item.
$00FB	251	**Cut** edit item.
$00FC	252	**Copy** edit item.
$00FD	253	**Paste** edit item.
$00FE	254	**Clear** edit item.
$00FF	255	Close command item.
$0100 - $FFFE	256-65534	Reserved for application use.
$FFFF	65535	Internal use, generally means end, or last item in menu.

Menu ID numbers are allocated accrodingly:

$0000	0	Internal use, generally means front, or first menu in bar.
$0001-$FFFE	1-65534	Reserved for application use.
$FFFF	65535	Internal use, generally means end, or last menu in bar.

What ID numbers to use? Here are two suggestions for schemes in ID number assignment. The first is to number menus from 1 to n, and items from 256 to 256+n. The item ID can then be used to index into a table of selection handling routines when a selection is made. The other scheme is to use the lower WORD of the handling routine's address as the ID. Different items still must have different ID numbers, so NOPs at the head of the routine could be used for different entry points and ID numbers.

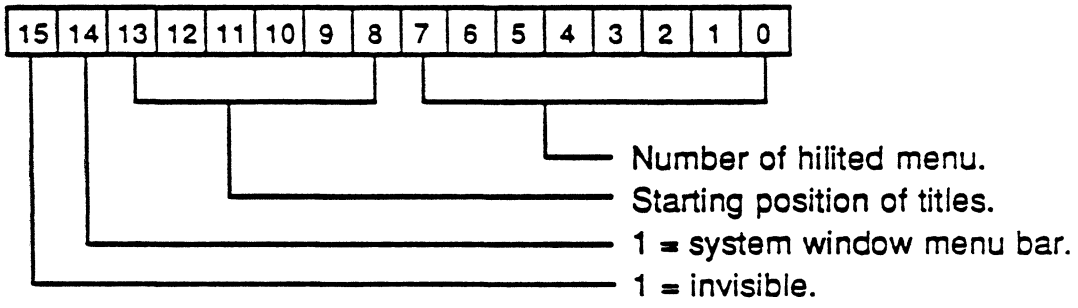
ID numbers can later be changed to anything you would like with calls to, **SetItemID**, **GetItemID**, **SetMenuID**, and **GetMenuID**.

Menu Records

The Menu Manager keeps all the information it requires for its operations on a particular menu bar in a **menu bar record.** The record contains the menu's position, color, menu lists, item lists, and other flags the Menu Manager needs to manage menus. The menu bar record is the same as a control record.

NextCtrl	LONG	Handle of next control.
CtrlOwner	LONG	Window menu belongs to, zero for system menu bars.
CtrlRect	RECT	Coordinates of menu bar.
CtrlFlag	WORD	Defined below.
CtrlValue	WORD	Not used, should be zero.
CtrlProc	LONG	$0A000000.
CtrlData	LONG	TRUE if system window's menu, FALSE if application's.
CtrlRefCon	LONG	Reserved for application's use.
CtrlColor	LONG	Pointer to color table, defined below.
MenuList	LONG[]	Array of menu handles, zero terminates list.

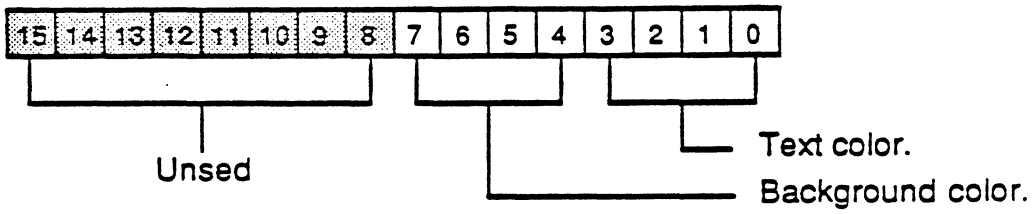
Menu Bar - CtrlFlag:



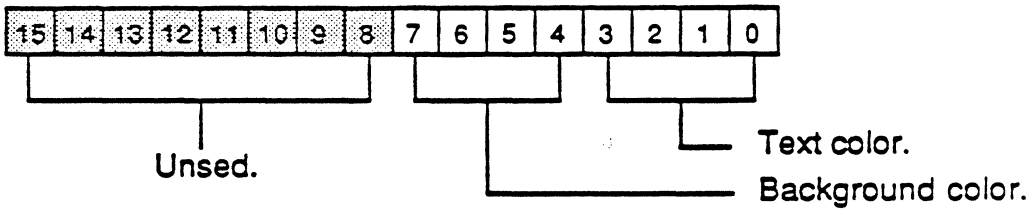
** Invisible flag is not yet implemented.*

Menu Bar Color Table:

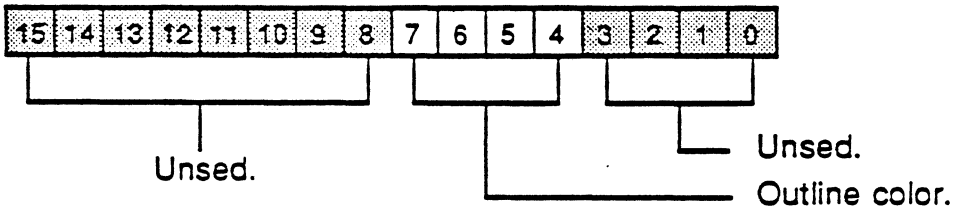
Color 0 - unhighlighted color of text and background:



Color 1 - highlighted color of text and background:



Color 2 - color of outline:



The MenuList is an array of menu handles in the menu bar. Menus records are only partially defined. Only the first half of the record is definded.

MenuID	WORD	Menu's ID number.
MenuWidth	WORD	Width of menu.
MenuHeight	WORD	Height of menu.
MenuProc	LONG	Pointer to menu definition procedure, zero for standard menu.
MenuFlag	BYTE	Defined below.
MenuRes	BYTE	Reserved.
FirstItem	BYTE	Reserved.
NumOfItems	BYTE	Reserved.
TitleWidth	WORD	Width of title.
TitleName	LONG	Pointer to title text, first byte equals length.

(the rest of record is not defined)

MenuID — See ID Number Assignment.

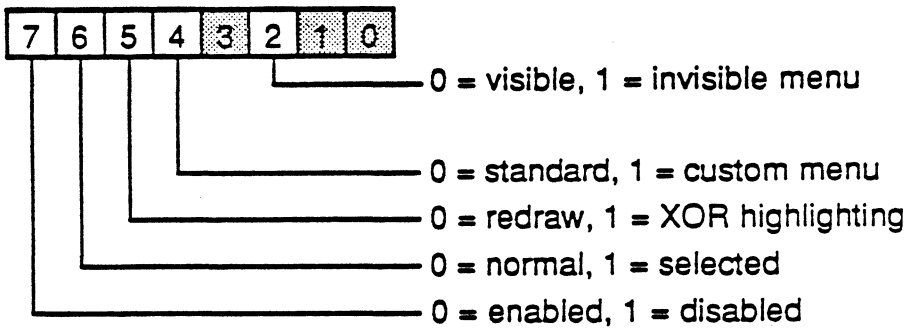
MenuWidth — The width of the menu is generally set by **CalcMenuSize**. However, the value can be changed by the application. Text will be drawn outside of the menu, and trash othere things on the screen, if the width is set to narrow.

MenuHeight — The height of the menu is generally set by **CalcMenuSize**. However, the value can be changed by the application. Text will be drawn outside of the menu, and trash othere things on the screen, if the width is set to short.

MenuProc — This field contains the address of the menu's definition procedure, unless it is a standard menu definition procedure. Standard definition procedures put a value between 0 and 255. Values greater than 255 are considered to be the address of a custom menu definition procedure and are called accordingly. See Defining Your Own Menus for more information about custom menus.

MenuFlag — This is a bit vector that flag various menu attributes. The grayed bits are undefined and may contain information internally used by Menu Manager, their values should never be changed by an application.

MenuFlag



** Invisible flag is not yet implemented.*

MenuRes — This BYTE is resevered and should not be relied on for information or modified by an application.

FirstItem — Resevered for the future implementation of scrollable menus. This field should not be relied on for information or modified by applications written before the implementation is defined.

NumOfItems — Resevered for the future implementation of scrollable menus. This field should not be relied on for information or modified by applications written before the implementation is defined.

TitleWidth — The width of the title can be set to any value between 1 and $FFFF (unsigned). The value is used to highlight the title and compute where the next menu title should start. The title's text is left justified.

TitleName — Pointer to the string to be used for the menu's title. The first byte of the string should be the length of the string followed by the ASCII text. Custom menus may store any type of value desired as long as the title is drawn by the custom definition procedure.

Not defining menu records completely has good and bad sides. Access to menu information will be slower if calls to the Menu Manager have to be made. However, the delay would have to be measured in miliseconds, and the delay never seen on the screen. On the plus side, future Menu Managers would not be tied to an older, possibly inadequate, record structure. The chances of improving the current Menu Manager, and maintaining compatibility accross future hardware, is greatly improved by allowing records to change.

Item records not defined at all for standard menus.

Defining Your Own Menus

The standard type of menu is predefined for you. However, you may want to define your own type of menu—one with more graphics, or perhaps a nonlinear text arrangement. QuickDraw and the Menu Manager make it possible for you to do this.

To define your own type of menu, you write a menu definition procedure. The Menu Manager calls the menu definition procedure to perform basic operations such as drawing the menu.

To create a custom menu record you will have to allocate a block of memory large enough for your menu record. Only the defined part (see MENU RECORDS) of the menu record has to follow Menu Manager form, the rest of the record is up to you. Another way is to pass a menu line with no items to **NewMenu** and then resize the allocated block to your needs. Fields in the menu record that need to be initialized are:

MenuID	Menu's ID number.
MenuWidth	Width of menu, or you can wait for the mSize.
MenuHeight	Height of menu, or you can wait for the mSize.
MenuProc	Pointer to menu definition procedure.
MenuFlag	In addition to other flags, bit 4 must be set.
TitleWidth	Width of title.
TitleName	Pointer to title text, first byte equals length. Or some other data you wish.

The Menu Definition Procedure

You may choose any name you wish for the menu definition procedure. The inputs and outputs are:

inputs:	message:WORD	Operation to perfrom.
	theMenu:LONG	Handle of menu.
	RectPtr:LONG	Pointer to RECT enclosing menu.
	xHitPt:WORD	X coordinate of point to check.
	yHitPt:WORD	Y coordinate of point to check.
	param:WORD	Addition parameter for each operation.
output:	Result:WORD	Depends on operation.

Warning: Do not change the RECT pointed to by RectPtr. If you need to change it make a copy of the RECT and then change the copy.

Warning: Do not call the Menu Manager from within a custom definition procedure.

Note: The term 'item number' referred to in the following sections of this document is not the same thing as an 'item ID number'. Item number can be any value, although zero and bit 15 have specific meanings. The Menu Manager only uses the value to pass back to the definition procedure and comparing it to other item numbers returned by the same definition procedure during a previous call. The definition procedure can use any numbering system desired, but it should be the same system for each definition procedure function.

The message parameter identifies the operation to be performed. It has one of the following values:

mDrawMenu	= 0	Draw the menu.
mChoose	= 1	Tell which item was chosen and highlight it.
mSize	= 2	Calculate the menu's dimensions.
mDrawTitle	= 3	Draw the menu's title.
mDrawItem	= 4	Highlight or unhighlight an item.
mGetItemID	= 5	Return item's ID number.

mDrawMenu

inputs:	message:WORD	mDrawMenu, 0.
	theMenu:LONG	Handle of menu.
	RectPtr:LONG	Pointer to RECT that defines the menu's interior.
	xHitPt:WORD	Not defined.
	yHitPt:WORD	Not defined.
	param:WORD	Not defined.
output:	Result:WORD	Not used, any value.

The message mDrawMenu tells the menu definition procedure to draw the menu inside the RECT pointed to by RectPtr. The RECT passed is the coordinates of the menu's interior, that is, the menu less its frame. The current grafPort will be the Menu Manager port. The standard menu definition procedure figures out how to draw the menu items by looking in the menu record at the data that defines them. For menus of your own definition, you may set up the data defining the menu items any way you like. You should also check the enableFlags field of the menu record to see whether the menu is disabled (or whether any of the menu items are disabled, if you're using all the flags), and if so, draw it in gray. You may even print the items in a different font, as long as you restore the original when you finish. Returned value is not used.

mChoose

inputs:	message:WORD	mChoose, 1.
	theMenu:LONG	Handle of menu.
	RectPtr:LONG	Pointer to RECT enclosing menu.
	xHitPt:WORD	X coordinate of point to check.
	yHitPt:WORD	Y coordinate of point to check.
	param:WORD	Not defined.
output:	Result:WORD	Zero if point is not over any item, else the item number of the item the point is over.

When the menu definition procedure receives the message mChoose, the yHitPt/xHitPt parameter is the mouse location (in global coordinates). The procedure should determine whether the given point is within an enabled menu item. Before call this function the Menu Manager checkes that the point is within the given RECT and that the menu is enabled.

- If the mouse location is in an enabled menu item, return the item number, with the high bit set, of the item.
- If the mouse location isn't in an enabled item return zero.

mSize

inputs:	message:WORD	mSize, 2.
	theMenu:LONG	Handle of menu.
	RectPtr:LONG	Not defined.
	xHitPt:WORD	Not defined.
	yHitPt:WORD	Not defined.
	param:WORD	Not defined.
output:	Result:WORD	Not defined, not used.

The message mSize tells the menu definition procedure to calculate the horizontal and vertical dimensions of the menu. The menu width should be computed and stored in the MenuWidth field of the menu record if MenuWidth was zero on entry, the same is true for MenuHeight. Do not replace the value in the MenuWidth field if it is nonzero on entry. Do not replace the value in the MenuHeight field if it is nonzero on entry.

mDrawTitle

inputs:	message:WORD	mDrawTitle, 3.
	theMenu:LONG	Handle of menu.
	RectPtr:LONG	Pointer to RECT enclosing title area.
	xHitPt:WORD	Not defined.
	yHitPt:WORD	Not defined.
	param:WORD	0 =draw normal, 1=draw inverted, bit 15 set to invert.
output:	Result:WORD	FALSE to draw default text title, TRUE if your definition procedure drew the title.

When the menu definition procedure receives the message mDrawTitle when the title of the menu must be drawn. Param is:

$0000 - if the title should be completely drawn, called this way from DrawMenuBar.
$0001 - if the title should be drawn as highlighted without special highlighting.
$800x - negative to use special highlighting.

If the menu's MenuFlag indicates that XOR highlighting is not to be done, param will never be negative. It is left to your definition procedure to define what special highlighting is. In the standard definition procedure the title RECT is XORed.

The RECT pointed to by RectPtr encloses the title area. Return FALSE to have the Menu Manager draw the title (the TitleName field of the menu record must contain a pointer to a text string for the title), otherwise return TRUE.

mDrawItem

inputs:	message:WORD	mDrawItem, 4.
	theMenu:LONG	Handle of menu.
	RectPtr:LONG	Pointer to RECT enclosing menu.
	xHitPt:WORD	Not defined.
	yHitPt:WORD	Not defined.
	param:WORD	Item number, high bit set to highlight, clear for unhighlighted.
output:	Result:WORD	Not defined, not used.

The mDrawItem command is a request to draw an item in its highlighted or unhighlighted state. If param is positive it is the item number and should be draw draw as unhighlighted. If param is negative it should be drawn as highlighted. Bits 0-14 of param are the same as the item number returned at one time by mChoose function of your definition procedure.

mGetItemID

inputs:	message:WORD	mGetItemID, 5.
	theMenu:LONG	Handle of menu.
	RectPtr:LONG	Not defined.
	xHitPt:WORD	Not defined.
	yHitPt:WORD	Not defined.
	param:WORD	Item number.
output:	Result:WORD	Item's ID number.

Param equals the item's number and the definition procedure is asked to return the item's ID number. The item number is the value returned by mChoose with the high bit masked off.

Dividing lines vs. Underlines

Dividing Lines

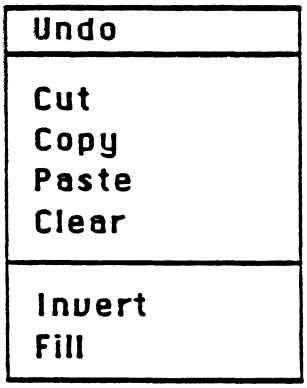
There are two standard ways to partition groups of items from one another. The first is a *dividing line*, selected by an item title which is a single dash. It uses the space of an entire item and a whole item record. The second way is a *underline*, set either in the menu line, or **SetItemFlag**. This will draw a solid line on the bottom most line of the item. The underline doesn't use any more space, on the screen or in memory, than the item would without it..

The disadvantage with an underline is there isn't as much space separating items, which is the dividing line's function.

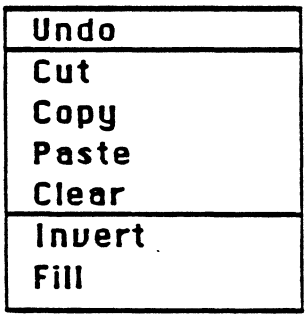
The advantage of an underline is you can get more items in the menu and still have dividing lines. Also, the user would have a shorter distance to go from the menu's title to the last item in the menu, it would save a little memory, and the menu would draw faster.

In the example below are two menus, both showing the same information. Menu A uses dividing lines and has 9 items. Menu B uses underlines and has 7 items. Menu B looks alittle crowded and would look even worse if one of the uderlined items had descending lower case letters.

Menu A - Dividing Lines



Menu B - Underlines



Cortland Menu Manager

Appendix A
Menu Calls

INITIALIZATION AND TERMINATION ROUTINES

MenuBootInit

input: None.
output: None.

Called when SetTSPtr is called.

MenuStartup

input: userID:WORD - user ID that the Menu Manager can use, mainly to allocate memory.
zeropg:WORD - zero page Menu Manager can use, must be on page boundary.

output: None.

Initializes system menu bar with no menus, and makes it the current menu bar.
Calls Desktop in the Window Manager to reserve space for the bar.
Menu Manager opens a grafPort.
Calls DrawMenuBar to draw an empty system menu bar.

MenuShutDown

input: None.
output: None.

Closes the Menu Manager's port and frees any allocated menus.

MenuVersion

input: None.

output: version:WORD - Menu

Manager's version number.

MenuReset

input: None.
output: None.

Does nothing.

MenuStatus

inputs: None.

output: status:WORD - TRUE if Menu Manager is initialized, else FALSE.

NewMenuBar

inputs: theWindow:LONG - pointer to window's port, owner of menu bar, zero for system.

output: BarHandle:LONG - handle of menu bar.

NewMenuBar will create a default menu bar with no menus. **MenuStartup** calls **NewMenuBar** to create a default system menu bar. If you are only going to use one system menu bar, **NewMenuBar** will not have to be called. The default size of the menu bar is, upper left corner matches the port, and the width is the width is the width of the screen. The height of the bar is 13. The menu bar is visible and has default colors of black text on a white background.

NewMenu

input: MenuString:LONG - pointer to a menu/item line list.

output: MenuHandle:LONG - handle of menu, zero if error.

NewMenu allocates space for a menu and its items. You pass a pointer to a menu/item line list which is a text string that describes the menu title and its items. See MENU LINES AND ITEM LINES for the format needed. The MenuHandle returned can then be inserted in a menu bar via an **InsertMenu** call.

Call **DisposeMenu** to deallocate the menu when finished.

DisposeMenu

input: MenuHandle:LONG - previously allocated via NewMenu.

output: None.

Frees the memory used by MenuHandle. The menu will no longer be usable.

Warning: The menu is not taken out of the menu list, call **DeleteMenu** to do that. To delete a menu from the menu list and free it's memory you could do something like this:

```
pha             Space for returned handle.
pha
pea  MenuID     ID of menu to delete.
_GetMHandle     Get the handle of the menu.
                Leave menu handle on stack.

pea  MenuID     ID of menu to delete from list.
_DeleteMenu     Delete menu from list.

                Handle still on stack.
_DisposeMenu    Deallocate menu record.
```

FixMenuBar

input: None.

output: MenuHeight:WORD - height of the menu bar.

This routine will compute standard sizes for your menu bar and menus.

FixMenuBar will search all the menu title fonts and use the tallest one to compute the height of the menu bar, add it to Bar.top, and store it in Bar.bottom. It will set the TitleWidth width for every menu TitleWidth that is given as zero. Finally it will call CalcMenuSize for each menu in the menu bar with newWidth and newHeight as negative values.

CalcMenuSize

input: newWidth:WORD - width, zero or negative.
newHeight:WORD - height, zero or negative.
MenuNum:WORD - menu ID number.

output: None.

This call lets you set menu dimensions, or have the Menu Manager do it.

newWidth	Function
Zero	Default width will be computed no matter what MenuWidth is.
Negative	Defalut width will only be computed if MenuWidth is zero.
$0001-$7FFF	MenuWidth will be set to newWidth and default width not computed.

newHeight	Function
Zero	Default height will be computed no matter what MenuHeight is.
Negative	Defalut height will only be computed if MenuHeight is zero.
$0001-$7FFF	MenuHeight will be set to newHeight and default height not computed.

To compute the width the Menu Manager will find the widthest item in the menu plus room for a mark and command key. A default width will be used if the menu does not contain any item text.

To compute the height, the Menu Manager will add up the font height of each item plus four, or use the value found in the font index of ItemFlag if bit 14 of ItemFlag is set.

This routine is called for each menu by the Menu Manager with newWidth and newHeight negative when **FixMenuBar** is called.

USER INTERACTION ROUTINES

MenuSelect

input: TaskRec:LONG - pointer to Task record which contains point of button down.
BarHandle:LONG - handle of menu bar, zero for system menu bar.

output: None ('TaskData' field of Task record contains return IDs).

Called when the a button goes down on a menu bar (see **FindWindow** if using the Window Manager). The routine will take care of drawing highlighted titles, pulling down menus, and user interaction. This is handled automatically for the system menu bar when using TaskMaster in Window Manager.

If a selection is made the low order WORD of the 'TaskData' element in the Task record will contain the ID number of the item selected, and the high order WORD will contain the menu's ID number. If there is a selection, the menu's title will be left highlighted. See **HiliteMenu** to redraw the title as normal.

If no selection is made by the user the low order WORD of the 'TaskData' element in the Task record will be zero.

BarHandle becomes the current menu bar.

The structure of TaskRec is:

what	WORD	Event record portion, unchanged from GetNextEvent.
message	LONG	
when	LONG	
where	LONG	
modifiers	WORD	
TaskData	LONG	Extended portion for TaskMaster.

MenuKey

input: TaskRec:LONG - pointer to Task record which contains the character to check.
BarHandle:LONG - handle of menu bar, zero for system menu bar.

output: None ('TaskData' field of Task record contains return IDs).

Maps the given character to the associated menu and item for that character. When you get a key-down event with the Command key held down—or auto-key event, if the command being invoked is repeatable—call **MenuKey** with a pointer to a Task record that contains the character and the state of the modifier keys (the format is the same as an Event record, see Event Manager). The match will only occur if it is indicated in the Task record that the command key was down. **MenuKey** highlights the appropriate menu title if the key matches, and returns Selection.

The items are searched starting with the first menu in the menu list and all the items in the menu starting with the first. Then the second menu, and so on. The given key is compare with every item's primary keyboard equivalent of every item. If no match is found, the cycle is repeated, this time comparing to each item's alternate keyboard equivalent.

There generally there should never be more than one item in the menu list with the same keyboard equivalent, but if there is, **MenuKey** returns the first one it encounters.

MenuKey will not convert lower case characters to upper case. If you want to match on either upper or lower case, set the primary character to the upper case character and the alternate to the lower case character.

If a selection is made the low order WORD of the 'TaskData' element in the Task record will contain the ID number of the item selected, and the high order WORD will contain the menu's ID number. If there is a selection, the menu's title will be left highlighted. See **HiliteMenu** to redraw the title as normal.

If no selection is made by the user the low order WORD of the 'TaskData' element in the Task record will be zero.

BarHandle becomes the current menu bar.

See **MenuSelect** for a description of TaskRec.

MenuRefresh

input: RedrawRoutine:LONG - address of routine in your application.

output: None.

Note: This is called only when using the Menu Manager without the Window Manager.

RedrawRoutine is called when the Menu Manager can not restore the screen under a menu. First the Menu Manager will try to allocate a buffer large enough to save the screen part before it draws the menu. If the buffer is allocated the screen will be restored from it and then deallocate the memory buffer. If the buffer can not be allocated the Menu Manager will try to call the Window Manager (via the call the Window Manager made to MenuRefresh during initialization) to refresh the screen when the menu goes away. If no buffer can be allocated and the Window Manager isn't installed, the Menu Manager will call RedrawRoutine to refresh the screen under the menu.

The RedrawRoutine should look something like this:

```
Refresh     START
;
rect_addr   equ     6       Offset down stack to RECT pointer.
;
                    :
                    :
            Needed operations to redraw the
            screen inside the given RECT.
                    :
                    :
;
;
; Remove the given pointer from the stack:
;
                    lda     0,s             Move the return
      address down
                    sta     4,s              the stack.
                    lda     2,s
                    sta     6,s
:
                    pla                     Move the stack back to
      the return
                    pla                      address.
;
                    rtl                     Return to the Menu
      Manager.
```

DRAWING

DrawMenuBar

input: None.

output: None.

Draws the current menu bar, along with any menu titles on the bar.

HiliteMenu

input: Hilite:WORD - FALSE to draw normal, TRUE to highlight the title.
MenuNum:WORD - menu's ID.

output: None.

MenuNum is the the menu's ID. Its title is drawn using the menu bar's normal color if Hilite is FALSE, or hilite color if TRUE. **HiliteMenu** should be called with Hilite FALSE, and the menu ID of the selected menu, after the application has finished acting on a menu selection.

FlashMenuBar

input: None.

output: None.

This will redraw the entire current menu bar using the bar's hilite color and then again using its normal color.

MENU AND ITEM SHUFFLING

InsertMenu

input: AddMenu:LONG - handle of menu to insert.
InsertAfter:WORD - menu ID, zero to insert at front.

output: None.

Inserts AddMenu into the current menu bar after InsertAfter, or at the front of the list if InsertAfter is zero. **DrawMenuBar** should be called to redraw the new menu bar after **InsertMenu.**

DeleteMenu

input: MenuNum:WORD - menu ID of menu to delete.

output: None.

MenuNum is take out of current menu bar. **DrawMenuBar** should be called to redraw the new menu bar after **DeleteMenu.** The menu is not deallocated, call **DisposeMenu** to do that.

InsertItem

input: AddItem:LONG - address of item line to insert.
InsertAfter:WORD - item ID, zero to add to front, $FFFF to append to end of menu.
MenuNum:WORD - menu ID number to add item to, zero for first menu.

output: None.

Inserts an item into the ItemList after InsertAfter. If InsertAfter is zero, the item will be inserted at the front of MenuNum. If InsertAfter is $FFFF, the item will be appended at the end of MenuNum. If MenuNum is zero, the menu will be considered the first menu. Call **CalcMenuSize** to resize the menu if needed afterward. See MENU LINES AND ITEM LINES for the definition of an item line.

DeleteItem

input: ItemNum:WORD - item ID of item to delete.

output: None.

ItemNum is taken out of ItemList of its menu in the current menu bar. Call CalcMenuSize to resize the menu if needed afterward.

MENU BAR ACCESS

SetSysBar

input: BarHandle:LONG - handle of new system menu bar.

output: None.

Handle of new system menu bar is given. The system menu bar becomes the current menu bar.

GetSysBar

input: None.

output: BarHandle:LONG - handle of the system menu bar.

Returns the handle of the system menu bar.

SetMenuBar

input: BarHandle:LONG - handle of current menu bar.

output: None.

Handle of menu bar to make current is given. If you want the system menu bar to be the current menu bar, pass zero for BarHandle.

GetMenuBar

input: None.

output: BarHandle:LONG - handle of current menu bar.

Returns the handle of the current menu bar.

SetBarColors

input: NewBarColor:WORD - normal bar color.
NewInvertColor:WORD - selected bar color.
NewOutColor:WORD - Outline color in bits 7-4.

output: None.

NewBarColor: bits 0-3 = text color when not selected.
bits 4-7 = background color when not selected.
bits 8-15 = zero.
Negative to not change normal color.

NewInvertColor: bits 0-3 = text color when selected.
bits 4-7 = background color when selected.
bits 8-15 = zero.
Negative to not change hilite color.

NewOutColor: bits 0-3 = zero.
bits 4-7 = color of menu bar outline, menu outline, underlines, and dividing lines
bits 8-15 = zero.
Negative to not change outline color.

Color of current menu bar is set to given values that are not negative. Call **DrawMenuBar** to draw menu bar in new colors.

GetBarColors

input: None.

output: Colors:LONG - colors of menu bar.

Returned menu bar colors are returned in one LONG of which:

bits 24-31 = zero.
bits 20-23 = color of menu bar outline, menu outline, underlines, and dividing lines
bits 16-19 = zero.
bits 12-15 = background color when selected.
bits 8-11 = text color when selected.
bits 4-7 = background color when not selected.
bits 0-3 = text color when not selected.

SetTitleStart

input: XStart:WORD - menu bar title starting position.

output: None.

XStart is the number of pixels from the left side of the menu bar that the titles should start. Xstart should be at least 1. Zero will over write the left side outline of the menu bar. 127 is the maximum value allowed.

GetTitleStart

input: None.

output: XStart:WORD - menu bar title starting position.

XStart is the number of pixels from the left side of the menu bar that the titles start from.

CountMItems

input: MenuNum:WORD - menu's ID.

output: NumOfItems:WORD - number of items in menu.

Returns the number of items, including any dividing lines, in the menu.

MENU RECORD ACCESS ROUTINES

GetMHandle

input: MenuNum:WORD - menu ID.

output: MenuHandle:LONG - handle of menu, zero if error.

Handle of menu with an ID number that matches menuNum is returned, or zero if the menu is not found.

SetTitleWidth

input: NewWidth:WORD - new width of title.
MenuNum:WORD - menu ID.

output: None.

Sets the width of a title. This is the area where the user can select a menu and the area that is inverted when the title is highlighted.

GetTitleWidth

input: MenuNum:WORD - menu ID.

output: TheWidth:WORD - width of title, zero if error.

Returns the width of a title. This is the area where the user can select a menu and the area that is inverted when the title is highlighted.

SetMenuFlag

input: NewValue:WORD - new bit value to set or clear.
MenuNum:WORD - menu ID.

output: None.

Possible NewValues:

EnableMenu	$FF7F	Menu will not be dimmed and will be selectable.
DisableMenu	$0080	Menu will be dimmed and not selectable.
ColorReplace	$FFDF	The menu's title and background will be redrawn to hilite.
XORhilite	$0020	The menu's title area will be XORed to hilite.
StandardMenu	$FFE7	The menu will be considered a standard menu.
CustomMenu	$0010	The menu will be considered a custom menu.

If you change a flag that affects the appearance of a menu title you should also call **DrawMenuBar** after **SetMenuFlag** to redraw the titles in their new state.

GetMenuFlag

input: MenuNum:WORD - menu ID.

output: MenuState:WORD - desired bits from MenuFlag.

Returns MenuNum_MenuFlag (see MENU RECORDS for definition).

SetMenuTitle

input: NewStrg:LONG - Address of string to replace ItemName.
MenuNum:WORD - menu ID.

output: None.

The value in NewStrg is moved into the menu's TitleName.

GetMenuTitle

input: MenuNum:WORD - menu ID.

output: TheTitle:LONG - pointer to TitleName.

Returns a pointer to the title of a menu.

SetMenuID

input: NewID:WORD - new ID to be assigned.
MenuNum:WORD - current menu ID.

output: None.

The menu is assigned the given ID number.

ITEM RECORD ACCESS ROUTINES

SetItem

input: NewStrg:LONG - Address of string to replace ItemName.
ItemNum:WORD - item ID.

output: None.

The item's ItemName pointer is replaced with NewStrg.

GetItem

input: ItemNum:WORD - item ID.

output: ItemStrg:LONG - pointer to ItemName.

Returns a pointer to an item's text string.

EnableItem

input: ItemNum:WORD - item ID.

output: None.

Item will appear as normal and selectable.

DisableItem

input: ItemNum:WORD - item ID.

output: None.

Item will appear dimmed and will not be selectable.

CheckItem

input: Checked:WORD - TRUE to check item, FALSE to uncheck item.
ItemNum:WORD - item ID.

output: None.

Item will appear with a check mark to the left of the item's text, or nothing will appear if Checked is zero.

SetItemMark

input: Mark:WORD - character to mark item with, zero for no mark.
ItemNum:WORD - item ID.

output: None.

Item will appear with the character given to the left of the item's text, or the mark will not appear if Mark is zero.

GetItemMark

input: ItemNum:WORD - item ID.

output: Mark:WORD - character that marks item, zero = no mark.

SetItemStyle

input: ChStyle:WORD - text style to use on item's text.
ItemNum:WORD - item ID.

output: None.

Bits in ChStyle are set to enable special text drawing. Bits affected are:

Bit 0	- set = **bold**, clear = no bold.
Bit 1	- set = *italic*, clear = no italic.
Bit 2	- set = underscore, clear = no underscore.
Bits 3-15	- zero.

Bits 0-2 of chStyle are all used to set the item's text style. You can not set only bold without affecting italic and underscore. If you need to change only one, perform a GetItemStyle first and use the current states for the attributes that should stay the same.

Note: Italic is not implemented in QuickDraw at the time of this document. Any menu items using the italic style will be italicized when using a new version of QuickDraw that has italic support.

Note: Underscore will only work with fonts that have a descent of two or more. The system font, that the time of this document, only has a descent of one and therefore will not be underscored.

GetItemStyle

input: ItemNum:WORD - item ID.

output: ChStyle:WORD - text style to use on item's text.

Bits in ChStyle are set to enable special text drawing. Bits affected are:

Bit 0	- set = **bold**, clear = no bold.
Bit 1	- set = *italic*, clear = no italic.
Bit 2	- set = underscore, clear = no underscore.
Bits 3-15	- undefined.

SetItemFlag

input: NewValue:WORD - new bits to set.
ItemNum:WORD - item ID.

output: None.

This call is used to set desire states of an item. Input flags are:

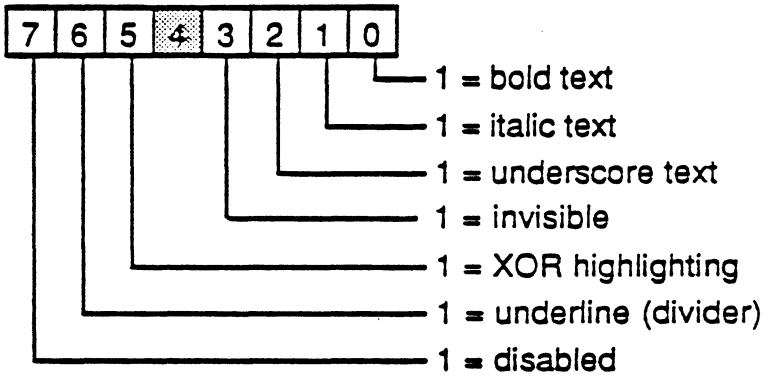
Function	NewValue
Underline item.	$0040
Not underline an item.	$FFBF
Use XOR highlighting.	$0020
Use redraw highlighting.	$FFDF

GetItemFlag

input: ItemNum:WORD - item ID.

output: itemFlag:WORD - item flag.

Bits in the low-order byte of itemFlag are:



The high-order byte of itemFlag is not defined and could be any value.

SetItemID

input: NewID:WORD - new ID to be assigned.
ItemNum:WORD - current item ID.

output: None.

The item is assigned the given ID number.

SetItemBlink

input: Count:WORD - number of times item should blink when selected.

output: None.

This call affects all menu bars, system and window. When an enabled item is selected by the user the item blinks briefly to conform the chioce. Normally, your application shouldn't be concerned with this blinking; the user sets it with the Control Panel desk accessory. If you're writing a desk accessory like the Control Panel. though, **SetItemBlink** allows you to control the duration of the blinking. The Count parameter is the number of times menu items will blink.

MISCELLANEOUS ROUTINES

GetMenuMgrPort

input: None.

output: MenuMgr:LONG - pointer to Menu Manager's port.

Getting the Menu Manager's port might be useful if you would like to change its font.

MNewRes

input: None.

output: None.

Called when the screen resolution changes. Menu Manager makes needed adjustments for the new resolution and redraws the current system menu bar.

InitPalette

input: None.
output: None.

Call when you've changed the color palattes. This will reinitize the palettes needed for the color Apple logo in the system menu bar.

Constants

Masks for MenuFlag:

M_INVIS	$04	FASLE if menu is visible *(not completed)*.
M_STANDARD	$10	FALSE if menu is a standard (not custom) menu.
M_NO_XOR	$20	TRUE if menu title is highlighted using XOR.
M_NORMAL	$40	TRUE if menu title is highlighted.
M_ENABLED	$80	FALSE if menu is disabled.

Commands to menu definition procedures:

mDrawMsg	0	Draw menu command.
mChooseMsg	1	Hit test item command.
mSizeMsg	2	Compute menu size command.
mDrawTile	3	Draw menu's title command.

Posiible inputs to **SetMenuFlag:**

EnableMenu	$FF7F	Menu will not be dimmed and will be selectable.
DisableMenu	$0080	Menu will be dimmed and not selectable.
ColorReplace	$FFDF	The menu's title and background will be redrawn to hilite.
XORhilite	$0020	The menu's title area will be XORed to hilite.
StandardMenu	$FFE7	The menu will be considered a standard menu.
CustomMenu	$0010	The menu will be considered a custom menu.

Possible NewValue input to **SetItemFlag:**

UnderItem	$0040	Underline item.
NoUnderItem	$FFBF	Do not underline item.
XORHilite	$0020	Use XOR highlighting on item.
NoXORHilite	$FFDF	Use redraw highlighting on item.

Data Types

TaskRec (TaskMaster record):

what	0	Integer	Same as event record.
message	2	LongInt	Same as event record.
when	6	LongInt	Same as event record.
where	10	LongInt	Same as event record.
modifiers	14	Integer	Same as event record.
TaskData	16	LongInt	Return for ID numbers.
TaskMask	20	LongInt	Not used by the Menu Manager.
TASKREC_SIZE	22		Size of TaskRec.

MENUBAR (Menu Bar Record):

CtlNext	0	Handle	Not used.
CtlOwner	4	Pointer	Pointer to menu bar's window.
CtlRect	8	RECT	Enclosing rectangle.
CtlFlag	16	Byte	Bit flags.
CtlHilite	17	Byte	Not used.
CtlValue	18	Integer	Not used.
CtlProc	20	Pointer	Not used.
CtlAction	24	Pointer	Not used.
CtlData	28	LongInt	Reserved for CtlProc's use.
CtlRefCon	32	LongInt	Reserved for application's use.
CtlColor	36	Pointer	Menu bar's color table.
MenuList	40	Handle[]	Menu bar's color table.

MENU (Menu record):

MenuID	0	Integer	Menu's ID number.
MenuWidth	2	Integer	Width of menu.
MenuHeight	4	Integer	Height of menu.
MenuProc	6	Pointer	Menu's definition procedure.
MenuFlag	10	Byte	Bit flags.
TitleWidth	11	Integer	Width of menu's title.
TitleName	13	Pointer	Menu's title.
(remainder is not defined)			

INDEX

Documentation Développeurs
Apple Computer France 1987

Document développeur numéro 57

Control Manager

type d'upgrade de ce ducument : 5
1 Documentation de première catégorie inchangée
2 Documentation de deuxième catégorie mise à jour
3 Documentation de deuxième catégorie inchangée
4 Mise à jour payante de la documentation de première catégorie
5 Mise à jour gratuite de la documentation de première catégorie
6 Nouveautés payantes non vitales
7 Nouveautés gratuites et vitales

Taille : 50 page(s) environ

Domaine : Tool 16

VERSION : 1.0
DATE : 25.09.86

CONTROL MANAGER

Dan Oliver

This ERS corresponds to the Beta release of the Control Manager, version 1.0. There will be no further changes to the Control Manager that will compromise applications written for this version. However, document errors will be changed to conform to actual code. Over the next few months I will be fixing bugs and releasing appendixes that detail Control Manager functions. If you have areas you would like clarified please let me know and I will try to publish an appendix. Send comments to:

Apple Computer, Inc.
20525 Mariani Ave, MS: 22X
Cupertino, CA 95014

ATTN: Dan Oliver

02/20/86 Initial release.

06/14/86 Revised release.

07/1586 Names changes; **BootCtrl, InitCtrlMgr, TermCtrlMgr,** to; **CtrlBootInit, CtrlStartup, CtrlShutDown.** Addition of **CtrlReset, CtrlStatus** and **GetCtrlzpage.**

07/16/86 Replacement for page 27, values for testCnlt and calcCRect were swapped.

08/13/86 Control record changed, CtrlAction added and CtrlHilite removed from CtrlFlag. The path to calling an action routine now includes CtrlAction. Color table colors corrected. **SetCMgrIcons** call added along with a CONTROL MANAGER ICON FONT section.

08/27/86 **SizeControl** removed. Calls added; **SetCtlAction, GetCtlAction, SetCtlRefCon, GetCtlRefCon, GrowSize.** Square corner added as an outline type for simple buttons, see simple button record.

Name changes:

From	To	From	To
CtrlBootInit	CtlBootInit	CTRL_VIS	CTL_VIS
CtrlVersion	CtrVersion	drawCtrl	drawCtl
CtrlReset	CtlReset	testCtrl	testCtl
CtrlStatus	CtlStatus	initCtrl	initCtl
CtrlStartup	CtlStartup	dispCtrl	dispCtl
CtrlShutDown	CtlShutDown	posCtrl	posCtl
CtrlNewRes	CtlNewRes	thumbCtrl	thumbCtl
GetCtrlzpage	GetCtlzpage	dragCtrl	dragCtl
SetCTitle	SetCtlTitle	moveCtrl	moveCtl
GetCTitle	GetCtlTitle	drawCntl	drawCtl
SetCMgrIcons	SetCtlIcons	testCntl	testCtl
		initCntl	initCtl
		dispCntl	dispCtl
		posCntl	posCtl
		thumbCntl	thumbCtl
		dragCntl	dragCtl
		moveCntl	moveCtl

'Ctl' will be the control abbreviation used in Cortland documents.

Tool Number: 16

Tools needed installed: Quick Draw
Memory Manager
Event Manager

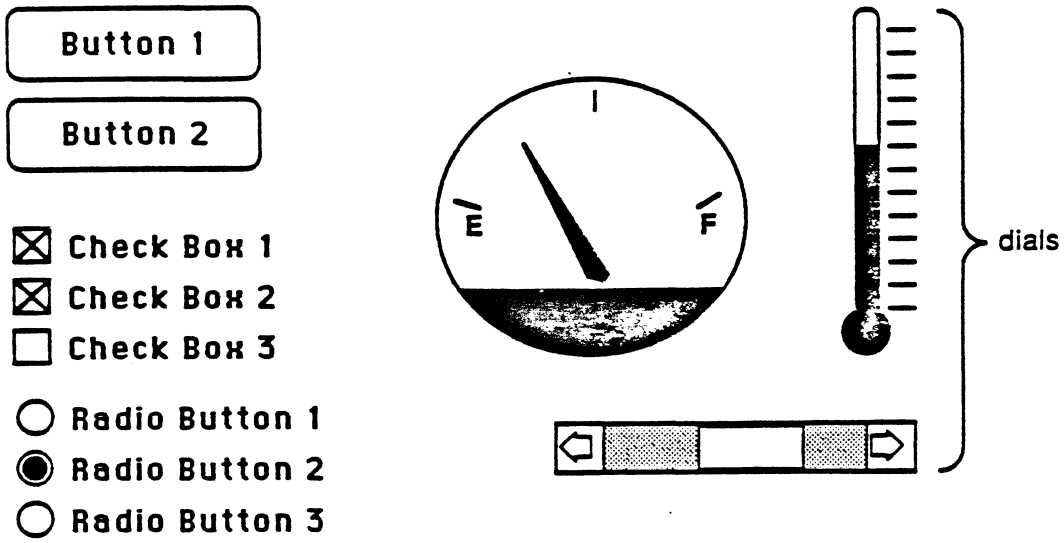
Stack requirement: 512 bytes.

The Control Manager is the part of the Cortland User Interface Toolbox that deals with controls. A control is an object on the Cortland screen with which the user, using the mouse, can cause instant action with graphic results or change settings to modify a future action. Using the Control Manager, your application can:

- display or hide controls
- monitor the user's operation of a control with the mouse and respond accordingly
- read or change the setting or other properties of a control
- change the size, location, or appearance of a control

Your application performs these actions by calling the appropriate Control Manager routines. The Control Manager carries out the actual operations, but it's up to you to decide when, where, and how.

Controls may be of various types, each with its own characteristic appearance on the screen and responses to the mouse. Each individual control has its own specific properties--such as its location, size, and setting--but controls of the same type behave in the same general way.



Certain standard types of controls are predefined for you. Your application can easily use controls of these standard types, and can also define its own "custom" control types. The predefined control types are the following:

- Buttons cause an immediate or continuous action when clicked or pressed with the mouse. They appear on the screen as rounded-corner rectangles with a title centered inside.

- Check boxes retain and display a setting, either checked (on) or unchecked (off); clicking with the mouse reverses the setting. On the screen, a check box appears as a small square with a title to the left of it; the box is either filled in with an "X" (checked) or empty (unchecked). Check boxes are frequently used to control or modify some future action, instead of causing an immediate action of their own.

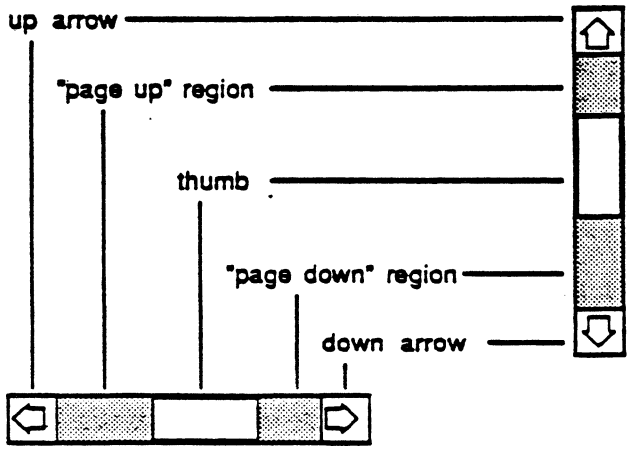
- Radio buttons also retain and display an on-or-off setting. They're organized into families, with the property that only one button in the family can be on at a time: clicking any button on turns off all the others in the family, like the buttons on a car radio. Radio buttons are used to offer a choice among several alternatives. On the screen, they look like round check boxes; the radio button that's on is filled with a small black circle instead of an "X".

  Note: The Control Manager knows which radio buttons belong in each family from the flag value passed to NewControl. Bits 8-14 of flag is the family number. Assign the same number for every member of a family, and different numbers for different families. Zero is an acceptable family number.

- Scroll bars are predefined dials. A dial displays a quantitative setting or value, typically in some pseudonanalog form such as the position of a sliding switch, the reading on a thermometer scale, or the angle of a needle on a gauge; the setting may be displayed digitally as well. The control's moving part that displays the current setting is called the

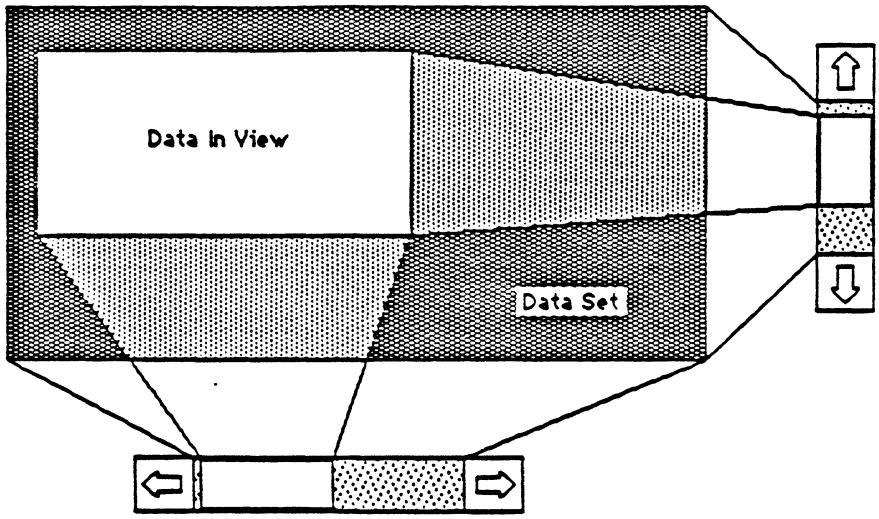
indicator. The user may be able to change a dial's setting by dragging its indicator with the mouse, or the dial may simply display a value not under the user's direct control (such as the amount of free space remaining on a disk).

The following diagram shows the parts of the vertical and horizontal scroll bars.



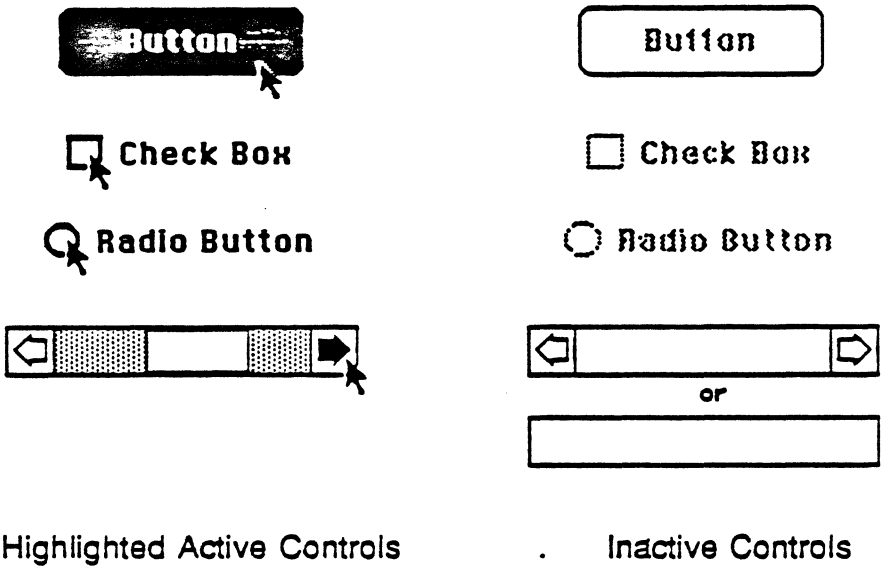
The parts of the scroll bars can be generalized into three regions; arrows, paging, and thumb (or thumber). The arrows scroll data a line at a time, paging regions scroll a "page" at a time, and the thumb can be dragged to any position within the scroll area. Although they may seem to behave like individual controls, these are all parts of a single control, the scroll bar type of dial. You can define other dials of any shape or complexity for yourself if your application needs them.

Standard scroll bars are proportional, that is they show the relationship between the total amount of data and the amount viewed, and where the view is in the data.


Data in View
Data Set

HIGHLIGHTING AND ACTIVE CONTROLS

When clicked or pressed, a control is usually highlighted. It's also possible for just a part of a control to be highlighted: for example, when the user presses the mouse button inside a scroll arrow in a scroll bar, the arrow, not the whole scroll bar, becomes highlighted.



Highlighted Active Controls . Inactive Controls

A control may be active or inactive. Active controls respond to the user's mouse actions; inactive controls don't. A control is made inactive when it has no meaning or effect in the current context, such as an "Open" button when no document has been selected to open, or a scroll bar when there's currently nothing to scroll to. An inactive is highlighted in some special way, depending on its control type. For example, the title of an inactive button, check box, or radio button is dimmed.

There are two ways a scroll bar can be made inactive. In the diagram above the top scroll is made inactive by making the data size equal to or smaller than the view size. This type of inactive state happens automatically when the data size equals or exceeds the view size. The bottom inactive scroll bar is made by passing 255 to HiliteControl and is inactive in the same sense that the other controls are inactive.

There is one more way in which controls can be made inactive, make them invisible. Invisible controls are inactive in the sense that it can not be selected. However its 'highlighting in some special way' is an extreme.

CONTROLS AND WINDOWS

Every control "belongs" to a window: When displayed, the control appears within that window's content region; when manipulated with the mouse, it acts on that window. All coordinates pertaining to the control (such as those describing its location) are given in its window's local coordinate system. Even the state of the control can be tied to the state of the window. A bit in wFrame of the window's record can be set so the controls in the window will be considered inactive if the the window is inactive. See the Window Manager ERS.

Warning: In order for the Control Manager to draw a control properly, the control's window must have the top left corner of its grafPort's portRect as coordinates (0,0). If you change a window's local coordinate system for any any reason (with the QuickDraw procedure **SetOrigin**), be sure to change it back-so that the top left corner is again at (0,0)-before drawing any controls. Since almost all of the Control Manager routines can (at least potentially) redraw a control, the safest policy is simply to change the coordinate system back before calling any Control Manager routine.

However: If you would like to have controls in a window scroll with the content region there is a way. Before call the Control Manager make sure the origin of the control's window is set to its scrolled value

PART CODES

Some controls, such as buttons, are simple and straightforward. Others can be complex objects with many parts: for example, a scroll bar may have two scroll arrows, two paging regions, and a thumb. To allow different parts of a control to respond to the mouse in different ways, many of the Control Manager routines accept a part code as a parameter or return one as a result.

A part code is a number between 1 and 253 that stands for a particular part of a control. Each type of control has its set of part codes. Some of the Control Manager routines need to give special treatment to the indicator of a dial (such as the thumb of a scroll bar). To allow the Control Manager to recognize such indicators, they always have part codes greater than 127.

The part codes are assigned as follows:

0	No part.
1	Reserved for internal use.
2	Simple button.
3	Check box.
4	Radio button.
5	Up arrow.
6	Down arrow.
7	Page up.
8	Page down.
9	Reserved for internal use.
10	Grow box icon.
11-31	Reserved for internal use.
32-127	Reserved for application's use.
128	Reserved for internal use.
129	Thumb.
130-159	Reserved for internal use.
160-253	Reserved for application's use.
254-255	Reserved for internal use.

USING THE CONTROL MANAGER

This section discusses how the Control Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use. The routines themselves are described in detail in CONTROL MANAGER ROUTINES, and examples are found in PROGRAMMING EXAMPLES.

To use the Control Manager, you must have previously called InitGraf to initialize QuickDraw and InitFonts to initialize the Font Manager if you are going to use controls with text in them.

> **Note:** For controls in dialogs or alerts, the Dialog Manager makes some of the basic Control Manager calls for you. Also, the Window Manager will make Control Manager calls concerning standard window controls.

Where appropriate in your program, use **NewControl** to add any controls you need. **NewControl** will set the control's owner to the window pointer passed and add the control to the head of the window's control list. When you no longer need a control, call **DisposeControl** to remove it from its window's control list and erase it from the screen. To dispose of all a window's controls at once, use **KillControls**.

> **Note:** The Window Manager procedure **CloseWindow** will automatically dispose of all the controls associated with the given window.

When the Event Manager function **GetNextEvent** reports that an update event has occurred for a window, the application should call **DrawControls** to redraw the window's controls as part of the process of updating the window.

After receiving a mouse-down event from **GetNextEvent**, do the following:

1. First call **FindWindow** to determine which part of which window the mouse button was pressed in. If it was in the content region of the active window, use that window's control list.

2. If the event did occur in a content area call **FindControl** with the pointer to the window to find out whether the event occurred on an active control.

3. Finally, if **FindControl** returns a control handle, call **TrackControl** to handle user interaction with the control. **TrackControl** will handle the highlighting of the control and determines whether the mouse is still in the control when the mouse button is released. It also handles the dragging of the thumb in a scroll bar and responds to presses or clicks in the other parts of a scroll bar. When **TrackControl** returns the part code for valid control, the application must do whatever is appropriate as a response.

The application's exact response to mouse activity in a control that retains a setting will depend on the current setting of the control, which is available from the **GetCtlValue** function. For controls whose values can be set by the user, the **SetCtlValue** procedure may be called to change the control's setting and redraw the control accordingly. You'll call **SetCtlValue**, for example, when a check box or radio button is clicked, to change the setting and draw or clear the mark inside the control.

Wherever needed in your program, you can call **HideControl** to make a control invisible or **ShowControl** to make it visible. Similarly, **MoveControl**, which simply changes a control's location without pulling around an outline of it, can be called at any time, as can **SizeControl**, which changes its size. For example, when the user changes the size of a document window that contains a scroll bar, you'll call **HideControl** to remove the old scroll bar, **MoveControl** and **SizeControl** to change its location and size, and **ShowControl** to display it as changed. Whenever necessary, you can read various attributes of a control with **GetCTitle**, **GetCtlParams**, or **GetCtlState**; you can change them with **SetCTitle**, **SetCtlParams**, or **SetCtlState**.

CONTROL MANAGER ICON FONT

The standard control definition procedures use a font to draw some control parts and their highlighted states. If you would like to use different icons you can replace the default font. To replace the icon font, or just get the handle to the current font, call **SetCMgrIcons**. The format of the font is as follows:

Character 0	Check box that is off and not highlighted.
Character 1	Check box that is off and is highlighted.
Character 2	Check box that is on and not highlighted.
Character 3	Check box that is on and not highlighted.
Character 4	Radio button that is off and not highlighted.
Character 5	Radio button that is off and is highlighted.
Character 6	Radio button that is on and not highlighted.
Character 7	Radio button that is on and is highlighted.
Character 8	Right arrow that is not highlighted.
Character 9	Right arrow that is highlighted.
Character 10	Left arrow that is not highlighted.
Character 11	Left arrow that is highlighted.
Character 12	Up arrow that is not highlighted.
Character 13	Up arrow that is highlighted.
Character 14	Down arrow that is not highlighted.
Character 15	Down arrow that is highlighted.
Character 16	Grow icon.

CONTROL RECORDS

Every control has the same front end to its control record. Additional data can then be appended to the end of the general control record. For example, **NewControl** will call the control's definition procedure to find out the size of the record to allocate, before the record is actually allocated. The **General Control Record** follows:

CtlNext	LONG	Handle to next control, zero = last control.
CtlOwner	LONG	Pointer to window the control belongs to.
CtlRect	RECT	Enclosing rectangle.
CtlFlag	BYTE	Flags that define the control, bit 7 = 0 if visible, 1 if invisible.
CtlHilite	BYTE	Currently highlighted part.
CtlValue	WORD	Current value.
CtlProc	LONG	Address of control's definition procedure.
CtlAction	LONG	Address of control's default action procedure.
CtlData	LONG	Data used by definition procedure.
CtlRefCon	LONG	Reserved for application's use only.
CtlColor	LONG	Pointer to control's color table, zero for default table.

CtlNext is a handle to the next control associated with this control's window. All the controls belonging to a given window are kept in a linked list, beginning in the wcontrol field of the window record and chained together through the CtlNext fields of the individual control records. The end of the list is marked by a zero value; as new controls are created, they're added to the beginning of the list.

CtlOwner is a pointer to the window port that this control belongs to.

CtlRect is the rectangle that completely encloses the control, in the local coordinates of the control's window.

CtlFlag is a bit vector that further describes the control. The only bit that is used for all controls is bit 7. Bit 7 is 0 if the control is visible, and 1 if invisible. Bits 0-6 are reserved for the control's definition procedure.

CtlHilite specifies whether and how the control is to be highlighted, indicating whether it's active or inactive. The value is zero if the control is active and has no highlighted parts. The value is 255 if the control is inactive. If the value is between 1 and 254, it's the part code of a highlighted part of the control. Therefore, only one part on a control can be highlighted at any one time, and no part can be highlighted on an inactive control. See **HiliteControl** for more information.

CtlValue is the control's current setting. For check boxes and radio buttons, 0 means the control is off and non-zero means it's on. For scroll bars, the value is between 0 and data size less view size. Custom controls can use the field as they see fit.

CtlProc is the address of the control definition procedure for this type of control. Standard controls do not use an address in this field. Instead, bits 0-23 are zero and only the high-order byte is used to determine which standard control the control is. Values for standard controls are:

$00000000	Simple button.
$02000000	Check box.
$04000000	Radio button.
$06000000	Scroll bar.

CtlData is reserved for use by the control definition procedure, typically to hold additional information specific to a particular control type. For example, the standard definition procedure for scroll bars uses the low-order word as view size, and the high-order word as data size. The standard definition procedures for simple buttons, check boxes, and radio buttons store the address of the control's title here.

CtlAction is a address of the control's default action procedure, if any. The procedure **TrackControl** may call the default action procedure to respond to the user's dragging the mouse inside the control. See **TrackControl** for more information.

CtlRefCon is the control's reference value field, which the application may store into and access for any purpose.

CtlColor is a pointer to the control's color table, which is used by the control's definition procedure to draw the control.

More fields can be added to the end of the control record to further define the control. See the control record of a standard scroll bar as an example.

The following are how control record fields are used by standard controls:

Simple Button:

CtlFlag bits 0-1
0 = single, round corner, outline.
1 = bold, round corner, outline.
2 = single, square corner, outline.
3 = single, square corner, outline with drop shadow.

CtlValue is always zero.

CtlProc equals $00000000.

CtlData is a pointer to the button's title string.

CtlColor is a pointer to control's color table, or zero for default table. The simple button color table is defined as:

color1 = outline color when normal, in high nibble.
color2 = interior color when normal.
color3 = interior color when selected.
color4 = text color when normal.
color5 = text color when selected.
color6 = special highlight color.
color7 = thick outline color.

A simple button can be drawn with one of two outlines. The thick outline should be used with buttons that would be selected by the user pressing Return on the keyboard. This would be a default key and should never cause a the destruction of something, like a default button "Delete File". The thin outline should be used for all other simple buttons.

Check Box:

CtlValue is 0 if not checked, non-zero if checked.

CtlProc equals $02000000.

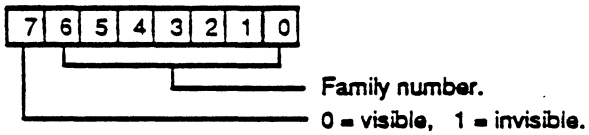
CtlData is a pointer to the check box's title string.

CtlColor is a pointer to control's color table, or zero for default table. The check box color table is defined as:

color1 = not used.
color2 = color of check box when not highlighted.
color3 = color of check box when highlighted.
color4 = color of title.

Radio Button:

CtlFlag is:



CtlValue is 0 if off, non-zero if on.

CtlProc equals $04000000.

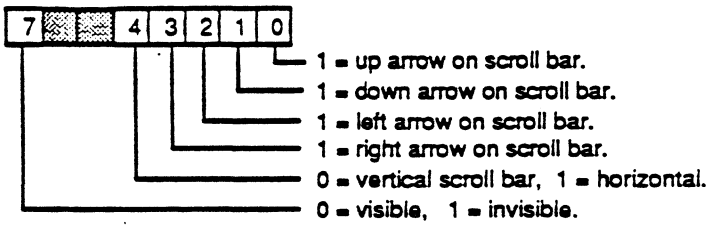
CtlData is a pointer to the radio button's title string.

CtlColor is a pointer to control's color table, or zero for default table. The radio button color table is defined as:

color1 = not used.
color2 = color of radio button when not highlighted.
color3 = color of radio button when highlighted.
color4 = color of title.

Scroll Bar:

CtlFlag is defined as:



CtlValue equals a number between zero and data size less view size.

CtlProc equals $06000000.

CtlData low-order WORD equals view size, and high-order word equals data size.

CtlColor is a pointer to control's color table, or zero for default table. The radio button color table is defined as:

color1 = outline color.
color2 = arrow color when normal.
color3 = arrow color when selected.
color4 = arrow box interior color.
color5 = thumber interior color when normal.
color6 = thumber interior color when selected.
color7 = page region's color, low nibble = background.
color8 = inactive color.

Additional data fields appended to the end of the control record:

Thumb	RECT	Thumber rectangle.
PageRegion	RECT	Page region, thumb's bounds.

CONTROL MANAGER ROUTINES

INITIALIZATION AND TERMINATION

CtlBootInit

input: None.

output: None.

Called only by the loader when loaded.

CtlVersion

input: None.

output: Version:WORD — Version number of the Control Manager.

CtlReset

input: None.

output: None.

Called on system reset.

CtlStatus

input: None.

output: status:WORD - TRUE if Control Manager is active, FALSE if not.

CtlStartup

input:	yourID:WORD	Your ID number, used for memory allocation.
	zeroPage:WORD	Zero page Control Manager can use.
output:	None.	

(2 pages)

CtlStartup allows the Control Manager to perform start up initialization. YourID will be used by the Control Manager when it allocates memory. ZeroPage is an address of a page (256 bytes) in bank zero that your application makes available to the Control Manager for its use. The page does not have to be page aligned, but the Control Manager will operate faster if it is.

CtlShutDown

input: None.

output: None.

Deactivates the Control Manager. No controls are disposed of, **CloseWindow** in the Window Manager disposes of all controls in a window. Therefore, the Control Manager should not be shutdown until after the Window Manager has been shutdown.

CtlNewRes

input: None.

output: None.

Call **CtlNewRes** after you have changed the video mode. This routine will reinitialize resolution and mode dependencies.

NewControl

input:	theWindow:LONG	Pointer to window owner.
	boundsRect:LONG	Pointer to enclosing RECT.
	title:LONG	Pointer to title string (CtlData).
	flag:WORD	Bit vector of flags.
	value:WORD	Control's starting value.
	param1:WORD	Additional parameter (view size for scroll bars).
	param2:WORD	Additional parameter (date size for scroll bars).
	defProc:LONG	Address of definition procedure, or standard.
	refCon:LONG	Any value you want, application reserved.
	colorTable:LONG	Pointer to control's color table.
output:	ControlHandle:LONG	Control's handle, zero if error.

NewControl creates a control, adds it to the beginning of theWindow's control list, and returns a handle to the new control. The values passed as parameters are stored in the corresponding fields of the control record, as described below. The field that determines highlighting is set to 0 (no highlighting).

> **Note:** The control definition function may do additional initialization, including changing any of the fields of the control record. The only standard control for which additional initialization is done is the scroll bar; its control definition procedure computes the thumber and page region from boundsRect and flag.

TheWindow is the window the new control will belong to. All coordinates pertaining to the control will be interpreted in this window's local coordinate system.

BoundsRect, given in theWindow's local coordinates, is the rectangle that encloses the control and thus determines its size and location. Note the following about the enclosing rectangle for the standard controls:

- Simple buttons are drawn to fit the rectangle exactly. (The control definition function calls the QuickDraw procedure FrameRoundRect.) To allow for the tallest characters in the system font, there should be at least a 20-point difference between the top and bottom coordinates of the rectangle.
- For check boxes and radio buttons, there should be at least a 16-point difference between the top and bottom coordinates.
- A standard scroll bar should be at least 48 pixels long, to allow room for the scroll arrows and thumb.

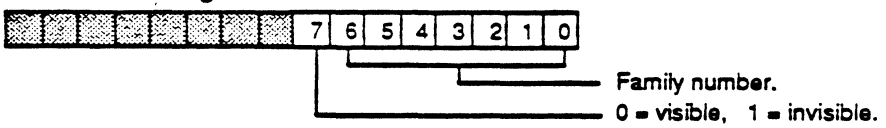
Title is the control's title, if any (if none, you can just pass the empty string as the title). Be sure the title will fit in the control's enclosing rectangle; if it won't it may not be completely erase with **HideControl**, along with other possible side effects.

Flag is a bit vector that further defines the control. Bit 7 is a visible/invisible flag for every kind of control. Bits 8-15 can be set to $FFxx to make the control inactive, but should be normally set to zero for an active control. Bits 0-6 are defined by each type of control. The bit vectors are defined below for standard controls.

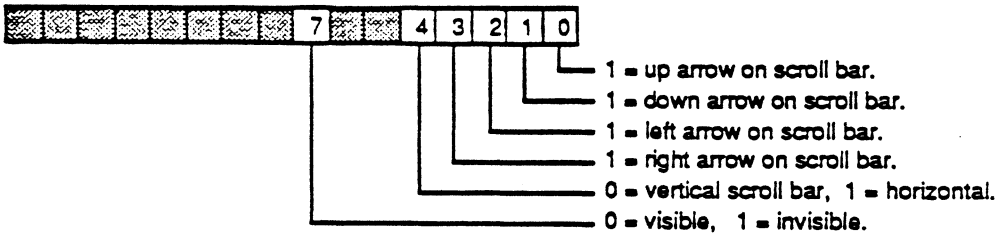
Simple button flag:

Check box flag:

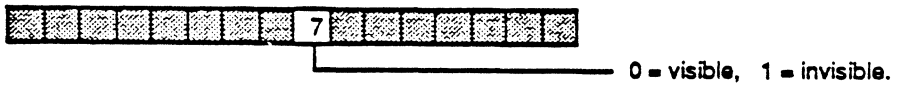
Radio button flag:



Scroll bar flag:



Grow box flag:



The value parameter gives the control an initial setting.

The param1 and param2 parameters are defined by the control's definition procedure. For standard scroll bars param1 is the size of the view, and param2 is the total data size. The standard scroll bar definition procedure will store the value of param1 in the CtlData field, and param2 in CtlData+2 field.

DefProc is the address of the control's definition procedure. DefProcs for custom control types are discussed later under "Defining Your Own Controls". The values for the standard control types are:

$00000000	- Simple button.
$02000000	- Check box.
$04000000	- Radio button.
$06000000	- Scroll bar.

RefCon is the control's reference value, set and used only by your application.

DisposeControl

input: theControl:LONG Handle of control.
output: None.

DisposeControl removes theControl from the screen, deletes it from its window's control list, and releases the memory occupied by the control record and any data structures associated with the control. **DisposeControl** does not erase the control, call **HideControl** prior to **DisposeControl** if the control must be erased.

KillControls

input: theWindow:LONG Pointer to window.
output: None.

KillControls disposes of all controls associated with theWindow by calling **DisposeControl** (above) for each control in theWindow's control list. **KillControls** does not erase the controls, call **HideControl** for each control in the list prior to KillControls if the they must be erased.

Note: Remember that the Window Manager procedures **CloseWindow** automatically dispose of all controls associated with the given window.

CONTROL DISPLAY

These procedures affect the appearance of a control but not its size or location.

SetCtlTitle

input:	title:LONG	Address of new title.
	theControl:LONG	Handle of control.
output:	None.	

SetCTitle sets theControl's title to the given string and redraws the control.

GetCtlTitle

input:	theControl:LONG	Handle of control.
output:	title:LONG	Pointer to control's title.

GetCTitle returns the value in theControl's CtlData field, which, for controls with titles, is the pointer to the control's title string.

HideControl

input:	theControl:LONG	Handle of control.
output:	None.	

HideControl makes theControl invisible. It fills the region the control occupies within its window with the background pattern of the window's grafPort. It also adds the control's enclosing rectangle to the window's update region, so that anything else that was previously obscured by the control will reappear on the screen. If the control is already invisible, HideControl has no effect.

ShowControl

input:	theControl:LONG	Handle of control.
output:	None.	

ShowControl makes theControl visible. The control is drawn in its window but may be completely or partially obscured by overlapping windows or other objects. If the control is already visible, ShowControl has no effect.

DrawControls

input: theWindow:LONG — Pointer to window, of which the control list is drawn.

output: None.

DrawControls draws all controls currently visible in theWindow. The controls are drawn in reverse order of creation; thus in case of overlap the earliest-created controls appear front most in the window.

> Note: Window Manager routines such as **SelectWindow**, **ShowWindow**, and **BringToFront** do not automatically call **DrawControls** to display the window's controls. They just add the appropriate regions to the window's update region, generating an update event. Your program should always call **DrawControls** explicitly upon receiving an update event for a window that contains controls.

HiliteControl

input: hiliteState:WORD — Operation to perform.
theControl:LONG — Handle of control.

output: None.

HiliteControl changes the way theControl is highlighted. HiliteState has one of the following values:

- The value 0 means no highlighting and the control is active. Any highlighted part of the control is unhighlighted. If the control is inactive, it's changed to active and redrawn.
- A value between 1 and 253 is interpreted as a part code designating the part of the (active) control to be highlighted.
- The value 255 means that the control is to be made inactive and redrawn accordingly.

Note: The value 254 should not be used; this value is reserved for future use.

HiliteControl calls the control definition function to redraw the control with its new highlighting.

MOUSE LOCATION

FindControl

input:	FoundCtl:LONG	Address of where to store control handle.
	xPoint:WORD	X coordinate, in global coordinates, to check.
	yPoint:WORD	Y coordinate, in global coordinates, to check.
	theWindow:LONG	Pointer of window to check.
output:	FoundPart:WORD	Part code of found part on control.

When the Window Manager function **FindWindow** reports that the mouse button was pressed in the content region of a window, and the window contains controls, the application should call **FindControl** with theWindow equal to the window pointer and thePoint equal to the point where the mouse button was pressed (in the window's global coordinates). **FindControl** tells which of the window's controls, if any, the mouse button was pressed in:

- If it was pressed in a visible, active control, **FindControl** sets the whichControl parameter to the control handle and returns a part code identifying the part of the control that it was pressed in.

- If it was pressed in an invisible or inactive control, or not in any control, **FindControl** sets whichControl to NIL and returns 0 as its result.

Note: **FindControl** also returns zero for whichControl and zero as its result if the window is invisible or doesn't contain the given point. In these cases, however, **FindWindow** wouldn't have returned this window in the first place, so the situation should never arise.

TestControl

input:	xPoint:WORD	X coordinate, in local coordinates, to check.
	yPoint:WORD	Y coordinate, in local coordinates, to check.
	theControl:LONG	Handle of control.
output:	PartCode:WORD	Part thePoint is over.

If theControl is visible and active, **TestControl** tests which part of the control contains thePoint (in the local coordinates of the control's window); it returns the corresponding part code, or zero if the point is outside the control. If the control is invisible or inactive, **TestControl** returns zero. **TestControl** is called by **FindControl** and **TrackControl**; normally you won't need to call it yourself.

TrackControl

input:	startX:WORD	X coordinate, in global coordinates, of starting point.
	startY:WORD	Y coordinate, in global coordinates, of starting point.
	actionProc:LONG	Address of routine, zero, or a negative number.
	theControl:LONG	Handle of control.
output:	PartCode:WORD	Selected part when button was released.

When the mouse button is pressed in a visible, active control, the application should call **TrackControl** with theControl equal to the control handle and startY and startX are equal to the point where the mouse button was pressed (in the global coordinates). **TrackControl** follows the movements of the mouse and responds in whatever way is appropriate until the mouse button is released; the exact response depends on the type of control and the part of the control in which the mouse button was pressed. If highlighting is appropriate, **TrackControl** does the highlighting, and will undo it before returning. When the mouse button is released, **TrackControl** returns with the part code if the mouse is in the same part of the control that it was originally in, or with zero if not (in which case the application should do nothing).

If the mouse button was pressed in an indicator, **TrackControl** drags a dotted outline of it to follow the mouse. When the mouse button is released, **TrackControl** calls the control definition procedure to reposition the control's indicator. The control definition function for scroll bars responds by redrawing the thumb, calculating the control's current setting based on the new relative position of the thumb, and storing the current setting in the control record. The application must then scroll to the corresponding relative position in the document.

TrackControl may take additional actions beyond highlighting the control or dragging the indicator, depending on the value passed in the actionProc parameter, as described below. The following tells you what to pass for the standard control types; for a custom control, what you pass will depend on how the control is defined.

- If actionProc is zero, **TrackControl** performs no additional actions. This is appropriate for simple buttons, check boxes, radio buttons, and the thumb of a scroll bar.

- ActionProc may be a pointer to an action procedure that defines some action to be performed repeatedly for as long as the user holds down the mouse button. (See below for details.)

- If actionProc is a negative number, **TrackControl** will check the CtlAction field of the control's record. No additional actions will be performed if CtlAction is zero. If CtlAction is negative, the control's definition procedure will be called with an autoTrack message. If CtlAction is neither zero or negative, it will be considered a valid address of an action routine and be called.

The action procedure in the control definition procedure is described in the section "Defining Your Own Controls". The action procedure should be of the form:

```
MyAction

     inputs:  partCode:WORD     Selected part.
              theControl:LONG   Handle of control.

     outputs: None.
```

In this case, **TrackControl** passes the control handle and the part code to the action procedure. (It passes zero in the partCode parameter if the mouse has moved outside the original control part.) As an example of this type of action procedure, consider what should happen when the mouse button is pressed in a scroll arrow or paging region in a scroll bar. For these cases, your action procedure should examine the part code to determine exactly where the mouse button was pressed, scroll up or down a line or page as appropriate, and call **SetCtlValue** to change the control's setting and redraw the thumb.

CONTROL MOVING AND SIZING

MoveControl

input:	NewX:WORD	New X origin of control.
	NewY:WORD	New Y origin of control.
	theControl:LONG	Handle of control.
output:	None.	

MoveControl moves theControl to a new location within its window. The top left corner of the control's enclosing rectangle is moved to the horizontal and vertical coordinates h and v (given in the local coordinates of the control's window); the bottom right corner is adjusted accordingly, to keep the size of the rectangle the same as before. If the control is currently visible, it's hidden and then redrawn at its new location.

DragControl

input:	startX:WORD	X coordinate, in local coordinates, of starting point.
	startY:WORD	Y coordinate, in local coordinates, of starting point.
	limitRect:LONG	Pointer to bounds rectangle.
	slopRect:LONG	Pointer to slop rectangle.
	axis:WORD	Movement constraint.
	theControl:LONG	Handle of control.
output:	None.	

Called with the mouse button down inside theControl, **DragControl** pulls a dotted outline of the control around the screen, following the movements of the mouse until the button is released. When the mouse button is released, **DragControl** calls **MoveControl** to move the control to the location to which it was dragged.

> Note: Before beginning to follow the mouse, **DragControl** calls the control definition function to allow it to do its own "custom dragging" if it chooses. If the definition function doesn't choose to do any custom dragging, **DragControl** uses the default method of dragging described here.

The startX, startY, limitRect, slopRect, and axis parameters have the same meaning as for the procedure **DragRect.**, see **DragRect.**

CONTROL RECORD ACCESS

SetCtlValue

input:	CurValue:WORD	Current value of control.
	theControl:LONG	Handle of control.
output:	None.	

SetCtlValue sets theControl's current setting to theValue and redraws the control to reflect the new setting. For check boxes and radio buttons, the value 1 fills the control with the appropriate mark, and zero clears it. For scroll bars, **SetCtlValue** redraws the thumb where appropriate.

If the specified value is out of range, it's forced to the nearest endpoint of the current range.

GetCtlValue

input:	theControl:LONG	Handle of control.
output:	CurValue:WORD	Control's current value.

GetCtlValue returns theControl's current setting.

SetCtlAction

input:	newAction:LONG	Pointer to control's action procedure.
	theControl:LONG	Handle of control.
output:	None.	

SetCtlAction set theControl's CtlAction field to newAction.

GetCtlAction

input:	theControl:LONG	Handle of control.
output:	Action:LONG	Value in theControl's CtlAction field.

SetCtlRefCon

input:	newRefCon:LONG	Value to store in theControl's CtlRefCon field.
	theControl:LONG	Handle of control.
output:	None.	

SetCtlRefCon set theControl's CtlRefCon field to newRefCon. This field is reserved for the application's use only and will not be used or changed (except for this call) by the Control Manager.

GetCtlRefCon

input:	theControl:LONG	Handle of control.
output:	refCon:LONG	Value in theControl's CtlRefCon field.

GetCtlRefCon set theControl's CtlRefCon field to newRefCon. This field is reserved for the application's use only and will not be used or changed by the Control Manager.

GetCtlParams

input:	theControl:LONG	Handle of control.
output:	params:LONG	TheControl's CtlData field value.

GetCtlParams returns the off theControl's CtlData field.

SetCtlParams

input:	param2:WORD	Additional control parameter, defined by control.
	param1:WORD	Additional control parameter, defined by control.
	theControl:LONG	Handle of control.
output:	None.	

SetCtlParams is a way of setting new parameters to the control's definition procedure, which will set the values and redraw the control if necessary. Simple buttons, check boxes, and radio buttons, do not use param1 or param2, and no action is performed.

Of the predefined controls, only scroll bars use the parameters. Param1 is used as the scroll bar's view, and param2 the data size. If, for either param1 or param2, a -1 is passed, that parameter will not be changed (this only applicable to predefined scroll bars, custom controls may not support this feature). Example:

> You want to show an editable text document, with a single vertical scroll bar to the right of the text. The text document has 300 lines, of which 30 can be displayed at one time. To set the scroll bar you would pass 30 for param1 and 300 for param2.
>
> If the user enters a line you would want to update the scroll bar. So, you pass -1 for param1 because there was no change in the view (although for predefined scroll bar there is no advantage to passing the view size again rather than -1), and 301 for param2 to show the increased data size.
>
> For this same document there is another approach you could take. You could pass the view and data sizes as pixels. If every line is 10 pixels high, counting leading, and there were 300 lines, of which 30 can be displayed, you would pass 300 for param1 and 3000 for param2. After the line was entered, you'd pass -1 for param1 (or 300 again), and 3010 for param2. Because passing the number of pixels, rather than the number of lines, is proportionally equivalent, the scroll bar will be identical for either method.

Miscellaneous Routines

DragRect

input:	actionProc:LONG	Address of routine, zero, or a negative number.
	dragPattern:LONG	Address of pattern to use for drag outline.
	startX:WORD	X coordinate, in local coordinates, of starting point.
	startY:WORD	Y coordinate, in local coordinates, of starting point.
	dragRect:LONG	Pointer to rectangle to be dragged.
	limitRect:LONG	Pointer to bounds rectangle.
	slopRect:LONG	Pointer to slop rectangle.
	axis:WORD	Movement constraint.
output:	MoveDelta:LONG	Low WORD is the amount Y changed, High WORD is the amount X changed.

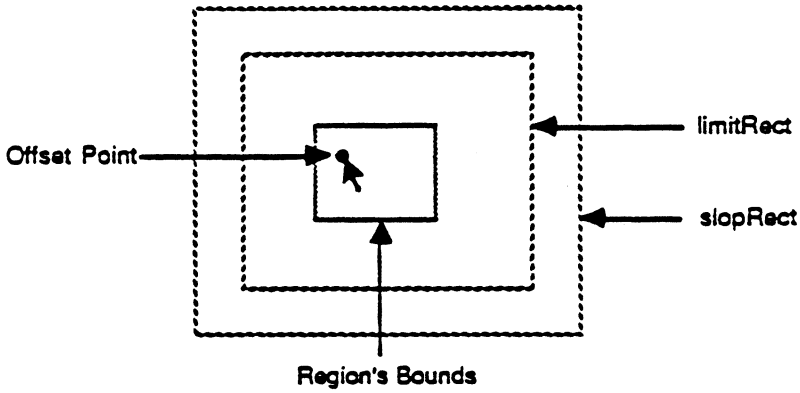
Called when the mouse button is down inside dragRect, **DragRect** drags a dotted (gray) outline of the RECT's bounds, which should be in global coordinates, following the movements of the mouse until the button is released.

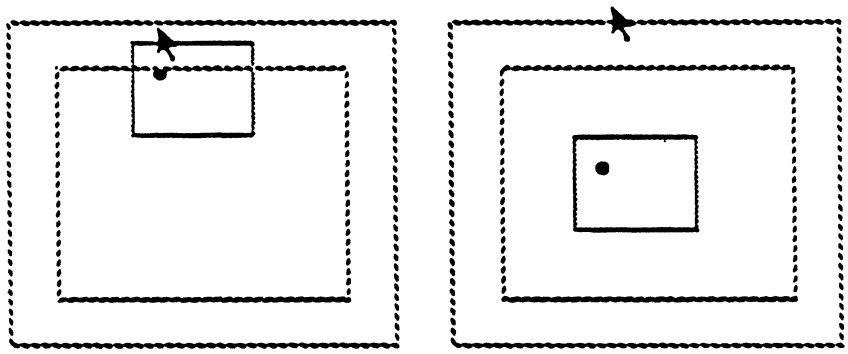
The startY,startX parameters are assumed to be the point where the mouse button was originally pressed, in the global coordinates.

LimitRect and slopRect are also in global coordinates. To explain these parameters, the concept of "offset point" must be introduced: This is initially the point whose vertical and horizontal offsets from the top left corner of the region's enclosing rectangle are the same as those of startY,startX. The offset point follows the mouse location, except that **DragRect** will never move the offset point outside limitRect; this limits the travel of the region's outline (but not the movements of the mouse). SlopRect, which should completely enclose limitRect, allows the user some "slop" in moving the mouse. **DragRect**'s behavior while tracking the mouse depends on the location of the mouse with respect to these two rectangles:

- When the mouse is inside limitRect, the RECT's outline follows it normally. If the mouse button is released there, the RECT should be moved to the mouse location.
- When the mouse is outside limitRect but inside slopRect, **DragRect** "pins" the offset point to the edge of limitRect. If the mouse button is released there, the region should be moved to this pinned location.
- When the mouse is outside slopRect, the outline disappears from the screen, but **DragRect** continues to follow the mouse; if it moves back into slopRect, the outline reappears. If the mouse button is released outside slopRect, the region should not be moved from its original position.

The diagrams below illustrates what happens when the mouse is moved outside limitRect but inside slopRect, and outside the slopRect.





The top diagram shows the starting position. As the cursor is moved, an outline of the window is dragged with it. The outline will seem to be glued to the cursor at the offset point. However, if the cursor moves outside of the limitRect, the outline will be left behind, as shown in the lower left diagram. As the cursor is moved outside of the limitRect, but within the slopRect, the outline will get as close to the cursor as possible without letting the offset point leave the limitRect. And finally, if the cursor moves outside the slopRect, the outline will snap back to its starting position. If the cursor moves back into the slopRect, the outline will snap out to get as close as it can.

If the mouse button is released within slopRect, the high-order word of the value returned by **DragRect** contains the vertical coordinate of the ending mouse location minus that of startY,startX and the low-order word contains the difference between the horizontal coordinates. If the mouse button is released outside slopRect, both words are zero.

The axis parameter allows you to constrain the region's motion to only one axis. It has one of the following values:

```
CONST  noConstraint  = 0    {no constraint}
       hAxisOnly     = 1    {horizontal axis only}
       vAxisOnly     = 2    {vertical axis only}
```

If an axis constraint is in effect, the outline will follow the mouse's movements along the specified axis only, ignoring motion along the other axis.

The actionProc parameter is a pointer to a procedure that defines some action to be performed repeatedly for as long as the user holds down the mouse button; the procedure should has no parameters. If actionProc is NIL, **DragRect** simply retains control until the mouse button is released.

GetCtlzpage

input: None.

output: CtlZPage:WORD - Control Manger's direct (zero) page.

This call will normally only be made by the Dialog Manager. The Dialog Manager makes this call because the Control and Dialog Managers share a single direct page.

GrowSize

input: None.

output: SizeOfGrow:LONG - low-order word is height, high-order word is width.

GrowSize returns the height and width, using the Control Manager's current icon font, of the grow box control. The height of the grow box is returned in the low-order word of SizeOfGrow, and the width in the high-order word.

SetCtlIcons

input: newFont:LONG - handle of new icon font, negative to not set new font.

output: oldFont:LONG - handle of current icon font (before newFont is set).

See CONTROL MANAGER ICON FONT for more information about the icon font.

DEFINING YOUR OWN CONTROLS

In addition to predefined controls, you can also define "custom" controls of your own. Maybe you need a three-way selector switch, a memory-space indicator that looks like a thermometer, or a thruster control for a spacecraft simulator—whatever your application needs. Controls and their indicators may occupy regions of any shape.

To define your own type of control, you write a control definition procedure in your application. The Control Manager stores this address in the CtlProc field of the control record. Later, when it needs to perform a type-dependent action on the control, it calls the control definition procedure.

The Control Definition Procedure

The inputs and output of the definition procedure are:

input:	message:WORD	Desired operation.
	param:LONG	Depends on operation.
	theControl:LONG	Handle of control.
output:	RetValue:LONG	Depends on operation.

The message parameter defines the operation. It has one of the following values:

drawCtl	= 0	Draw the control (or control part).
calcCRect	= 1	Compute the rectangle to drag.
testCtl	= 2	Test where mouse button was pressed.
initCtl	= 3	Do any additional control initialization.
dispCtl	= 4	Take any additional disposal actions.
posCtl	= 5	Move the control's indicator.
thumbCtl	= 6	Compute the parameters for dragging an indicator.
dragCtl	= 7	Drag either a control's indicator, or the whole control.
autoTrack	= 8	Called while dragging if -1 passed to **TrackControl**.
newValue	= 9	Called when control gets new value.
setParams	= 10	Called when control gets new additional parameters.
moveCtl	= 11	Called control moves, compute new position for parts.
recSize	= 12	Return record size of control (in bytes).

As described below in the discussions of the routines that perform these operations, the value passed for param, depends on the operation. Similarly, the control definition procedure is expected to return a function result only where indicated; in other cases, the function should return zero.

The Draw Routine

message = drawCtl.
param = part code - draw part.
= zero - draw entire control.
(Only the low WORD is used, high WORD is undefined.)
RetValue = undefined.

The message drawCtl asks the control definition function to draw all or part of the control within its enclosing rectangle. The low-order WORD of param is a part code specifying which part of the control to draw, or zero for the entire control. If the control is invisible, there's nothing to do; if it's visible, the definition procedure should draw it (or the requested part), taking into account the current highlighting and value.

If param is the part code of the control's indicator, the draw routine can assume that the indicator hasn't moved; it might be called, for example, to highlight the indicator.

The Test Routine

message = testCtl.
param = low-order WORD = y point to check, in window's local coordinates.
= high-order WORD = x point to check, in window's local coordinates.
RetValue = undefined.

The Control Manager function **TestControl** sends the message testCtl to the control definition function when the mouse button is pressed in a visible control. This message asks in which part of the control, if any, a given point lies. The point is passed as the value of param, in the local coordinates of the control's window; the vertical coordinate is in the low-order word of the long integer and the horizontal coordinate is in the high-order word. The control definition function should return the part code for the part of the control that contains the point; it should return zero if the point is outside the control or if the control is inactive.

The Routine to Calculate Indicator Rectangle

message = calcCRect.
param = address of RECT.
RetValue = zero for default RECT, non-zero if RECT is set.

Just before the Control Manager starts to drag a control, or its indicator, it will call the control's definition procedure to determine the coordinates of the control, or its indicator. The highest bit of param will be clear if the whole control is to be dragged, or set if its indicator is to be dragged.

If the definition procedure returns zero, and the whole control is to be dragged, the RECT is set to the control's enclosing rectangle. If the definition procedure returns zero, and the control's indicator is to be dragged, the RECT is set to the thumb rectangle (see Scroll Bar Control Record) .

The Initialize Routine

message = initCtl.
param = low-order WORD is the param1 value passed to **NewControl.**
= high-order WORD is the param2 value passed to **NewControl.**
RetValue = undefined.

After allocating and initializing the control record as appropriate when creating a new control, the Control Manager sends the message initCtl to the control definition procedure. This gives the definition procedure a chance to perform any type-specific initialization it may require. For example, the control definition procedure for scroll bars initializes the thumb and page RECTs, and also stores param1 and param2 in the CtlData field. The initialize routine for standard buttons, check boxes, and radio buttons does nothing.

The Dispose Routine

message = dispCtl.
param = undefined.
RetValue = zero to continue disposal, non-zero to abort disposal.

The Control Manager's DisposeControl procedure sends the message dispCtl to the control definition function, telling it to carry out any additional actions required when disposing of the control. The predefined controls always return zero. If the definition procedure returns zero for RetValue, the control will be erased, taken out of the control list, and its record deallocated.

By returning a non-zero number for RetValue, the definition procedure has a chance to abort the disposal. This feature is provided even though I am unable to provide an example of when this feature might be useful.

The Position Routine

message = posCtl.
param = low-order WORD is the vertical offset (delta y).
high-order WORD is the horizontal offset (delta x)
RetValue = zero for default reposition, non-zero if reposition completed.

When dragging a control's indicator to completed, **TrackControl** calls the control definition procedure with the message posCtl to reposition the indicator and update the control's setting accordingly. The value of param is a point giving the vertical and horizontal offset, in pixels, by which the indicator is to be moved relative to its current position. (Typically, this is the offset between the points where the user pressed and released the mouse button while dragging the indicator.) The vertical offset is given in the low-order word of param and the horizontal offset in the high-order word. The definition procedure should calculate the control's new setting based on the given offset, update the CtlValue field, and redraw the control within its window to reflect the new setting.

Note: The Control Manager procedures **SetCtlValue** and **SetCtlParams** do not call the control definition procedure with this message; instead, they pass the newValue and setParams message (see below).

The Thumb Routine

message = thumbCtl.
param = pointer to parameter block for dragging an indicator.
RetValue = zero for default reposition, non-zero if reposition completed.

Before the Control Manger begins to drag a control's indicator, it will call the control's definition procedure with the message thumbCtl. The control definition procedure should respond by calculating the limiting rectangle, slop rectangle, axis constraint, and outline pattern to use for dragging the control's indicator. Param is a pointer to the following data structure:

```
limit_blk
   bound_rect:RECT    Limit of drag (not a pointer).
   slop_rect:RECT     Limit of cursor (not a pointer).
   axis_param:WORD    Movement constrain.
   drag_patt:LONG     Pointer to pattern for drag outline.
```

If the definition procedure returns zero, default parameters will be used. The defaults are computed thus:

bound_rect	PageRegion (see "Scroll Bar Control Record").
slop_rect	PageRegion plus 16 all around.
axis_param	2 if bit 12 of CtlFlag is clear, 1 if set.
drag_patt	Pattern generated from color7 in control's color table.

See DragRect for more information about the parameters in the limit_blk. The parameters in limit_blk will be passed to DragRect.

The Drag Routine

message = dragCtl.
param = part code to drag, zero to drag the entire control.
RetValue = zero to use default dragging, non-zero if dragging is completed.

The message dragCtl asks the control definition procedure to drag the control or its indicator around on the screen to follow the mouse until the user releases the mouse button. Param specifies whether to drag a part or the whole control: zero means drag the whole control, while a non-zero value is the part code of the control part to drag.

The control definition procedure need not implement any form of "custom dragging"; if it returns a result of zero, the Control Manager will use its own default method of dragging (calling **DragControl** to drag the control or **DragRect** to drag its indicator). Conversely, if the control definition procedure chooses to do its own custom dragging, it should signal the Control Manager not to use the default method by returning a non-zero result.

If the whole control is being dragged, the definition function should call **MoveControl** to reposition the control to its new location after the user releases the mouse button. If just the indicator is being dragged, the definition function should execute its own position routine (see below) to update the control's setting and redraw it in its window.

The Track Routine

message = autoTrack.
param = part code, zero if not currently in part.
RetValue = undefined.

You can design a control to have its action procedure in the control definition procedure. To do this, pass -1 for actionProc parameter to **TrackControl**. **TrackControl** will respond by calling the control definition procedure with the message autoTrack. The definition function should respond like an action procedure, as discussed in detail in the description of **TrackControl**. It can tell which part of the control the mouse button was pressed in from param, which contains the part code. The track routine for each of the standard control types does nothing.

The New Value Routine

message = newValue.
param = *low WORD is the new value, high WORD is the previous value**.
RetValue = undefined.

*Note: *Param is undefined in Control Manager versions 1.01 and lower.*

The Control Manager will call the control's definition procedure with the message newValue whenever a control's value changes. First, the Control Manager will store the new value in the CtlValue field of the control's record. The definition should compute any new parameters affected by the change, like a new thumb position for scroll bars, and then redraw the control (if visible). *The previous and new values are passed in param to make delta computation easier**. The definition procedure can assume that control is already drawn is the window, so, in the case of scroll bars, only the thumb has to be erased, and redrawn. Actually, the definition procedure for standard scroll bars only erases the part of the thumb that uncovered the page region, rather than the entire thumb.

The New Parameters Routine

message = setParams.
param = new parameters.
RetValue = undefined.

The Control Manager will call the control's definition procedure with the message setParams whenever a control's additional parameters change. The term 'additional parameters' is defined by the control. The values could be anything, even a pointer to more parameters. The definition should the perform necessary actions the new parameters cause, including redrawing the control if needed. The definition procedure can assume that control is already drawn is the window, unlike when new parameters are sent with the message initCtl (see "The Initialize Routine").

The only predefined control that uses additional parameters is the scroll bar. The low-order WORD is the view value, and the high-order WORD is the data size. Simple buttons, check boxes, and radio button do nothing with addition parameters. The standard scroll bar definition procedure will store the values in the CtlData field of the control's record, compute a new thumb, and draw the new thumb in the scroll bar (if visible).

The Move Routine

message = moveCtl.
param = low-order WORD is the change in the vertical axis (delta y), high-order WORD is the change in the horizontal axis (delta x).
RetValue = undefined.

The Control Manager will call the control's definition procedure with the message moveCtl from **MoveControl**. The Control Manager will first hide the control, with **HideControl**, if it was visible and move the control's enclosing rectangle (CtlRect field). The definition procedure should compute any other parameters necessary and return. For example, the standard definition procedure for scroll bars will also move the Thumb and PageRegion fields in the control record. Upon return, the Control Manager will do a **ShowControl** if the control was visible on entry, to draw the control at its new position. The definition procedure should not redraw the control here, but should do everything necessary to ensure the control will be drawn properly at its new position.

The Record Size Routine

message = recSize.
param = undefined.
RetValue = number of bytes needed for control's record.

The Control Manager call the control's definition procedure the message recSize from **NewControl** before it allocates memory for the control's record. **NewControl** will then allocate how ever many bytes is returned in RetValue for the control's record.

If your control only needs the standard control record, like buttons, check boxes, and radio buttons, return the size of the standard record. If your control needs additional data fields, like a scroll bar, return the size of the standard record, plus the additional size. You should never return a number less than the number of bytes in a standard record.

Note: TheControl, the handle of the control, passed to the definition procedure is not valid in this case. Because the control's record has not been allocated, no access to the record should be performed during this call. After the record has been allocated and initialized by the Control Manager, the definition procedure will be called again with the message initCtl, see "The Initialize Routine" below.

Constants

```
NoPart          0
SimpleButt      2
CheckBox        3
RadioButt       4
UpArrow         5
DownArrow       6
PageUp          7
PageDown        8
GrowBox         10
Thumb           129

SimpleProc      $00000000
CheckProc       $02000000
RadioProc       $04000000
ScrollProc      $06000000

CTL_VIS         $0080
UP_FLAG         $0001
DOWN_FLAG       $0002
LEFT_FLAG       $0004
RIGHT_FLAG      $0008
DIR_SCROLL      $0010
FAMILY          $007F
BOLD_BUTT       $0001

noConstraint    0       No constraint on movement.
hAxisOnly       1       Horizontal axis only.
vAxisOnly       2       Vertical axis only.

drawCtl         0       Draw control command.
calcCRect       1       Compute drag RECT command.
testCtl         2       Hit test command.
initCtl         3       Initialize command.
dispCtl         4       Dispose command.
posCtl          5       Move indicator command.
thumbCtl        6       Compute drag parameters command.
dragCtl         7       Drag command.
autoTrack       8       Action command.
newValue        9       Set new value command.
setParams       10      Set new parameters command.
moveCtl         11      Move command.
recSize         12      Return record size command.
```

Data Types

CtlNext	0	Handle	Handle of next control.
CtlOwner	4	Pointer	Pointer to control's window.
CtlRect	8	RECT	Enclosing rectangle.
CtlFlag	16	Byte	Bit flags.
CtlHilite	17	Byte	Highlighted part.
CtlValue	18	Integer	Control's value.
CtlProc	20	Pointer	Control's definition procedure.
CtlAction	24	Pointer	Control's action procedure.
CtlData	28	LongInt	Reserved for CtlProc's use.
CtlRefCon	32	LongInt	Reserved for application's use.
CtlColor	36	Pointer	Control's color table.

color1	0	Integer	
color2	2	Integer	
color3	4	Integer	
color4	6	Integer	
color5	8	Integer	
color6	10	Integer	
color7	12	Integer	

bound_rect	0	RECT	Drag bounds.
slop_rect	8	RECT	Cursor bounds.
axis_param	16	Integer	Movement constrains.
drag_patt	18	Pointer	Pattern for drag outline.

INDEX

Documentation Développeurs
Apple Computer France 1987

Document développeur numéro 48

System Loader

type d'upgrade de ce ducument : 5
1 Documentation de première catégorie inchangée
2 Documentation de deuxième catégorie mise à jour
3 Documentation de deuxième catégorie inchangée
4 Mise à jour payante de la documentation de première catégorie
5 Mise à jour gratuite de la documentation de première catégorie
6 Nouveautés payantes non vitales
7 Nouveautés gratuites et vitales

Taille : 40 page(s) environ

Domaine : Tool 17

VERSION : 01.20
DATE : 17.12.86

Date: December 17, 1986

Author: Lou Infeld

Subject: System Loader ERS

Document Version Number: 01:20

Revision History

01:00	(09/09/86)	Definition of System Loader version 1.0
01:10	(11/14/86)	Changes made in support of OMF Version 2 Function name changes to correspond to macros OMF Version check logic added Error $1102 added UserID removed at UserShutdown in certain cases
01:20	(12/17/86)	Jump Table Segment Processed Flag added to Pathname Table Description of initial loading of Direct Page/Stack Segments changed Support of Reload and Initialization Segments added to Restart Example of finding a Load Segment changed Output from LoadSegName changed and support for Jump Table Segment Processed Flag added Description of UserShutDown changed

Index

History

The Apple // under ProDOS has a very basic System Loader. It is the part of the boot code that searches the boot disk for the first System file (any file of type $FF whose name ends with ".SYSTEM") and loads it into location $2000. If a System program wants to load another System program, it has to do all the work by making ProDOS calls.

Some programming environments such as Apple // Pascal and AppleSoft Basic provide loaders for programs running under them. The AppleSoft loader loads either System files, Basic files or binary code files. All these files are loaded either at a fixed address in memory or at an address specified in the file.

Since Cortland will have a large amount of "clean" memory, a more dynamic load facility is needed. Programs should be able to be loaded anywhere that is available in memory. The burden of determining where to load a program should be on a loader and not on the applications programmer. Also programs should be able to be broken into smaller program segments which can be loaded independently.

Therefore on Cortland, there will be a relocating System Loader. Files generated by the Linker will be loadable by the System Loader. The System Loader will provide a very powerful and flexible facility not currently available on the Apple //.

Overview

The System Loader will load programs or program segments by first calling the Memory Manager to find available memory. It will perform relocation during the load as necessary and will load each segment independently. Therefore, a large program can be broken up into smaller program segments each of which is loaded at separate locations in memory. Program segments can also be loaded dynamically as they are referenced rather than at program boot time. Additionally, the System Loader can be called by the program itself to load and unload program (or data) segments.

Definitions

The **Linker** is the program that combines files generated by compilers and assemblers, resolves all symbolic references and generates a file that can be loaded into memory and executed.

The **System Loader** is the part of the Operating System that reads the files generated by the **Linker** and loads them into memory (performing relocation if necessary).

Object Files are the output from an assembler or compiler and the input to the **Linker**.

Library Files are files containing general program Segments that the **Linker** can search.

Load Files are the output of the **Linker** and contain memory images which the **System Loader** will load into memory. Shell Load Files and Startup Load Files are special Load Files used by the Shell and ProDOS16 respectively.

Run Time Library Files are Load Files that contain general program Segments which can be loaded as needed by the **System Loader** and shared between applications.

Object Module Format is the general format used in Object Files, Library Files and Load Files.

An **OMF File** is a file in Object Module Format (i.e. an Object File, Library File or Load File).

A **Segment** is a individual component of an OMF file. Each file contains one or more Segments.

A **Code Segment** is a Segment in an Object File that contains program code.

A **Data Segment** is a Segment in an Object File that contains program data.

A **Load Segment** is a Segment in a Load File.

The **Controlling Program** is the program that requests the **System Loader** to initially load and run other programs and is responsible for shutting these programs down when they exit. A Finder is an example of a Controlling Program.

General

The **System Loader** processes files which conform to the Cortland Development Environment's definition of a Load File (see Cortland Object Module Format ERS). A Load File consists of Load Segments, each of which can be loaded independently. The Load Segments are numbered sequentially from 1.

Certain Load Segments are Static Load Segments. These Segments are meant to be loaded into memory at initial program load time and must stay in memory until program completion.

Another general type of Load Segments is the Dynamic Load Segment. These Segments are not loaded at boot time. They are loaded dynamically during program execution. This can happen automatically by means of the **Jump Table** mechanism or manually at the specific request of the application. When these Segments are not being referenced, they can be purged by the **Memory Manager.**

There are several other attributes that Load Segments can have (see OMF ERS for a complete list of attributes).

There are several special types of Load Segments. The **Jump Table Segment** (KIND=$02), when loaded into memory, provides a mechanism whereby Segments in memory can trigger the loading of other Segments not yet in memory.

The **Pathname Segment** (KIND=$04). It contains information about the Load Files that are referenced.

The **Initialization Segment** (KIND=$10). It is used for code that is to be executed before all the rest of the Load Segments are loaded.

The **Direct Page/Stack Segment** (KIND=$12) defines the application's Direct Page and stack requirements. This segment will be loaded into Bank 0 and its starting address and length are passed to the **Controlling Program** who will set the Direct Register and Stack Pointer to the start and end of this segment before transferrring control to the program.

During the initial load, the **System Loader** has all the information needed to resolve all inter-segment references between the Static Load Segments. But during the dynamic loading of Dynamic Load Segments, it can only resolve references in the Dynamic Load Segment to the already loaded Static Load Segments. Therefore, the general rule is that Static Segments can be referenced by any type of segment but Dynamic Segments can only be referenced through JSL calls through the **Jump Table.**

If the **System Loader** is called to perform the initial load of a program, it will load all the Static Load Segments and the Segment Jump and Pathname Tables (if they exist). A RAM based **Memory Segment Table** will be constructed during this process.

If the **System Loader** is called during an interrupt and it is already processing a request, a BUSY error ($1105) will result.

Memory Manager Interface

The **System Loader** and the **Memory Manager** work closely together.

When the **System Loader** loads Static Segments, it calls the **Memory Manager** to allocate corresponding memory blocks which are marked as unpurgeable and unmoveable. Dynamic Segments are marked as purgeable but locked. Position Independent Segments are marked as moveable.

When the **System Loader** unloads a specific segment, it calls the **Memory Manager** to purge the corresponding memory blocks. However, if the **Controlling Program** wishes to unload all segments associated with a UserID (application shut-down), it calls the **System Loader** Application Shutdown function which calls the **Memory Manager** to first purge all Dynamic Segments for the UserID and then make all the Static Segments purgeable. The purpose of this is to keep an application in memory, if possible, in case it needs to be re-loaded in the near future. This will greatly speed up a Finder or Switcher. The complication occurs when the **Memory Manager** has to actually purge one of the segments of a User. The **System Loader** must then purge all the remaining segments. Otherwise, the program will not have all its static segments in memory when it is re-loaded and executed.

The relationship between a Load Segment in a Load File and the corresponding memory block is very close. The average Load Segment will be loaded into a memory block having the attributes:

Locked
Fixed
Purge Level=0 (for Static)
Purge Level=1 (for Dynamic)

Depending on the ORG, KIND, BANKSIZE and ALIGN fields in the Segment Header, other memory attributes will be used:

if ORG>0, the "Fixed Address" attribute is set.
if BANKSIZE=$10000, the "May not cross bank boundry" attribute is set.
if 0<BANKSIZE<$10000 then use Align factor=MAX(BANKSIZE,ALIGN)
otherwise use Align factor=ALIGN:
if 0<Align Factor<=$100, the "Page Aligned" attribute is set.
if Align Factor>$100, Bank Alignment is forced (not an attribute).
if bit 13 of KIND=1, the "Fixed" attribute is removed.
if bit 11 of KIND=1, the "Fixed Address" attribute is removed and the "Fixed Bank" attribute is set.
if KIND indicates Direct Page/Stack Segment, the "Fixed Bank" and "Page Aligned" attributes are set.

A memory block can be made purgeable ("unloaded") by a call to the **System Loader**. However, the other attributes must be changed through **Memory Manager** calls. Since the Memory Handle for a memory block is stored in the **Memory Segment Table, Memory Manager** information is accessible. Other memory block information that may be useful to a program are:

Start location
Size of segment
UserID
Purge Level (0 - UnPurgeable
1 - Least Purgeable
3 - Most Purgeable)

Also, if the Memory Handle is NIL (i.e the Memory Address is 0), the memory block has been purged.

Restrictions

The Object Module Format and the Linker have general capabilities above what is needed or desired for the Cortland computer. The **System Loader**, on the other hand, is designed specifically for the Cortland computer. Therefore, there are certai abilities that are not supported or are restricted. This section will list these differences.

The NUMSEX field of the Segment Header must be 0.
The NUMLEN field of the Segment Header must be 4.
The BANKSIZE field of the Segment Header must be <=$10000.
The ALIGN field of the Segment Header must be <=$10000.

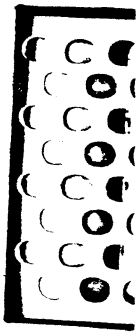
If any of the above is not true, the **System Loader** will return with a "Segment is foreign" error ($110B). The BANKSIZE and ALIGN restrictions will be enforced by the **Linker** and should not make it to the Load File.

ALIGN and BANKSIZE can be any multiple of 2. The **Memory Manager**, and therefore the **System Loader**, can not handle so general a requirement. The **Memory Manager** can currently only be told that a memory block be page aligne or not cross a bank boundary. The **Memory Manager** may handle Bank Alignme in the near future. All is not lost, however, because the **System Loader** will fulfill the general requirements in the following, somewhat inefficient, way:

Any value of BANKSIZE other than 0 and $10000 will result in a memory blc that is either page aligned (if BANKSIZE<=$100) or bank aligned (if BANKSIZE>$100). Since the **Linker** will make sure that the segment smaller than BANKSIZE, the requirement that the segment not extend past BANKSIZE boundary will be met (there will be wasted space in the memory block however).

Any value of ALIGN will be bumped to either page alignment or bank alignment.

If there is a BANKSIZE other than 0 and $10000 and a non-zero ALIGN, the maximum of the two will be used to determine the alignment to be used.



Data Structures

Globals

SEGTBL -- Absolute address of Memory Segment Table
JMPTBL -- Absolute address of Jump Table List
PATHTBL -- Absolute address of Pathname Table
USERID -- UserID of current application

Memory Segment Table

The **Memory Segment Table** is a linked list. Each entry corresponds to one memory block known to the **System Loader**. These memory blocks were the result of loading Load Segments from a Load File. The format of each entry in the **Memory Segment Table** is:

Next entry handle	-- 4 bytes
Previous entry handle	-- 4 bytes
UserID	-- 2 bytes
Memory Handle	-- 4 bytes
Load File Number	-- 2 byte
Load Segment Number	-- 2 bytes
Load Segment Kind	-- 2 bytes

where:

"Next entry handle" is the memory handle of the next entry in the Memory Segment Table. This handle is 0 in the last entry.

"Previous entry handle" is the memory handle of the previous entry in the Memory Segment Table. This handle is 0 in the first entry.

"UserID" is the UserID associated with this segment. It is needed in case the "Memory Handle" is NIL and the UserID can therefore not be determined directly from the Memory Manager.

"Memory Handle" is the handle of the memory block obtained from the **Memory Manager**. More information about the segment is available through this handle (e.g. UserID, Purge Priority).

"Load File Number" corresponds to the Load File or Run Time Library File from which the segment was obtained. If this number is 1, this segment is in the initial Load File.

"Load Segment Number" is the segment number of the Load Segment in the Load File.

"Load Segment Kind" is the KIND field from the Segment Header of this segment.

Jump Table List

The **Jump Table List** (or **Jump Table**) is the mechanism that allows programs to reference segments that are loaded into memory only when they are needed. The **Jump Table** is a linked list containing the UserID and Handle to each **Jump Table Segment** (KIND=$02) that the **System Loader** has encountered. Any Load File and Run Time Library File may contain a **Jump Table Segment.** The format of each entry in the **Jump Table List** is:

Next entry handle	-- 4 bytes
Previous entry handle	-- 4 bytes
UserID	-- 2 bytes
Memory Handle	-- 4 bytes

where:

"Next entry handle" is the memory handle of the next entry in the Jump Table List. This handle is 0 in the last entry.

"Previous entry handle" is the memory handle of the previous entry in the Jump Table List. This handle is 0 in the first entry.

"UserID" is the UserID associated with this Jump Table Segment.

"Memory Handle" is the handle of the memory block associated with this Jump Table Segment.

When the **Linker** encounters a JSL to an external Dynamic Segment, it creates an entry in the **Jump Table Segment.** It then links the JSL to the **Jump Table Segment** entry it just created. The format of this entry in the **Jump Table Segment** is:

UserID (2 bytes)
Load File Number (2 bytes)
Load Segment Number (2 bytes)
Load Segment Offset (4 bytes)
jsl Jump Table Load Function

where the Load File Number, Segment Number and Offset refer to the location of the external reference. The rest of the entry is a call to the **System Loader** Jump Table Load function. The UserID and the actual address of the **System Loader** function will be patched by the **System Loader** during Initial Load. This format is considered the "unloaded" state of the entry.

When the JSL instruction actually executes, control is transferred to the **Jump Table** entry which in turn transfers to the **System Loader**. The **System Loader**

extracts the segment information from the **Jump Table** entry, the file information from the **Pathname Table** and loads the Dynamic Segment, changes the entry in the **Jump Table** to its "loaded" state and transfers to the location in the just loaded segment. Typically, the location in the loaded segment is a subroutine and when it exits with a RTL, control is eventually transferred to the location following the original JSL instruction.

The loaded state of a **Jump Table** entry is very similar to the unloaded state except that the JSL to the **System Loader** Jump Table Load function is replaced by a JML to the external reference. A typical loaded entry would look like this:

UserID (2 bytes)
Load File Number (2 bytes)
Load Segment Number (2 bytes)
Load Segment Offset (4 bytes)
jml external reference

Pathname Table

The **Pathname Table** is created by **System Loader** to remember the pathnames associated with each Load File it comes across. At initial load, the **System Loader** creates the first entry in the **Pathname Table** from the pathname specified in the Initial Load function call. During the load, if the **System Loader** comes across a Pathname Segment (KIND=$04), it adds all the pathname entries to the **Pathname Table**. If Run Time Library Files are referenced during program execution, other Pathname Segments may be added.

Each entry in the **Pathname Table** is in the following format:

Next entry handle (4 bytes)
Previous entry handle (4 bytes)
UserID (2 bytes)
File Number (2 bytes)
File Date (2 bytes)
File Time (2 bytes)
Direct Page/Stack Address (2 bytes)
Direct Page/Stack Size (2 bytes)
Jump Table Segment Processed Flag (2 bytes)
File Pathname (Pascal string)
---

where:

"Next entry handle" is the memory handle of the next entry in the **Pathname Table**. This handle is 0 in the last entry.

"Previous entry handle" is the memory handle of the previous entry in the **Pathname Table**. This handle is 0 in the first entry.

"UserID" is the UserID associated with this entry. In general, each Load File and each Run Time Library will have a different UserID and one entry in the **Pathname Table**. When a Run Time Library is first encountered during an Application execution, the **System Loader** will have a Run Time Library type of UserID assigned to it.

"File Number" is a number assigned by the **Linker** or **System Loader** for a specific Load File. File number 1 is reserved for the initial Load File.

The "File Date" and "File Time" are ProDOS directory items that the **Linker** retrieved during the link process. The **System Loader** will compare these values with the ProDOS directory of the Run Time Library File at run time. If they don't compare, the **System Loader** will not load the requested Load Segment. This facility guarantees that the Run Time Library File used at link time is the same Run Time Library File loaded at execution time.

The "Direct Page/Stack" Address and Size is the information about the Direct Page and Stack buffer that was allocated during the Initial Load of this Load File (not applicable to Run Time Library Load Files). This allows the Restart function to resurrect an application without performing a Get File Info call on the Load File.

The "Jump Table Segment Processed Flag" indicates whether the Jump Table Segment (if any) in the Load File has been loaded yet. This flag will always be set for Initial Load Files but will be clear for Run Time Library Files. The first time a Segment in a Run Time Library File is requested, the **System Loader** will first load the Jump Table Segment if this flag is clear.

The "File Pathname" is the pathname of this entry. ProDOS 16 supports 8 prefixes, three of which have fixed definitions:

0/ - Boot volume
1/ - Application subdirectory (out of which the application is running)
2/ - System Library subdirectory (initially /BOOT/SYSTEM/LIBS)

The pathname must be a complete pathname except if either prefix 1/ or 2/ are used.

Mark List

The **Mark List** is created by **System Loader** to remember the file locations of the relocation dictionary of each Load Segment. The format of the **Mark List** is:

Next Available Slot (4 bytes)
End of Table (4 bytes)
Segment Number (2 bytes)

File Mark (4 bytes)
Segment Number (2 bytes)
File Mark (4 bytes)
Segment Number (2 bytes)
File Mark (4 bytes)
...
...
Segment Number (2 bytes)
File Mark (4 bytes)
---

where:

"Next Available Slot" is the relative offset of the next empty entry in the **Mark List**.

"End of List" is the relative offset to the end of the **Mark List.**

The **Mark List** is initially large enough for 100 Marks and grows larger as needed.

Functions

General

Since the **System Loader** is a Cortland Tool, its functions are called by making calls through the Cortland Tool mechanism. The calling sequence for **System Loader** functions is the standard Tool calling sequence. Space for the output parameter (if any) is pushed on the stack followed by each input parameter ***in the order specified in the function description.*** This is followed by:

```
ldx #$11+FuncNum|8
jsl Dispatcher
```

where "FuncNum" is the **System Loader** function number and the "$11" is the Tool Number for the **System Loader**. Upon return, the A register will contain the status and the Carry will be set if an error occurred. If there is output, each output parameter must be pulled off the stack ***in the order specified in the function description.***

The Jump Table Load function does not use the above calling sequence. It can not be called by an application directly but is called indirectly by a Jump Table entry. In this case the absolute address of the function is patched by the **System Loader**.

LoaderInitialization

Function Number:	$01
Input:	none
Output:	none
Errors:	none

This function will initialize the System Loader. It should only be called at system initialization time. All System Loader tables are cleared and no assumptions are made about the current or previous state of the system.

LoaderStartup

Function Number:	$02
Input:	none
Output:	none
Errors:	none

This function does nothing and need not be called.

LoaderShutDown

Function Number:	$03
Input:	none
Output:	none
Errors:	none

This function does nothing and need not be called.

LoaderVersion

Function Number:	$04
Input:	none
Output:	Loader Version (2 bytes)
Errors:	none

This function will return the Version Number of the System Loader.

LoaderReset

Function Number:	$05
Input:	none
Output:	none
Errors:	none

This function does nothing and need not be called.

LoaderStatus

Function Number:	$06
Input:	none
Output:	Status (TRUE or FALSE 2 bytes)
Errors:	none

This function will always return TRUE since the **System Loader** will always be in the initialized state.

InitialLoad

Function Number:	$09
Input:	UserID (2 bytes)
	Address of Load File Pathname (4 bytes)
	Don't Use Special Memory Flag (2 bytes)
Output:	UserID (2 bytes)
	Starting Address (4 bytes)
	Address of Direct Page/Stack buffer (2 bytes)
	Size of Direct Page/Stack buffer (2 bytes)
Errors:	$0000 - Operation succcessful
	$1102 - OMF Version error
	$1104 - File not Load File
	$1105 - System Loader is busy
	$1109 - SegNum out of sequence
	$110A - Illegal load record found
	$110B - Load Segment is foreign
	$00xx - ProDOS error
	$02xx - Memory Manager error

A **Controlling Program** (such as ProDOS, Basic, Switcher, etc.) will call the **System Loader** to perform an "Initial Load".

If a complete UserID is specified, the **System Loader** will use that when allocating memory for the Load Segments. If the Main ID portion of the UserID is 0, a new UserID is obtained from the UserID Manager based on the Type portion of the UserID. If the Type portion is 0, an Application type UserID is requested from the UserID Manager.

If the Don't Use Special Memory Flag is TRUE (i.e. not 0), the **System Loader** will **NOT** load any static load segments into Special Memory. However, dynamic load segments will be loaded into any memory.

ProDOS is called to open the specified Load File using the input pathname. If any ProDOS errors occurred or if the file is not a Load File type ($B3-$BE), the **System Loader** will return the appropriate error.

If the Load File was successfully opened, the **System Loader,** adds the Load File information to the **Pathname Table**, and calls the Load Segment by Number function for each Static Load Segment in the Load File.

If an Initialization Segment (KIND=$10) is loaded, the **System Loader** will immediately transfer control to that segment in memory. When the **System Loader** regains control, the rest of the static segments are loaded normally.

If the Direct Page/Stack Segment (KIND=$12) is loaded, its starting address and length are returned as output.

If any of the static segments could not be loaded, the **System Loader** will abort the load and return the error.

After all the Static Load Segments have been loaded, return is made to the **Controlling Program** with the starting address of the first Load Segment (not an Initialization Segment) of File Number 1. Note that the **Controlling Program** is responsible for setting up the stack and Direct Page registers and actually transferring control to the loaded program.

Restart

Function Number:	$0A
Input:	UserID (2 bytes)
Output:	UserID (2 bytes)
	Starting Address (4 bytes)
	Address of Direct Page/Stack buffer (2 bytes)
	Size of Direct Page/Stack buffer (2 bytes)
Errors:	$0000 - Operation succcessful
	$1101 - Application not found
	$1105 - System Loader is busy
	$1108 - UserID error
	$00xx - ProDOS error
	$02xx - Memory Manager error

A **Controlling Program** (such as ProDOS, Basic, Switcher, etc.) can call the **System Loader** to perform a "restart" of an application still in memory. Only software that is "reentrant" can be successfully restarted. For a program to be "reentrant", it must initialize its variables and not assume that they will be preset at Load time. A Reload Segment can be used for initializing data because it is reloaded from the file during a Restart. The **Controlling Program** must determine whether a given program can be restarted.

An existing UserID (ignoring the Aux ID) must be specified, otherwise the **System Loader** will return error $1108. If the UserID is not known to the **System Loader**, error $1101 will be returned.

Applications can be "restarted" only if all the segments in the Memory Segment table with the specified UserID are in memory. Note these segments are the application's static segments. If this is the case, the **System Loader** resurrects the application by calling the Memory Manager to lock and make all its segments unpurgeable. The UserID and the starting address obtained from the first segment are returned as well as the Direct Page/Stack information from the Pathname Table. After all the static segments are resurrected, the **System Loader** looks for Initialization and Reload Segments and executes the former and reloads the latter.

If there is a Pathname Table entry for the UserID but not all the segments are in memory, the **System Loader** will call its Cleanup Routine to purge the UserID from all its tables and then perform an Initial Load instead of a Restart.

LoadSegNum (Load Segment by Number)

Function Number:	$0B
Input:	UserID (2 bytes)
	Load File Number (2 bytes)
	Load Segment Number (2 bytes)
Output:	Address of segment (4 bytes)
Errors:	$0000 - Operation succcessful
	$1101 - Segment not found
	$1102 - OMF Version error
	$1104 - File not Load File
	$1105 - System Loader is busy
	$1107 - File Version error
	$1109 - SegNum out of sequence
	$110A - Illegal load record found
	$110B - Segment is foreign
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will load a specific Load Segment into memory. This is the workhorse function of the **System Loader**. Normally, a program will call this function to manually load a Dynamic Load Segment. If a program calls this function to load a Static Load Segment, the **System Loader** will not patch any existing references to the newly loaded segment.

First the **Memory Segment Table** is searched to see if there is an entry for the requested Load Segment. If there is already an entry, the handle to the memory block is checked to verify it is still in memory. If it is still in memory, this function does nothing further and returns without an error. If the memory block has been purged, the **Memory Segment Table** entry is deleted.

Next the "Load File Number" is looked up in the **Pathname Table** to get the Load File pathname.

Next the Load File type is checked. If it is not a Load File (types $B3-$BE), error $1104 is returned.

Next the Load File's "last_mod" value is compared to File Date and File Time values in the **Pathname Table**. If these values do not match, error $1107 is returned. This indicates that the Run Time Library File at the specified pathname is not the Run Time Library File that was scanned when the application was linked together.

ProDOS is then called to open the specified Load File. If ProDOS has a problem, its error code is returned.

Next the Load File is searched for a Load Segment corresponding to the specified "Load Segment Number". If there is no segment corresponding to the "Load Segment Number", error $1101 is returned. If the VERSION field contains a value which is not supported by the **System Loader**, error $1102 is returned. If the

SEGNUM field does not correspond to the "Load Segment Number", error $1109 is returned. If the NUMSEX and NUMLEN fields are not "0" and "4", error $110B is returned.

If the Load Segment is found and its Segment Header is correct, a memory block is requested from the **Memory Manager** of size specified in the LENGTH field in the Segment Header. If the ORG field in the Segment Header is not 0, a memory block starting at that address is requested. Other attributes are set according to Segment Header fields (see Memory Manager Interface section).

If the UserID specified is not 0, it is used as the UserID of the memory block. If the UserID specified is 0, the memory block will be marked as belonging to the UserID of the current User (in USERID).

If the requested memory is not available, the **Memory Manager** and the **System Loader** will try several techniques to free up memory:

The **Memory Manager** will purge memory blocks that are marked purgeable

The **Memory Manager** will move moveable segments to enlarge contiguous memory

The **System Loader** will call its Cleanup routine to free its own unused internal memory

If all these techniques fail, the **System Loader** will return with the last **Memory Manager** error.

Once enough memory is available, the Load Segment is loaded into memory and the relocation dictionary (if any) is processed. Note only the following Object Module Format records are supported by the **System Loader:**

LCONST	($F2)
DS	($F1)
RELOC	($E2)
INTERSEG	($E3)
cRELOC	($F5)
cINTERSEG	($F6)
SUPER	($F7)
END	($00)

Any other records encountered will result in a $110A error.

A new entry is added to the **Memory Segment Table.**

Finally, the **System Loader** returns with the Memory Handle of the memory block.

Note that since Load Segments in a Load File are numbered sequentially starting at 1, to find Load Segment 5, the **System Loader** must scan through the first 4 Load

Segments before finding Load Segment 5. Each Load Segment Header must be processed because Load Segments as well as Load Segment Header are variable length. It is simpler than it sounds because Load Segments start on block boundaries and the number of blocks in each Load Segment is the first field in the Segment Header. The following logic sample will find and read the specified Load Segment:

```
segnum:=Load Segment number;
fileid:=Load File;
open(fileid);
file_mark:=0;
for i:=1 to segnum do
  begin
    seek(fileid,file_mark); {find next header}
    get(fileid);            {read}
    file_mark:=file_mark+fileid^.BYTECNT;
  end;
```

UnloadSegNum (Unload Segment by Number)

Function Number:	$0C
Input:	UserID (2 bytes)
	Load File Number (2 bytes)
	Load Segment Number (2 bytes)
Output:	none
Errors:	$0000 - Operation succcessful
	$1101 - Segment not found
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will unload a specific Load Segment that is currently in memory.

The **System Loader** searches the **Memory Segment Table** for the "Load File Number" and "Load Segment Number". If there is no such entry, error $1101 is returned.

Next the **Memory Manager** is called to make the memory block purgeable using the Memory Handle in the table entry.

All entries in the **Jump Table** referencing the unloaded segment are changed to their "unloaded" states.

If the input UserID is 0, the UserID of the current user (in USERID) is assumed.

If both the Load File Number and the Load Segment Number are specified, the specific Load Segment is made purgeable whether it is static or dynamic. Note , if a static segment is unloaded, the application can not be ReStarted. If either input is 0, only dynamic segments will be made purgeable.

If the input Load Segment Number is 0, all dynamic segments in the specified Load File are unloaded.

If the input Load File Number is 0, all dynamic segments for the UserID are unloaded.

LoadSegName (Load Segment by Name)

Function Number:	$0D
Input:	UserID (2 bytes)
	Address of Load File Name (4 bytes)
	Address of Load Segment Name (4 bytes)
Output:	Address of segment (4 bytes)
	UserID (2 bytes)
	Load File Number
	Load Segment Number
Errors:	$0000 - Operation succcessful
	$1101 - Segment not found
	$1104 - File not Load File
	$1105 - System Loader is busy
	$1107 - File Version error
	$1109 - SegNum out of sequence
	$110A - Illegal load record found
	$110B - Load Segment is foreign
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will load a named Load Segment into memory.

The Load File type is checked. If it is not a Load File (types $B3-$BE), error $1104 is returned.

ProDOS is then called to open the specified Load File. If ProDOS has a problem, its error code is returned.

Next the Load File is searched for a Load Segment corresponding to the specified "Load Segment Name". If there is no segment with Segment Name requested, error $1101 is returned.

Now that the **System Loader** has located the requested Load Segment (and knows the Load Segment Number), it checks the **Pathname Table** to see whether the Load File is represented. If so, it uses the File Number from the table. Otherwise, the **System Loader** adds a new entry to the **Pathname Table** with an unused File Number. If the Jump Table Segment Processed Flag in the **Pathname Table** is clear, the **System Loader** loads the Jump Table Segment (if any) from the Load File and sets the Flag.

Next the **System Loader** attempts to load this Load Segment by calling the Load Segment by Number function. If the Load Segment by Number function returns an error, the Load Segment by Name function, in turn, returns this error. If the Load Segment by Number function is successful, the Load Segment by Name function returns the Load File Number, the Load Segment Number and the Memory Address of the segment in memory.

UnloadSeg

Function Number:	$0E
Input:	Address in Segment (4 bytes)
Output:	UserID (2 bytes)
	Load File Number (2 bytes)
	Load Segment Number (2 bytes)
Errors:	$0000 - Operation succcessful
	$1101 - Segment not found
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will unload the Load Segment which contains the specified address.

The **Memory Manager** is called to locate the memory block containing the specified address. If no memory block contains the address, error $1101 is returned. The UserID associated with the Handle of the memory block returned by the **Memory Manager** is extracted (from the **Memory Manager**'s internal table). The **Memory Segment Table** is scanned looking for the UserID and Handle. If an entry is not found, error $1101 is returned.

If the entry in the **Memory Segment Table** is for a Jump Table Segment, the specified address should be pointing to the **Jump Table** entry for a dynamic segment reference. The Load File Number and Segment Number of the **Jump Table** entry are extracted.

If the entry in the **Memory Segment Table** is not for a Jump Table Segment, the Load File Number and Segment Number of the **Memory Segment Table** entry are extracted.

The UnloadSegNum function is now called to actually unload the segment. The outputs of this function can be used as input to other **System Loader** functions.

GetLoadSegInfo

Function Number:	$0F
Input:	UserID (2 bytes)
	Load File Number (2 bytes)
	Load Segment Number (2 bytes)
	Address of User Buffer (4 bytes)
Output:	filled User Buffer
Errors:	$0000 - Operation succcessful
	$1101 - Entry not found
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will return the Memory Segment Table entry corresponding to the specified Load Segment.

The Memory Segment Table is searched for the specified entry. If the entry is not found, error $1101 is returned. If the entry is found, the contents except for the link pointers are moved into the User Buffer.

GetUserID

Function Number:	$10
Input:	Address of Pathname (4 bytes)
Output:	UserID (2 bytes)
Errors:	$0000 - Operation succcessful
	$1101 - Entry not found
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will search the Pathname Table for the specified Pathname. If a match is found, the corresponding UserID is returned. If the input Pathname does not start with prefix 1/ or 2/, it is first expanded to a full pathname before the search. A **Controlling Program** can use this function to determine whether to perform a Restart of an application or an Initial Load.

LGetPathname

Function Number:	$11
Input:	UserID (2 bytes)
	File Number (2 bytes)
Output:	Address of Pathname (4 bytes)
Errors:	$0000 - Operation succcessful
	$1101 - Entry not found
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function will search the Pathname Table for the specified UserID and File Number. If a match is found, the address of the Pathname in the Pathname Table is returned. ProDOS uses this call to get the pathname of an existing application so that it can set the Application prefix before restarting it.

UserShutDown

Function Number:	$12
Input:	UserID (2 bytes)
	Quit Flag (2 bytes)
Output:	UserID (2 bytes)
Errors:	$0000 - Operation succcessful
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function is called by the **Controlling Program** to close down an application which has just terminated. If the UserID specified is 0, the current UserID (USERID) is assumed.

The Quit Flag corresponds to the Quit Flag used in the ProDOS 16 Quit call.

If the Quit Flag is 0, all Memory Blocks for the UserID (with the AuxID set to 0) are disposed and the Cleanup Routine is called to purge the **System Loader**'s internal tables of the UserID. The application can not be Restarted. The UserID is also removed from the system so that it can be reused.

if the Quit Flag is $8000, all Memory Blocks for the UserID are purged (not disposed). All the **System Loader**'s internal tables for the UserID remain intact. The application can be reloaded but not Restarted (i.e. the pathname for the application is remembered).

If the Quit Flag is any other value, the memory blocks associated with the specified UserID (with Aux ID cleared) are processed as follows:

all memory blocks corresponding to Dynamic Load Segments are disposed
all memory blocks corresponding to Static Load Segments are made purgeable
all other memory blocks are purged

In addition, all dynamic segment entries in the **Memory Segment Table** and all entries in the **Jump Table List** for the specified UserID are removed. The application is now in a "zombie" state and can be resurrected by the **System Loader** very quickly because all the static segments are still in memory. However, as soon as any one static segment is purged by the **Memory Manager** for whatever reason, the **System Loader** must reload the application from its original Load File.

Jump Table Load

Function Number:	none
Input:	UserID (2 bytes)
	Load File Number (2 bytes)
	Load Segment Number (2 bytes)
	Load Segment Offset (4 bytes)
Output:	none
Errors:	$0000 - Operation succcessful
	$1101 - Segment not found
	$1104 - File not Load File
	$1105 - System Loader is busy
	$00xx - ProDOS error
	$02xx - Memory Manager error

This function is called by an "unloaded" **Jump Table** entry to load a Dynamic Load Segment.

This function calls the Load Segment by Number function with the the Load File Number and Load Segment Number. If any errors occurred, the **System Loader** will report a System Death.

If the Load Segment by Number function has sucessfully loaded the segment, the **Jump Table** entry is made "loaded" by replacement of the JSL to the Jump Table Load function with a JML to the absolute address of the reference in the Dynamic Load Segment.

The **System Loader** will now transfer control to the absolute address.

Cleanup Routine

Function Number: none
Input: UserID
Output: none
Errors: none

This function is internal to the **System Loader.** Its function is to cleanup the **System Loader**'s internal tables in order to free memory.

If the UserID is 0, the **Memory Segment Table** is scanned and all dynamic segments for all UserID's will be purged.

If the UserID is not 0, all Load Segments (both dynamic and static) for that UserID will be disposed. In addition, all entries for the UserID in the **Memory Segment Table**, **Jump Table List** and **Pathname Table** will be removed.

Error Codes

$0000	Operation successful
$1101	Segment /Application/Entry not found
$1102	OMF Version error
$1103	not used
$1104	File is not a Load File
$1105	Loader is busy
$1106	not used
$1107	File version error
$1108	UserID error
$1109	SegNum out of sequence
$110A	Illegal load record found
$110B	Segment is foreign

References

"Object Module Format ERS" by Lou Infeld -- Apple Computer
"ProDOS ORCA/M User's Guide" -- The Byte Works
"Cortland Development Environment Core" -- The Byte Works
"The Tool Locator ERS", by Steve Glass -- Apple Computer

Documentation Développeurs
Apple Computer France 1987

Document développeur numéro 54

Line Edit

type d'upgrade de ce ducument : 5
1 Documentation de première catégorie inchangée
2 Documentation de deuxième catégorie mise à jour
3 Documentation de deuxième catégorie inchangée
4 Mise à jour payante de la documentation de première catégorie
5 Mise à jour gratuite de la documentation de première catégorie
6 Nouveautés payantes non vitales
7 Nouveautés gratuites et vitales

Taille : 16 page(s) environ

Domaine : Tool 20

VERSION : 00:30
DATE : 12.08.86

Date - August 12, 1986

Author - Cheryl Ewy

Subject - LineEdit ERS

Document Version Number - 00:30

Revision History

00:00	(06-17-86)	Initial Release
00:10	(07-14-86)	LEFromScrap routine added LEToScrap routine added Call numbers changed for some routines Support for control-F, control-Y and control-X added to LEKey Support for triple-click added to LEClick LETextBox information updated Error codes updated
00:20	(08-07-86)	LETextBox no longer requires the text to end with a CR Minor clarifications have been added
00:30	(08-12-86)	Input parameters to LEStartUp reversed

OVERVIEW

LineEdit is a set of routines that provide basic line editing capabilities. These capabilities include:

- Inserting new text.
- Deleting characters that are backspaced over.
- Translating mouse activity or arrow keys into text selection.
- Deleting selected text and possibly inserting it elsewhere, or copying text without deleting it.

The LineEdit routines follow the Apple Human Interface Guidelines and support these standard features:

- Positioning the insertion point by clicking the mouse.
- Moving the insertion point 1 character at a time by using the left and right arrow keys.
- Moving the insertion point 1 word at time by using Option-LeftArrow or Option-RightArrow.
- Moving the insertion point to the beginning or end of the line by using OpenApple-LeftArrow or OpenApple-RightArrow.
- Selecting text by clicking and dragging with the mouse.
- Selecting text by using Shift-LeftArrow and Shift-RightArrow.
- Selecting words by double-clicking the mouse.
- Selecting words by using Shift-Option-LeftArrow or Shift-Option-RightArrow.
- Selecting the whole line by triple-clicking the mouse.
- Selecting from the insertion point to the beginning or end of the line by using Shift-OpenApple-LeftArrow or Shift-OpenApple-RightArrow.
- Extending or shortening the selection by clicking the mouse while holding down the Shift key.
- Deleting the selection or the character to the left of the insertion point by using Backspace.
- Deleting the selection or the character to the right of the insertion point by using Control-F.
- Deleting the selection or the whole line by using Control-X.
- Deleting the selection or from the insertion point to the end of the line using Control-Y.
- Inverse highlighting of the current text selection, or display of a blinking vertical bar at the insertion point.

- Cutting (or copying) and pasting. LineEdit puts text you cut or copy into the LineEdit scrap.

LineEdit does not support:

- more than 256 characters per line (except when using LETextBox)
- centered or right-justified text (except when using LETextBox)
- fully justified text (text aligned with both the left and right margins)
- automatic line wrap
- scrolling
- the use of more than one font or stylistic variation per line
- fonts which kern
- "intelligent" cut and paste (adjusting spaces between words during cutting and pasting)
- tabs

EDIT RECORDS

To edit a line of text on the screen, LineEdit needs to know where and how to display the text, where to store the text, and other information related to editing. This display, storage, and editing information is contained in an **edit record** that defines the complete editing environment.

You prepare to edit text by specifying a destination rectangle in which to draw the text and a view rectangle in which the text will be visible. LineEdit incorporates the rectangles and the drawing environment of the current grafPort into an edit record, and returns a handle to the record. Most of the LineEdit routines require you to pass this handle as a parameter.

In addition to the two rectangles and a description of the drawing environment, the edit record also contains:

- a handle to the text to be edited
- a pointer to the grafPort in which the text is displayed
- the current selection range, which determines which characters will be affected by the next editing operation

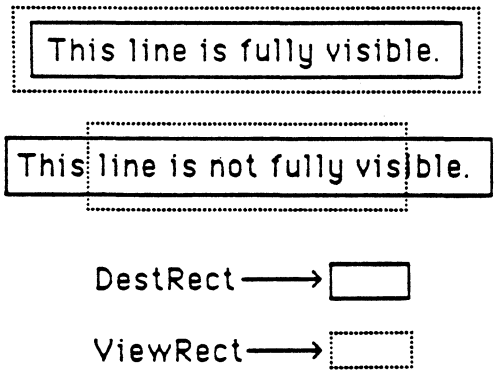
For most operations, you don't need to know the exact structure of an edit record since the LineEdit routines access the record for you. The structure of an edit record is given below.

```
TextHandle      HANDLE      {hndl to text to be edited}
Length          INTEGER     {current length of text}
MaxLength       INTEGER     {maximum text length}
DestRect        Rect        {destination rectangle}
ViewRect        Rect        {view rectangle}
PortPtr         POINTER     {ptr to grafPort}
LineHite        INTEGER     {used for highlighting}
BaseHite        INTEGER     {used for drawing the text}
SelStart        INTEGER     {start of selection range}
SelEnd          INTEGER     {end of selection range}
ActFlg          INTEGER     {used internally}
CarAct          INTEGER     {used internally}
CarOn           INTEGER     {used internally}
CarTime         LONGINT     {used internally}
HiliteHook      POINTER     {ptr to highlight routine}
CaretHook       POINTER     {ptr to caret routine}
```

Warning: Never change any of the fields in the edit record directly. The fields can only be changed by calling LineEdit routines.

The DestRect and ViewRect Fields

The destination rectangle is the rectangle in which the text is drawn. The view rectangle is the rectangle within which the text is actually visible. In other words, the view of the text drawn in the destination rectangle is clipped to the view rectangle. The view rectangle also determines the area in which mouse activity affects the text. Clicking or dragging the mouse outside the view rectangle does not affect the insertion point or selection range. In most cases, the view rectangle should be a few pixels larger than the destination rectangle on all sides. This provides some slop area around the text in which the mouse activity will still have an effect on the text.

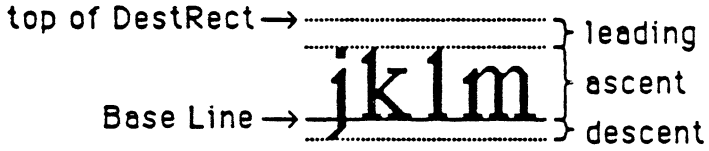


You specify both rectangles in the local coordinates of the grafPort. To ensure that the first and last characters in each line are legible in a document window, you may want to inset the destination rectangle at least four pixels from the left and right edges of the grafPort's portRect.

Edit operations may of course lengthen or shorten the text. If the text becomes too long to be enclosed by the destination rectangle, it's simply drawn beyond the right edge. LineEdit doesn't support scrolling or wrapping to the next line.

The LineHite and BaseHite Fields

The BaseHite field has to do with where the the text is drawn relative to the top of the DestRect. The LineHite field has to do with where the caret or highlighting of the selection range is drawn relative to the text. The BaseHite field specifies the distance between the top of the DestRect and the base line (leading + ascent). The LineHite field specifies the height of the line (leading + ascent + descent).



The SelStart and SelEnd Fields

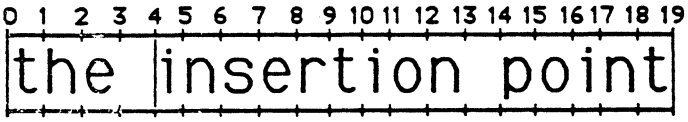
In the text editing environment, a **character position** is an index into the text, with position 0 corresponding to the first character. The edit record includes fields for character positions that specify the beginning and end of the current selection range, which is the series of characters where the next editing operation will occur. For example, the procedures that cut or copy from the text of an edit record do so to the current selection range.

The selection range, which is inversely highlighted when the window is active, extends from the beginning character position to the end character position. The figure below shows a selection range between positions 3 and 8, consisting of five characters (the character at position 8 isn't included). The end position of a selection range may be 1 greater than the position of the last character of the text, so that the selection range can include the last character.

> **Note** - Highlighting of the selection range is done by calling the QuickDraw routine InvertRect, not by swapping the text and background colors.

If the selection range is empty - that is, its beginning and end positions are the same - that position is the text's insertion point, the position where characters will be inserted. By default, it's marked with a blinking caret (actually a vertical bar).

selection range
beginning at position 3
and ending at position 8



insertion point
at position 4

If you call a procedure to insert characters when there's a selection range of one or more characters rather than an insertion point, the editing procedure automatically deletes the selection range and replaces it with an insertion point before inserting the characters.

The HiliteHook and CaretHook Fields

The HiliteHook and CaretHook fields are used for text highlighting and drawing the caret. These fields are initialized to $00000000. You can set the contents of these fields by calling the LESetHilite and LESetCaret routines.

If you store the address of a routine in HiliteHook, that routine will be used instead of the QuickDraw procedure InvertRect whenever a selection range is to be highlighted or unhighlighted. For example, you can write a routine which underlines selection ranges instead of highlighting them. The routine will be called with the stack containing a pointer to the rectangle enclosing the text being highlighted or unhighlighted. The routine must remove the pointer from the stack and return with an RTL.

The routine whose address is stored in CaretHook acts exactly the same way as the HiliteHook routine, but on the caret instead of the selection highlighting, allowing you to change the appearance of the caret. The routine will be called with the stack containing a pointer to the rectangle that encloses the caret. The routine must remove the pointer from the stack and return with an RTL.

USING LINE EDIT

Before using LineEdit, you must initialize the Memory Manager, QuickDraw, the Event Manager and the Window Manager, in that order.

The first LineEdit routine to call is the initialization routine LEStartUp. The other LineEdit routines do not check to see if LineEdit is active when they are called.

Call LENew to allocate an edit record; it returns a handle to the record. Most of the text editing routines require you to pass this handle as a parameter.

When you're completely done with an edit record and want to dispose of it, call LEDispose.

To make a blinking caret appear at the insertion point, call the LEIdle routine as often as possible (at least once each time through the main event loop); if it's not called often enough, the caret will blink irregularly.

When a mouse-down event occurs in the view rectangle (and the window is active) call the LEClick routine. LEClick controls the placement and highlighting of the selection range in response to mouse activity, including supporting use of Shift-Click to make extended selections.

Key-down, auto-key, and mouse events that pertain to text editing can be handled by several LineEdit routines:

- LEKey inserts characters, deletes characters backspaced over, controls the placement and highlighting of the selection range in response to the LeftArrow and RightArrow keys, and handles the Control-F, Control-X and Control-Y commands.

- LECut transfers the selection range to the LineEdit scrap, removing the selection range from the text.

- LEPaste inserts the contents of the LineEdit scrap. By calling LECut, changing the insertion point, and then calling LEPaste, you can perform a "cut and paste" operation, moving text from one place to another.

- LECopy copies the selection range to the LineEdit scrap. By calling LECopy, changing the insertion point, and then calling LEPaste, you can make multiple copies of text.

- LEDelete removes the selection range (without transferring it to the scrap). You can use LEDelete to implement the Clear command.

- LEInsert inserts specified text. Since LEDelete and LEInsert do not modify the scrap, they're useful for implementing the Undo command.

After each editing procedure, LineEdit redraws the text if necessary from the insertion point to the end of the text. You never have to set the selection range or insertion point yourself; LEClick and the editing routines leave it where it should be. If you want to modify the selection range directly, however - to highlight an initial default name or value, for example - you can use the LESetSelect routine.

To implement cutting and pasting of text between different applications, or between applications and desk accessories, you need to transfer the text between the LineEdit scrap (which is a private scrap used only by LineEdit) and the Scrap Manager's desk scrap. To do this, use the LEFromScrap and LEToScrap routines.

When an update event is reported for a text editing window, call LEUpdate (along with the Window Manager routines BeginUpdate, EraseRect and EndUpdate) to redraw the text.

> **Note:** After changing any fields of the edit record that affect the appearance of the text, you should call the Window Manager routine InvalRect so that the text will be updated.

The LEActivate and LEDeactivate routines must be called each time GetNextEvent reports an activate event for a text editing window. LEActivate simply highlights the selection range or displays a caret at the insertion point; LEDeactivate unhighlights the selection range or removes the caret.

The LESetText routine lets you change the text being edited. For example, if your application has several separate pieces of text that must be edited one at a time, you don't have to allocate an edit record for each of them. Allocate a single edit record, and then use LESetText to change the text.

If you want to draw noneditable text in any given rectangle, you can use the LETextBox routine.

LINE EDIT ROUTINES

HouseKeeping

```
LEBootInit              Call # $01
```

LEBootInit is called at boot time. It does nothing.

```
LEStartUp              Call # $02

     input             ProgramID       INTEGER
     input             ZeroPageAdrs    INTEGER
```

LEStartUp initializes LineEdit and allocates a handle for the LineEdit scrap. The scrap is initially empty. ProgramID is the ID LineEdit will use when getting memory from the Memory Manager. ZeroPageAdrs is the starting address in Bank 0 of a 1-page work area assigned to LineEdit. Duplicate LEStartUp calls will cause an error to be returned.

> **Note:** You should call LEStartUp even if your application doesn't use LineEdit, so that desk accessories and dialog and alert boxes will work correctly.

```
LEShutDown             Call # $03
```

LEShutDown shuts down LineEdit and releases any workspace allocated to it. If LineEdit is not active, LEShutDown will return an error.

```
LEVersion              Call # $04

     input             Result space    WORD
     output            VersionInfo     INTEGER
```

LEVersion returns identifying information for LIneEdit.

```
LEReset                Call # $05
```

LEReset returns an error if LineEdit is active, otherwise it does nothing.

```
LEActive               Call # $06

     input             Result space    WORD
     output            ActiveFlag      INTEGER
```

LEActive returns a non-zero value if LineEdit is active, otherwise it returns a 0.

Edit Record Allocation

```
LENew                   Call # $09

      input             Result space    LONG WORD
      input             DestRectPtr     POINTER to Rect
      input             ViewRectPtr     POINTER to Rect
      input             MaxTextLen      INTEGER
      output            hLE             HANDLE
```

LENew allocates space for the text, creates and initializes an edit record, and returns a handle to the new edit record. DestRect and ViewRect are the destination and view rectangles, respectively. Both rectangles are specified in the current grafPort's coordinates. The view rectangle must not be empty. For example, don't make its right edge less than its left edge. If you don't want any text visible - specify a rectangle off the screen instead. MaxTextLen specifies how many bytes to allocate for the text. It should be a value from 1 to 256. The text will be limited to this length.

Call LENew once for every edit record you want allocated. The edit record incorporates the drawing environment of the grafPort, and is initialized with an insertion point at character position 0.

> **Note:** The caret won't appear until you call LEActivate.

```
LEDispose               Call # $0A

      input             hLE             HANDLE
```

LEDispose releases the memory allocated for the edit record and text specified by hLE. Call this procedure when you're completely through with an edit record.

Changing the Text of an Edit Record

```
LESetText               Call # $0B

      input             TextPtr         POINTER
      input             Length          INTEGER
      input             hLE             HANDLE
```

LESetText incorporates a copy of the specified text into the edit record specified by hLE. The TextPtr parameter points to the text, and the Length parameter indicates the number of characters in the text. If Length is greater than the maximum text length allowed for the edit record, only the maximum number of characters allowed will be copied into the edit record. The selection range is set to an insertion point at the end of the text. LESetText doesn't affect the text currently drawn in the destination rectangle, so call InvalRect afterward if necessary.

Insertion Point and Selection Range

```
LEIdle                  Call # $0C

     input              hLE                 HANDLE
```

LEIdle should be called repeatedly to make a blinking caret appear at the insertion point (if any) in the text specified by hLE. (The caret appears only when the window containing that text is active.) LineEdit observes a minimum blink interval: No matter how often LEIdle is called, the time between blinks will never be less than the minimum interval. The user can adjust the minimum blink interval with the Control Panel desk accessory.

To provide a constant frequency of blinking, LEIdle should be called as often as possible - at least once each time through the main event loop. Call it more than once if your application does an unusually large amount of processing each time through the loop.

> **Note:** LEIdle actually only needs to be called when the window containing the text is active.

```
LEClick                 Call # $0D

     input              EventPtr            POINTER to event record
     input              hLE                 HANDLE
```

LEClick controls the placement and highlighting of the selection range as determined by mouse events. Call LEClick whenever a mouse-down event occurs in the view rectangle of the edit record specified by hLE, and the window associated with that edit record is active. The EventPtr parameter should be a pointer to the mouse-down event record.

LEClick unhighlights the old selection range unless the selection range is being extended. If the mouse moves, meaning that a drag is occurring, LEClick expands or shortens the selection range accordingly. In the case of a double-click, the word under the cursor becomes the selection range; dragging expands or shortens the selection a word at a time. In the case of a triple-click, the entire line becomes the selection range. LEClick keeps control until the mouse button is released.

```
LESetSelect             Call # $0E

     input              SelStart            INTEGER
     input              SelEnd              INTEGER
     input              hLE                 HANDLE
```

LESetSelect sets the selection range to the text between SelStart and SelEnd in the text specified by hLE. The old selection range is unhighlighted, and the new one is highlighted. If SelStart equals SelEnd, the selection range is an insertion point, and a caret is displayed.

SelEnd and SelStart can range from 0 to 256. SelStart must be <= SelEnd. If SelEnd is anywhere beyond the last character of the text, the position just past the last character is used.

```
LEActivate          Call # $0F

     input          hLE             HANDLE
```

LEActivate highlights the selection range in the view rectangle of the edit record specified by hLE. If the selection range is an insertion point, it displays a caret there. This procedure should be called every time the Event Manager routine GetNextEvent reports that the window containing the edit record has become active.

```
LEDeactivate        Call # $10

     input          hLE             HANDLE
```

LEDeactivate unhighlights the selection range in the view rectangle of the edit record specified by hLE. If the selection range is an insertion point, it removes the caret. This procedure should be called every time the Event Manager routine GetNextEvent reports that the window containing the edit record has become inactive.

Editing

```
LEKey               Call # $11

     input          Key             WORD
     input          Modifiers       WORD
     input          hLE             HANDLE
```

LEKey replaces the selection range in the text specified by hLE with the character given by the Key parameter, and leaves an insertion point just past the inserted character. If the selection range is an insertion point, LEKey just inserts the character there.

If the Key parameter contains a Backspace character, the selection range or the character immediately to the left of the insertion point is deleted. If the Key parameter contains a Control-F character, the selection range or the character immediately to the right of the insertion point is deleted. If the Key parameter

contains a Control-X character, the selection range or the entire line is deleted. If the Key parameter contains a Control-Y character, the selection range or the text from the insertion point to the end of the line is deleted.

If the Key parameter contains a LeftArrow or RightArrow character, LEKey will move the insertion point or extend the selection range depending on the contents of the Modifiers parameter.

LEKey redraws the text as necessary.

Call LEKey every time the Event Manager routine GetNextEvent reports a keyboard event that your application decides should be handled by LineEdit. The Key parameter should be the key reported by the event record. The Modifiers parameter should be a copy of the Modifiers field in the event record.

> **Note:** LEKey inserts every character passed in the Key parameter (except for Backspace, Control-F, Control-X, Control-Y, LeftArrow and RightArrow), so it's up to the application to filter out all characters that aren't actual text (such as Command keys and other Control characters).

```
LECut                   Call # $12

     input              hLE                 HANDLE
```

LECut removes the selection range from the text specified by hLE and places it in the LineEdit scrap. The text is redrawn as necessary. Anything previously in the scrap is deleted. If the selection range is an insertion point, the scrap is emptied.

```
LECopy                  Call # $13

     input              hLE                 HANDLE
```

LECopy copies the selection range from the text specified by hLE into the LineEdit scrap. Anything previously in the scrap is deleted. The selection range is not deleted. If the selection range is an insertion point, the scrap is emptied.

```
LEPaste                 Call # $14

     input              hLE                 HANDLE
```

LEPaste replaces the selection range in the text specified by hLE with the contents of the LineEdit scrap, and leaves an insertion point just past the inserted text. The text is redrawn as necessary. If the scrap is empty, the

selection range is deleted. If the selection range is an insertion point, LEPaste just inserts the scrap there.

```
LEDelete            Call # $15

     input          hLE             HANDLE
```

LEDelete removes the selection range from the text specified by hLE, and redraws the text as necessary. LEDelete is the same as LECut (above) except that it doesn't transfer the selection range to the scrap. If the selection range is an insertion point, nothing happens.

```
LEInsert            Call # $16

     input          TextPtr         POINTER
     input          Length          INTEGER
     input          hLE             HANDLE
```

LEInsert takes the specified text and inserts it just before the selection range into the text indicated by hLE, redrawing the text as necessary. The TextPtr parameter points to the text to be inserted, and the Length parameter indicates the number of characters to be inserted. LEInsert doesn't affect either the current selection range or the scrap.

Text Display

```
LEUpdate            Call # $17

     input          hLE             HANDLE
```

LEUpdate redraws the text specified by hLE. Call LEUpdate every time the Event Manager routine GetNextEvent reports an update event for a text editing window - after you call the Window Manager routines BeginUpdate and EraseRect, and before you call the Window Manager routine EndUpdate. If you don't include the EraseRect call, the caret may sometimes remain visible when the window is deactivated.

```
LETextBox           Call # $18

     input          TextPtr         POINTER
     input          Length          INTEGER
     input          BoxPtr          POINTER to a Rect
     input          Just            INTEGER
```

LETextBox draws the specified text in the rectangle indicated by the BoxPtr parameter, with justification Just. LETextBox does an EraseRect on the rectangle before drawing the text and clips the text to the rectangle. LETextBox

is not limited to a single line on the screen as the other LineEdit routines are. LETextBox will wrap to the next line whenever a CR (ASCII $0D) character occurs in the text string.

The TextPtr parameter points to the text, and the Length parameter indicates the length of the text including the CR characters. The Length parameter can range from 0 to 32,767. The rectangle is specified in local coordinates. The Just parameter should be set to 0 for left justified text, 1 for centered text , and -1 for right justified text.

LETextBox creates its own edit record, which it deletes when it's finished, so the text it draws cannot be edited. LETextBox does not allocate space for the text string or make any copies of the text string.

Scrap Handling

```
LEFromScrap          Call # $19
```

LEFromScrap copies the desk scrap to the LineEdit scrap. If the number of characters in the desk scrap is > 256, an error is returned and the scrap is not copied.

```
LEToScrap            Call # $1A
```

LEToScrap copies the LineEdit scrap to the desk scrap.

```
LEScrapHandle        Call # $1B

     input           Result space    LONG WORD
     output          ScrapHndl       HANDLE
```

LEScrapHandle returns a handle to the LineEdit scrap.

```
LEGetScrapLen        Call # $1C

     input           Result space    WORD
     output          ScrapLength     INTEGER
```

LEGetScrapLen returns the size of the LineEdit scrap in bytes.

```
LESetScrapLen          Call # $1D

     input             NewLength       INTEGER
```

LESetScrapLen sets the size of the LineEdit scrap to the given number of bytes. NewLength should be a value from 0 to 256. If NewLength is > 256, it is set to 256.

Setting HiliteHook and CaretHook

```
LESetHilite            Call # $1E

     input             HiliteAdrs      POINTER
     input             hLE             HANDLE
```

LESetHilite sets the HiliteHook field to HiliteAdrs which should be the address of a routine which will be used to do highlighting and unhighlighting of the selection range.

```
LESetCaret             Call # $1F

     input             CaretAdrs       POINTER
     input             hLE             HANDLE
```

LESetCaret sets the CaretHook field to CaretAdrs which should be the address of a routine which will be used to draw the caret.

LINE EDIT ERROR CODES

$1401	Duplicate LEStartUp call
$1402	Reset error
$1403	LineEdit not active
$1404	Desk scrap too big to copy

Chapter 15

Print Manager

The Print Manager allows you to use standard QuickDraw II routines to print text or graphics on a printer. The Print Manager calls a printer driver to perform the specific printing tasks, so that your application doesn't need to know what kind of printer is connected to the computer.

You should already be familiar with QuickDraw II.

An application that supports printing must have three items in its File menu: Choose Printer, Page Setup, and Print. The following actions occur when the user selects one of these items:

- **Choose Printer:** When the user selects the Choose Printer item, a dialog is displayed that allows the user to select a destination device from the printer drivers on the system disk. The Choose Printer dialog also lets the user pick the port or slot to which the device is connected from the port drivers on the system disk. If AppleTalk is installed, the network is scanned for the names of all printers of the specified printer type.

  ◆ *Macintosh programmers:* On the Apple IIGS, the Choose Printer function is part of the Print Manager, rather than being implemented as part of the Chooser desk accessory.

- **Page Setup:** When the user selects the Page Setup item, the style dialog is displayed. The style dialog allows the user to specify formatting information, such as the page size and printing orientation. This information is not changed frequently and is usually saved with the document.

- **Print:** When the user selects the Print item, the job dialog is displayed. The job dialog lets the user select print quality, page range, number of copies, and printer-specific features, such as color printing.

Your application defines the image to be printed by using either the GrafPort that the Print Manager automatically gives you when you open a document for printing, or by supplying its own GrafPort.

◆ *Note:* The Print Manager does not automatically save the current GrafPort when it initializes the new GrafPort. If the application will need that GrafPort's information for later use, it *must* save the information itself.

Your application then prints text and graphics by drawing into the GrafPort with QuickDraw II, just as if it was drawing on the screen. The Print Manager installs its own versions of QuickDraw II's low-level drawing routines in this GrafPort, causing your higher-level QuickDraw II calls to drive the printer instead of drawing on the screen.

Important

Don't customize the QuickDraw II routines in the GrafPort being used for printing unless you're sure of what you're doing.

A preview of the Print Manager routines

To introduce you to the capabilities of the Integer Math Tool Set, all Print Manager routines are grouped by function and briefly described in Table 15-1. These routines are described in detail later in this chapter, where they are separated into housekeeping routines (discussed in routine number order) and the rest of the Print Manager routines (discussed in alphabetical order).

Table 15-1
Print Manager routines and their functions

Routine	Description
Housekeeping routines	
PMBootInit	Initializes the Print Manager; called only by the Tool Locator—must not be called by an application
PMStartUp	Starts up the Print Manager for use by an application
PMShutDown	Shuts down the Print Manager
PMVersion	Returns the version number of the Print Manager
PMReset	Resets the Print Manager; called only when the system is reset—must not be called by an application
PMStatus	Indicates whether the Print Manager is active
Print record and dialog routines	
PrDefault	Fills the fields of a specified print record with default values for the appropriate printer
PrValidate	Checks the contents of the specified print record for compatibility with the current version number of the Print Manager and the currently installed printer
PrStlDialog	Conducts a style dialog with the user to determine the page dimensions and other information needed for page setup
PrJobDialog	Conducts a job dialog with the user to determine the print quality, range of pages to print, and so on
PrChoosePrinter	Conducts a Choose Printer dialog with the user to determine the printer and port driver to use
Printing routines	
PrOpenDoc	Initializes a GrafPort for use in printing a document, makes it the current port, and returns a pointer to the port
PrCloseDoc	Closes the GrafPort being used for printing
PrOpenPage	Begins a new page
PrClosePage	Ends the printing of the current page
PrPicFile	Prints a spooled document
PrPixelMap	Prints all or part of a specified pixel map
Error handling routines	
PrError	Returns the last printer error code left during the printing loop by Print Manager routines
PrSetError	Stores a specified value into the global variable where the Print Manager keeps its printer error code
Printer driver and port driver routines	
PrDriverVer	Returns the version number of the currently installed printer driver
PrPortVer	Returns the version number of the currently installed port driver

Print dialog boxes

The dialog boxes seen by the user when he or she chooses an item from the File menu are the Choose Printer dialog, the style dialog, and the job dialog. These dialogs allow the user to specify information needed by the Print Manager to process the print job. This information is stored in the appropriate print record fields.

Choose Printer dialog box

The Choose Printer dialog box allows the user to choose the printer to be used for printing and the port that connects that printer to the system, as illustrated in Figure 15-1.

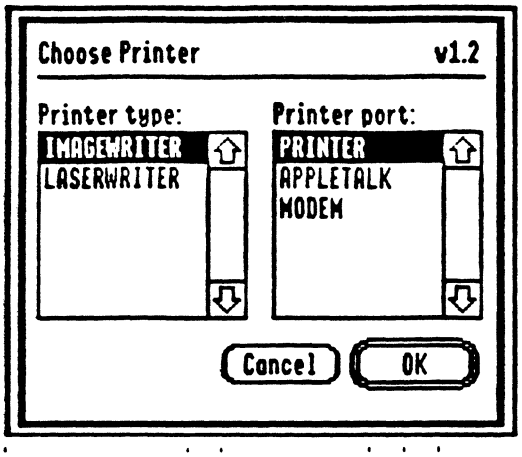


Figure 15-1
Choose Printer dialog box

If the printer chosen is connected to the system via AppleTalk, then an additional dialog box appears, showing all available printers of the specified type on the network, as illustrated in Figure 15-2.

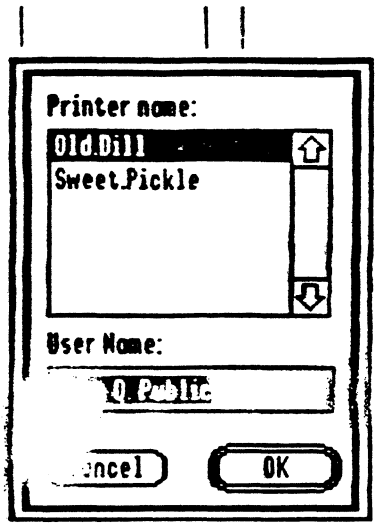


Figure 15-2
Printer names dialog box

Style dialog box

The style dialog box presents the user with a choice of paper size and orientation.

Both the ImageWriter® and the LaserWriter® printers interpret the paper options as shown in Table 15-2, although each printer does not offer all of the options.

Table 15-2
Printer paper sizes

Option	Dimensions
US Letter	8 1/2 by 11 inches
US Legal	8 1/2 by 14 inches
A4 Letter	210 by 297 millimeters
B5 Letter	176 by 250 millimeters
International fanfold	210 millimeters by 12 inches

Both printers also use the same definitions of vertical sizes, although the ImageWriter cannot print intermediate text. The vertical-size definitions are

- Normal, which prints for 640 mode at 80 ppi (pixels per inch), and for 320 mode at 40 ppi horizontally and 36 ppi vertically
- Intermediate, which prints at 54 ppi vertically
- Condensed, which prints at 72 ppi vertically

◆ *Macintosh programmers:* The condensed vertical size resembles the screen size of Macintosh text.

Both printers also allow the user to choose between printing in **portrait mode** (in which text prints from left to right) and **landscape mode** (in which text prints from top to bottom).

Other options differ on the two printers, so the style dialog boxes also differ. The dialog box that appears if the user chooses an ImageWriter is shown in Figure 15-3.

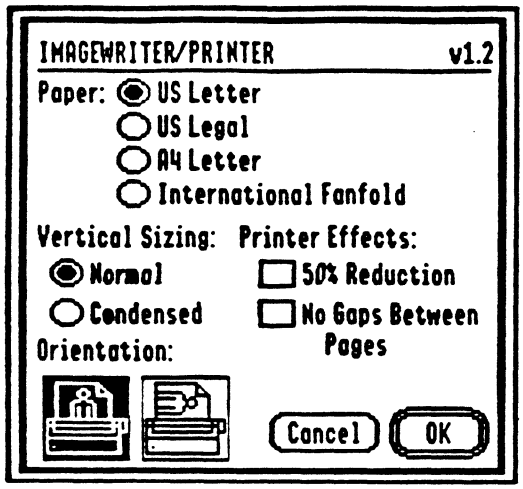


Figure 15-3
Style dialog box for ImageWriter

The printer effects choices for the ImageWriter allow the user to print at half size as well as with no gaps between pages (that is, the printer uses the full vertical 11 inches).

The dialog box that appears if the user chooses a LaserWriter is shown in Figure 15-4.

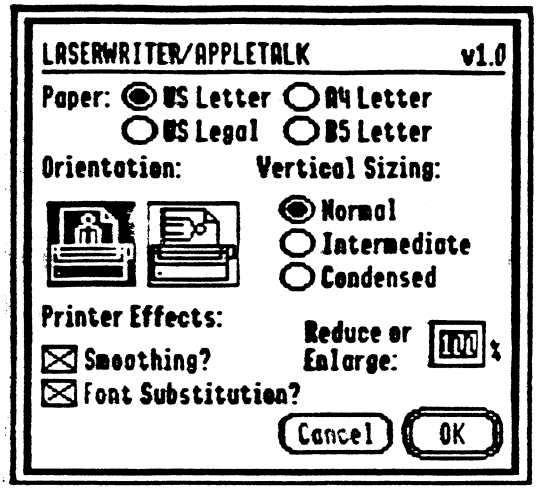


Figure 15-4
Style dialog box for LaserWriter

There are two printing options available on the LaserWriter that don't exist on the ImageWriter.

Smoothing asks the system to smooth out any bit-mapped fonts with jagged edges. **Font substitution** tells the system to make the following substitutions if the specified font is not in the LaserWriter: Helvetica for Geneva, Courier for Monaco, Times for New York. Any other font is downloaded as a bit map to the LaserWriter.

- *Note:* At the time of publication, the Print Manager did not support downloading bit-mapped fonts to the LaserWriter; that is, Courier will be substituted for all fonts other than those listed above. However, on the Laserwriter Plus, the Zapf Chancery font will be substituted for the Venice font.

The printed representation of the document can range from 25 to 400 percent of the original size, with 100 percent representing normal size.

Job dialog box

The **job dialog box** allows the user to communicate the page range, the number of copies, and the paper source to the Print Manager. In addition, the ImageWriter job dialog offers print-quality choices and the option to print in color.

In most cases, your application doesn't need to know what choices the user makes for this dialog; that is, the application will simply use QuickDraw II calls to draw into the GrafPort being used for printing, and the Print Manager will handle the user's selections.

The ImageWriter job dialog box is shown in Figure 15-5.

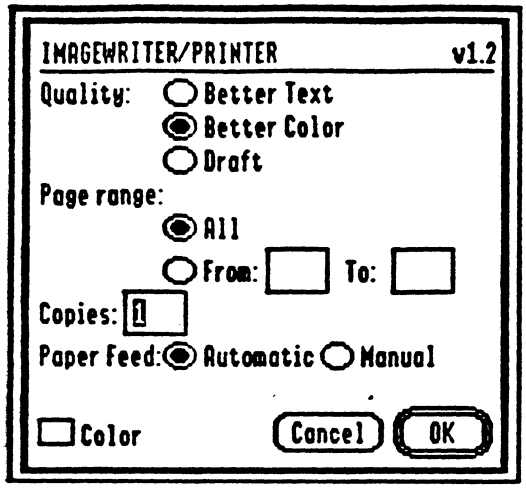


Figure 15-5
Job dialog box for ImageWriter

The Better Text option doubles the resolution, but halves the color choices available; therefore, if your application isn't printing color graphics, this choice by the user will produce the highest quality.

The Better Color option prints the document at the same resolution as the screen, with the same number of screen colors available. The option is most appropriate when printing color graphics.

The Draft option is useful only when the user wants to quickly print text without any formatting information.

If the user clicks the Color box, the ImageWriter driver will print using a color ribbon if the ribbon is available on the specified ImageWriter. If the ribbon is not available, or if the user does not click the Color box, the ImageWriter will print in black and white.

The LaserWriter job dialog box is shown in Figure 15-6.

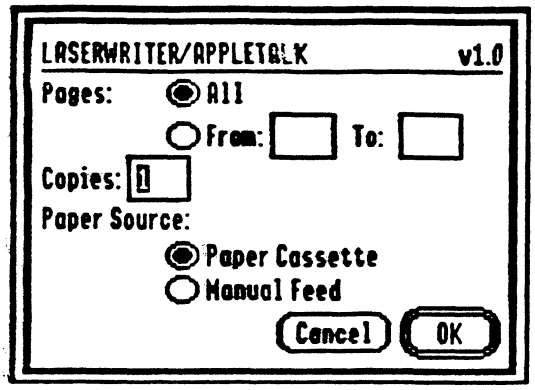


Figure 15-6
Job dialog box for LaserWriter

Print records

To format and print a document, the Print Manager checks the information contained in a data structure called a **print record**. The Print Manager fills in the entire print record for you by using information specified by the user during the job and style dialogs.

- *Note:* Whenever your application saves a document, it should write an appropriate print record into the document. This sets up the printing parameters so that the document retains and uses those parameters the next time the document is printed.

Information contained in the print record includes the following:

- Dimensions of the printable area of the page
- Whether the application must calculate the margins, the size of the physical sheet of paper, and the printer's vertical and horizontal resolution
- Whether draft or spool printing is being used

Important

Your application doesn't need to change much of the data in the print record; usually, the only fields you'll need to set directly are those containing optional information in the job subrecord. You should use use standard dialog routines for controlling the print record information. If you want to directly change values in the print record, be sure you know what you are doing.

Print records are referred to by handles. The structure of a print record is shown in Figure 15-7.

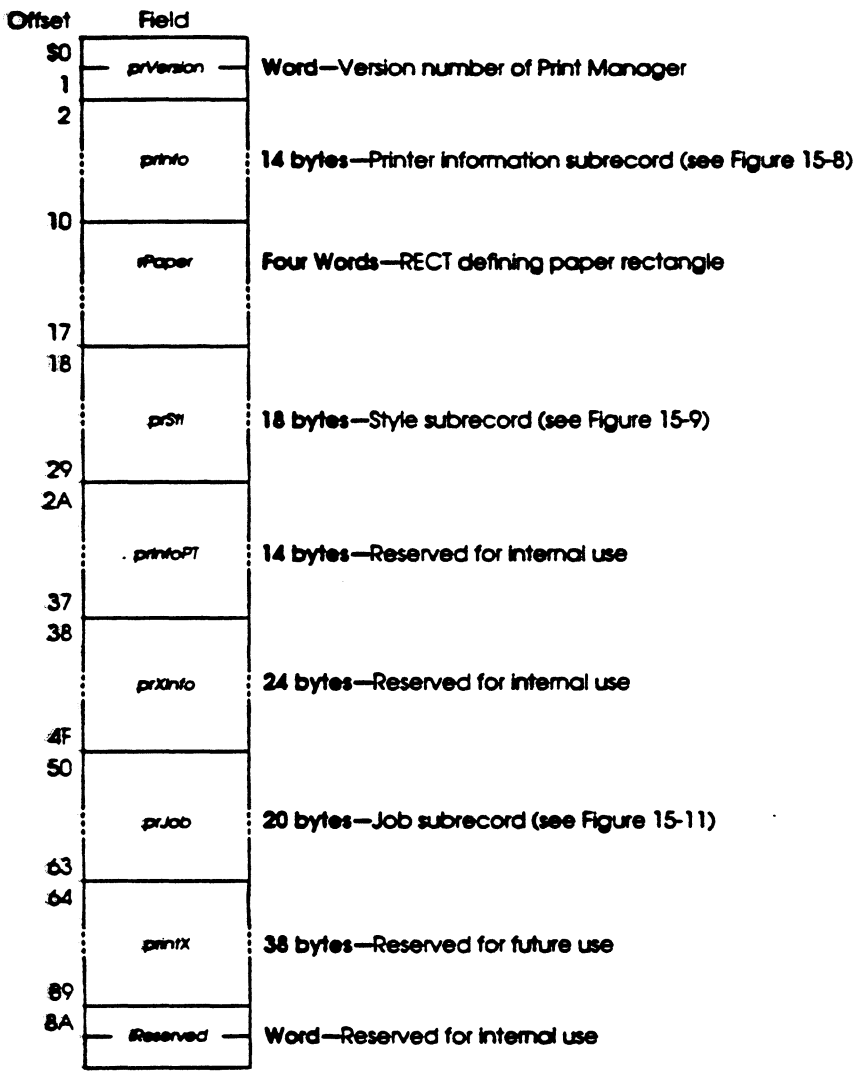


Figure 15-7
Print record

The *prVersion* field identifies the version of the printer driver that initialized this print record. If you try to use a print record that is invalid for the current version of the Print Manager or for the currently installed printer, the Print Manager will correct the record by filling it with default values.

The other fields of the print record are discussed in the following sections.

Printer information subrecord

The printer information subrecord (the print record field *prInfo*) gives you the information needed for page composition. The structure of the printer information subrecord is shown in Figure 15-8.

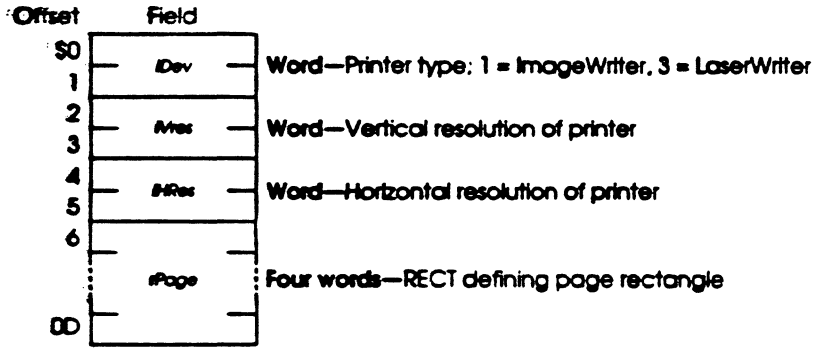


Figure 15-8
Printer information subrecord

The *iDev* field identifies which type of printer the user selected in the Choose Printer dialog.

The *rPage* field is the page rectangle, representing the boundaries of the printable page. The GrafPort's boundsRect, portRect, and clipRgn are set to this rectangle. Its top-left corner always has coordinates (0,0); the coordinates of the bottom-right corner give the maximum page height and width attainable on the given printer, in pixels. Typically these are slightly less than the physical dimensions of the paper, because of the printer's mechanical limitations. The value of *rPage* is set as a result of the style dialog.

The *rPage* rectangle is inside the paper rectangle, specified by the print record *rPaper* field. The *rPaper* field gives the physical paper size, defined in the same coordinate system as *rPage*. Thus, the top-left coordinates of the paper rectangle are typically negative, and the bottom-right coordinates are greater than those of the page rectangle.

The *iVRes* and *iHRes* fields contain the printer's vertical and horizontal resolution in pixels per inch. Thus, if you divide the width of *rPage* by *iHRes*, you get the width of the page rectangle in inches.

Style subrecord

The style subrecord (the print record field *prStl*) contains information obtained from the user via the style dialog as well as the job dialog. Some of the fields in this subrecord have different meanings for different printers. The structure of the style subrecord is shown in Figure 15-9.

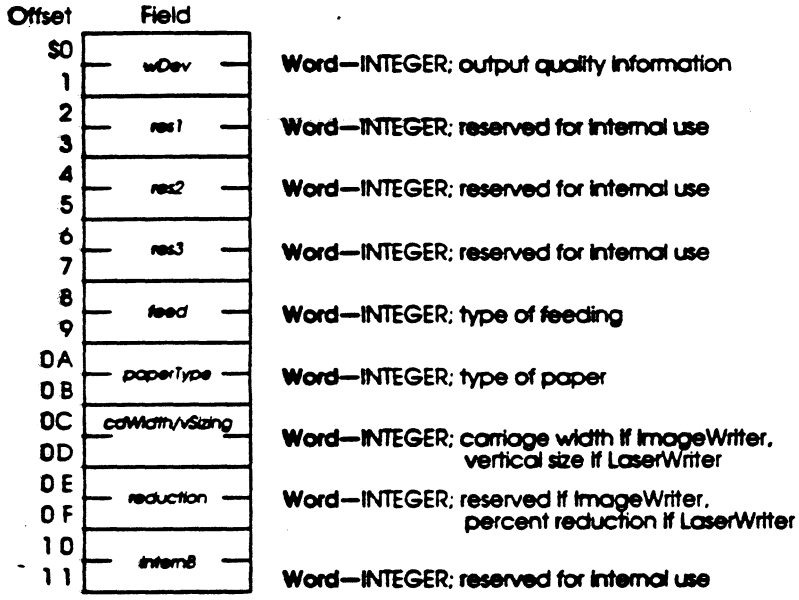


Figure 15-9
Printer style subrecord

The possible values for these fields are shown in the following sections.

ImageWriter style subrecord values

wDev
- Bit 6 0 = no gap, 1 = gap
- Bit 5 0 = black and white, 1 = color
- Bit 4 reserved
- Bit 3 0 = 50% reduction, 1 = full size
- Bit 2 0 = condensed, 1 = normal
- Bit 1 0 = landscape, 1 = portrait
- Bit 0 0 = normal quality, 1 = best quality

feed feedManual = 0, feedAuto = 1

paperType
- UsLetter = 0
- UsLegal = 1
- A4Letter = 2
- IntlFanFold = 3

crWidth 960 for all paper types

LaserWriter style subrecord values

wDev
- Bit 3 0 = substitute fonts, 1 = don't substitute fonts
- Bit 2 0 = smoothing; 1 = no smoothing
- Bit 1 0 = portrait, 1 = landscape
- Bit 0 Reserved

feed feedManual = 0, feedAuto = 1

paperType
- UsLetter = 0
- UsLegal = 1
- A4Letter = 2
- B5Letter = 3

vSizing
- Normal = 0
- Intermediate = 1
- Condensed = 2

Job subrecord

The job subrecord (the print record field *prJob*) contains information about a particular printing job. Its contents are set as a result of the job dialog. The job subrecord is defined as shown in Figure 15-10.

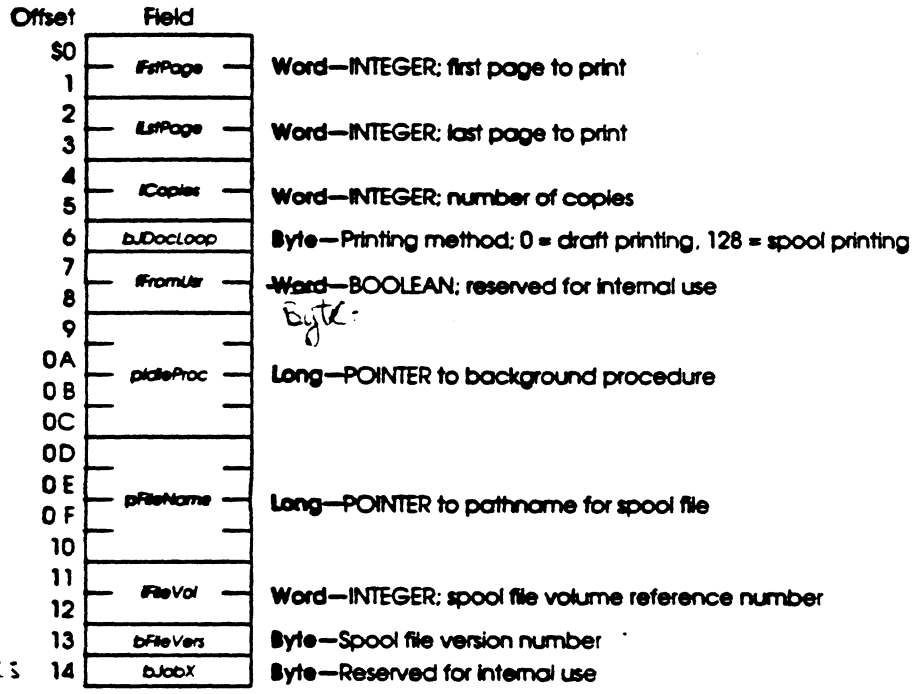


Figure 15-10
Job subrecord

The *iFstPage* and *iLstPage* fields designate the first and last pages to be printed. These page numbers are relative to the first page counted by the Print Manager. The Print Manager cannot use any page numbering placed within a document by an application.

The *iCopies* field is the number of copies to print. The Print Manager automatically handles multiple copies for spool printing or for printing on the LaserWriter. Your application needs this number only for draft printing on the ImageWriter.

The *bJDocLoop* field designates the printing method, either draft or spool, that the Print Manager will use. Draft printing means that the document will be printed immediately. Spool printing means that printing may be deferred: The Print Manager writes a representation of the document's printed image to a disk file (or possibly to memory); this information is then converted into a bit image and printed. Your application can check this field to determine which type of printing the user selected.

The *pIdleProc* field is a pointer to the background procedure (explained later) for this printing operation. In a newly initialized print record this field is set to NIL, designating the default background procedure, which just polls the keyboard and cancels further printing if the user types Apple-period. You can install your own background procedure by storing a pointer to that procedure directly into the *pIdleProc* field.

For spool printing, your application may optionally provide a spool file pathname, volume reference number, and version number, as follows:

- *pFileName* is the full pathname of the spool file as a ProDOS string. This field is initialized to NIL and is generally not changed by the application. NIL denotes the default filename.
- *tFileVol* is the unit number of spool file volume, initialized to 0.
- *tFileVers* is the version number of the spool file, initialized to 0.

Printing modes and resolutions

The Print Manager supports the various display modes available on the Apple IIGS. The IIGS is capable of both color and gray scale in either 320 or 640 modes. When the user requests color printing via the job dialog, the Print Manager prints the document in color on a printer that is capable of color printing, such as the ImageWriter II; otherwise it will print it in black and white.

This next section presents an overview of the algorithm for printing the Super Hi-Res color screen to the ImageWriter printer. It describes both the transformation from the color screen to a color printer and the method used to print the color screen in black and white (gray scale).

The IIGS screen has two resolution modes:

- 320 horizontal by 200 vertical pixels, 16 colors per line
- 640 horizontal by 200 vertical pixels, 4 colors per line

For more detail regarding the exact implementation of the color, see Chapter 16, "QuickDraw II," in Volume 2.

The ImageWriter II can print horizontally at 160 dots per inch and vertically at 144. With a color ribbon, which consists of bands of cyan, magenta, yellow, and black, the ImageWriter II supports eight possible colors per dot. These color dot combinations, except for black, are composed of all the different mixtures of the colors cyan, magenta, and yellow. Black is represented by the black ribbon, and no other color dots include black and another color.

The color screen is initially converted into color pixels before the screen-to-printer transformation occurs. Each pixel consists of a total of 12 bits: 4 bits each for red, blue, and green (RGB). Thus, each color has 16 possible levels, and each pixel has 4096 possible colors (or levels). The first part of the algorithm must convert these RGB pixels into color pixels that the printer understands. These pixels consist of three colors: cyan, magenta, and yellow (CMY).

This conversion routine performs a straightforward matrix multiplication to achieve the translation from RGB to CMY space. All level information is retained, which means that the CMY result is also 4 bits per color. The matrix values used for this translation are described at the end of this section.

After the algorithm has calculated the CMY levels, it enters the second translation phase: converting from 16 levels of each color to a number of color levels supported by the printer. The number of color levels and the resolution (color pixels per inch) can be traded off by the higher-level software routines. For example, at 160 color pixels per inch (ppi), each pixel can represent one of 8 possible colors, but if the resolution is cut in half (80 ppi), then each pixel has 27 possible colors. Table 15-3 illustrates the possible tradeoffs allowed by the low-level driver.

Table 15-3
Resolution, colors, and gray scales

Average resolution (ppi)	Colors per pixel	Levels per ribbon color	Gray-scale levels
160	8	2	2
80	27	3	3
40	125	5	5
20	729	9	9

The gray-scale column in Table 15-3 helps clarify how the colors-per-pixel value is determined. For example, when there are only 40 pixels per inch, there can be 4 possible dots for each pixel (as determined by the 160-dots-per-inch capability of the printer). These 4 possible dots represent five different levels of brightness: 0 dots, 1 dot, 2 dots, 3 dots, and 4 dots can be printed at a time. Regardless of which 3 dots are printed, the eye will perceive the same amount of brightness and color from that pixel. Since there are 3 possible colors (CMY) that can be printed at each dot, this means that there are 5^3, or 125, possible colors.

Table 15-3 specifies average resolution because the dots for each pixel don't always fall on the same printing line. At 40 pixels per inch, each pixel is actually comprised of two horizontal by two vertical dots. At 20 pixels per inch, the pixel consists of four horizontal by two vertical dots. The concepts are illustrated in Figure 15-11.

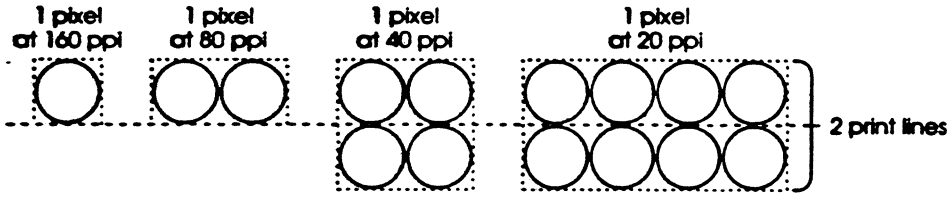


Figure 15-11
Pixels and print lines

The second phase of the color transformation consists of a simple linear translation of the old color-level value (16 possible levels per color) to a new color-level value (either 2, 3, 5, or 9 possible levels, depending on the mode).

At this point, the color data information is defined for printing on the color printer. However, for a black-and-white printer, there must be one more transformation that changes the pixels from color space to black-and-white space. This transformation is performed by another matrix multiplication that converts the three color values (CMY) into a single black-and-white value.

The resultant black-and-white value is actually a gray-scale value that is sent to the printer in the same manner as the color values. The only difference is that each pixel is a specified number of black dots instead of color dots. The level information is retained in the translation from color to black and white.

The horizontal and vertical resolutions, the pixel size, and the page rectangle size of the various modes, and the print quality for the ImageWriter II are summarized in Figure 15-12.

- *Note:* The terms *portrait* and *landscape* refers to the page orientation, representing vertical and horizontal orientations, respectively. The printing area for a page with a gap is 8 x 10 1/2 inches, and that for a page without a gap is 8 x 11 inches.

Dots per pixel	Horizontal resolution	Vertical resolution	Page rectangle
Portrait, 320 mode			
4 x 2 = 8 dots Normal quality	40 ppi/160 dpi	36 ppi/72 dpi	with gap: 0,0,378,320 w/o gap: 0,0,396,320
2 x 1 = 2 dots Best quality	80 ppi/160 dpi	72 ppi/72 dpi	Same as above
Portrait, 640 mode			
2 x 2 = 4 dots Normal quality	80 ppi/160 dpi	36 ppi/72 dpi	with gap: 0,0,378,640 w/o gap: 0,0,396,640
1 dot Best quality	160 ppi/160 dpi	72 ppi/72 dpi	Same as above
Landscape, 320 mode			
4 x 2 = 8 dots Normal quality	36 ppi/72 dpi	40 ppi/160 dpi	with gap: 0,0,320,378 w/o gap: 0,0,320,396
2 x 1 = 2 dots Best quality	72 ppi/72 dpi	80 ppi/160 dpi	Same as above
Landscape, 640 mode			
4 x 1 = 4 dots Normal quality	72 ppi/72 dpi	40 ppi/160 dpi	with gap: 0,0,320,756 w/o gap: 0,0,320,792
2 x 1 = 2 dots Best quality	144 ppi/144 dpi	80 ppi/160 dpi	Same as above

Figure 15-12
Resolution, pixel size, page size, and print quality

Using the Print Manager

This section discusses how the Print Manager routines fit into the general flow of an application and gives you an idea of which routines you'll need to use under normal circumstances. Each routine is described in detail later in this chapter.

The Print Manager depends upon the presence of the tool sets shown in Table 15-4 and requires that at least the indicated version of the tool set be present.

Table 15-4
Print Manager—other tool sets required

Tool set number		Tool set name	Minimum version needed
$01	#01	Tool Locator	1.0
$02	#02	Memory Manager	2.0
$03	#03	Miscellaneous Tool Set	2.0
$04	#04	QuickDraw II	2.0
$05	#05	Desk Manager	1.0
$0E	#14	Window Manager	1.3
$0F	#15	Menu Manager	1.3
$10	#16	Control Manager	1.3
$12	#18	QuickDraw II Auxiliary Routines	1.0
$14	#20	LineEdit Tool Set	1.0
$15	#21	Dialog Manager	1.1
$1B	#27	Font Manager	1.0
$1C	#28	List Manager	1.0

Your application must make a PMStartUp call once before making any other Print Manager calls. Conversely, if the application makes a PMStartUP call, the application must make a PMShutDown call before it quits or before it unloads the Print Manager.

Before you can print a document, you need a valid print record. You can use an existing print record (for instance, one saved with a document), or you can initialize one by calling PrDefault. If you use an existing print record, be sure to call PrValidate to make sure it's valid for the current version number of the Print Manager and for the currently installed printer.

To create a new print record, you must first create a handle to it with the Memory Manager NewHandle routine. A print record is 140 bytes long.

Printer and print record information is obtained via the style and job dialogs:

- Call PrChoosePrinter when the user chooses the Select Printer command from the File menu. No print record is required for this call.
- Call PrStlDialog to get the page dimensions when the user chooses the Page Setup command. From the *rPage* field of the printer information subrecord, you can then determine where page breaks will be in the document. You can show rulers and margins correctly by using the information in the print record *iVRes, iHRes,* and *rPaper* fields.
- Call PrJobDialog to get the specific information about that printing job (such as the page range and number of copies) when the user chooses the Print command.

When the user chooses the Print command, your application normally should immediately start its printing loop in order to conform to the *Human Interface Guidelines: The Apple Desktop Interface*.

The first step of the printing loop is using the PrOpenDoc routine to obtain a pointer to the GrafPort to be used for printing. This must be done only once for each print job.

The next step calls for beginning the inner loop of printing the pages one by one, as detailed in the next section.

Printing loop

To print a document, you call the following routines:

1. PrOpenDoc, which returns a pointer to the GrafPort to be used for printing
2. PrOpenPage, which starts each new page (reinitializing the GrafPort)
3. QuickDraw routines, for drawing the page into the GrafPort whose pointer was returned by PrOpenDoc
4. PrClosePage, which terminates the page
5. PrCloseDoc, at the end of the entire document, to close the GrafPort being used for printing

Each page is either printed immediately (draft printing) or written to the disk or to memory (spool printing). You should test to see whether spooling was done and, if so, print the spooled document. First, your application should use the Memory Manager routine MaxBlock to ensure that a 10K block of memory is available. If it isn't, you should swap out enough memory to allow for that 10K block and then call PrPicFile.

You should check for errors after each Print Manager call. If an error occurs, you should abort printing by setting the error to prAbort using the PrSetError routine. Be sure that your application exits the printing loop normally so that all allocated memory is deallocated accordingly; that is, be sure that PrOpenDoc is matched by PrCloseDoc and that every PrOpenPage is matched by a PrClosePage.

◆ *Note:* The maximum number of pages in a spool file is 16,382. If you need to print more than 16,382 pages at one time, just repeat the printing loop (without calling PrValidate, PrStlDialog, or PrJobDialog).

Printing a specified range of pages

Your application can try to print an entire document even when the user wants to print only a selected subrange of pages. The Print Manager processes each page but actually prints only the pages from *tFstPage* to *tLstPage*.

However, if the application knows the page boundaries in the document, it is much faster to loop through only the specified pages. The application can do this by saving the values of *tFstPage* and *tLstPage* after the PrJobDialog call, recalculating the page range using 1 as the starting page and storing these values into the appropriate fields in the print record. For example, to print pages 20 to 25 of a document, you would set *tFstPage* to 1 and *tLstPage* to 6 and then begin the printing loop at the document's page 20.

Remember that *tFstPage* and *tLstPage* are relative to the first page counted by the Print Manager. The Print Manager counts one page each time PrOpenPage is called; the count begins at 1.

Using QuickDraw II for printing

When drawing into the QuickDraw II GrafPort being used for printing, you should note the following:

- With each new page, you get a completely reinitialized GrafPort, so you'll need to reset font information and other GrafPort characteristics as you want them.
- Don't use clipping to select text to be printed. There are a number of subtle differences between how text appears on the screen and how it appears on the printer; you can't count on knowing the exact dimensions of the rectangle occupied by the text.
- Don't use fixed-width fonts to align columns. Since spacing is adjusted by the printer, you should explicitly move the pen to where you want it.
- Don't make calls that don't do anything on the printer. For example, erase operations are time consuming and normally aren't needed on the printer.

For printing to the LaserWriter, you'll need to observe the following limitations:

- Regions aren't supported; try to simulate them with polygons.
- Clipping regions should be limited to rectangles.
- Invert routines, such as the QuickDraw II routines InvertRect and InvertRgn, aren't supported.
- Copy is the only transfer mode supported for all objects except text and bit images. For text, Bic is also supported. For bit images, the only transfer mode not supported is XOR.
- Don't change the GrafPort's local coordinate system (with SetOrigin) within the printing loop (between PrOpenPage and PrClosePage).

Sequence of events

The following pseudocode will help you understand the entire sequence of calls necessary to print a document. Your application should take the following steps:

1. Call PrChoosePrinter, if user selects Choose Printer menu item.
2. Call PrStlDialog, if user selects Page Setup menu item.
3. Call PrJobDialog, when user selects Print menu item.
4. Call either

   PrDefault, if no existing print record

   or

   PrValidate, if there is an existing print record.
5. Call PrOpenDoc.

   If tool call error, then PrSetError to `prAbort`.
6. Enter printing loop:

   Call PrOpenPage.

   If tool call error, then PrSetError to `prAbort`.

   Call appropriate QuickDraw II routines, including those that reset font information.

   Call PrClosePage.

   If tool call error, then PrSetError to `prAbort`.
7. Repeat loop for each page printed.
8. Call PrCloseDoc.
9. Call PrError; if error not zero, then skip PrPicFile call.
10. Call Memory Manager routine MaxBlock to check for a 10K block, if none, swap out parts of application to make room.
11. Call PrPicFile.

Methods of printing

There are two basic methods of printing documents: draft and spool.

- **Draft printing:** Your QuickDraw II calls are converted directly into command codes the printer understands, which are then immediately used to drive the printer. The LaserWriter always uses draft printing, since the QuickDraw II calls are translated immediately into PostScript commands. The ImageWriter and other unintelligent dot matrix printers are written to in draft mode for text only. High-quality pixel-map images are produced only during spool printing.
- **Spool printing:** The Print Manager processes your printing requests in two steps. First it writes (spools) a representation of your document's printed image to a disk file or to memory. Second, this information is converted into a bit image and printed. This method is used to print graphics on the ImageWriter.

Printer and port drivers

Both the ImageWriter and LaserWriter printers are fully supported. Other printers should work so long as drivers are written for them; these drivers may be developed by Apple or third-party developers.

Printer drivers

Apple provides the printer drivers for the ImageWriter and LaserWriter.

The user can install new printer drivers into the system by saving a printer driver file into the DRIVERS subdirectory within the SYSTEM subdirectory. The printer driver file must be of file type $BB and have an aux type of $0001.

At the time of publication, no more information about printer driver formats was available.

Printer peripheral cards and printer ports

Port drivers are used to support the various methods of connecting a printer to the Apple IIGS. A port driver can be written to work with a built-in port or with a peripheral card in a slot. Currently, the following three types of drivers are defined, all of which support the internal ports:

- Printer.Port
- Modem.Port
- Appletalk.Port

The user can install new port drivers into the system by saving a port driver file into the DRIVERS subdirectory within the SYSTEM subdirectory. The port driver file must be of file type $BB and have an aux type of $0002.

At the time of publication, no more information about port driver formats was available.

Background processing

As already mentioned, the job subrecord includes a pointer, *pIdleProc*, to an optional **background procedure** run whenever the Print Manager has directed output to the printer and is waiting for the printer to finish. The background procedure has no parameters and returns no result; the Print Manager simply runs it at every opportunity.

If you don't designate a background procedure, the Print Manager uses a default procedure for canceling printing. The default procedure polls the keyboard and sets a Print Manager error code if the user types Apple-period. If you use this option, you should display a dialog box during printing to inform the user that the Apple-period option is available.

If you do designate a background procedure, you must set *pIdleProc* after presenting the dialogs, validating the print record, and initializing the GrafPort. The routines that perform these operations reset *pIdleProc* to NIL.

Important

If you write your own background procedure, you must be careful to avoid a number of subtle concurrency problems that can arise. For instance, if the background procedure uses QuickDraw II, it must be sure to restore the GrafPort being used for printing as the current port before returning. It's particularly important not to attempt any printing from within the background procedure: The Print Manager is not reentrant. If you use a background procedure that runs your application concurrently with printing, it should disable all menu items having to do with printing, such as Page Setup and Print.

$0113 PMBootInit

Initializes the Print Manager; called only by the Tool Locator.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C Call must not be made by an application.

$0213 PMStartUp

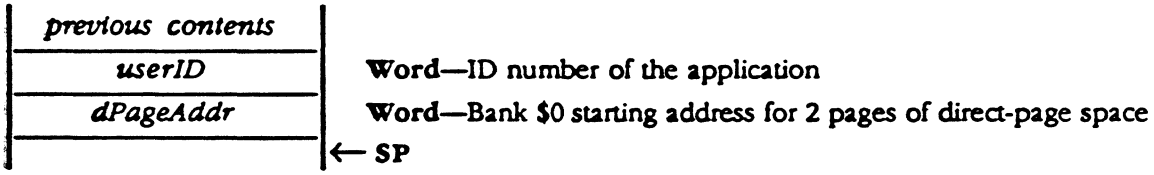
Starts up the Print Manager for use by an application.

Important

Your application must make this call before it makes any other Print Manager calls.

Parameters

Stack before call



Stack after call

Errors

$1301	missingDriver	Specified driver not in DRIVERS subdirectory of SYSTEM subdirectory
System Loader errors		Returned unchanged
Tool Locator errors		Returned unchanged
Memory Manager errors		Returned unchanged

C

```
extern pascal void PMStartUp(userID,dPageAddr)
Word     userID;
Word     dPageAddr;
```

$0313 PMShutDown

Shuts down the Print Manager.

Important
If your application has started up the Print Manager, the application must make this call before it quits.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

C

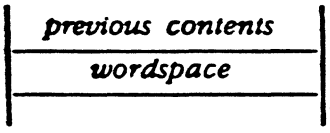
```
extern pascal void PMShutDown()
```

$0413 PMVersion

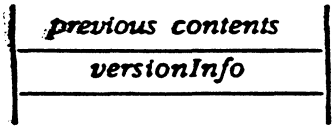
Returns the version number of the Print Manager.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal Word PMVersion()
```

$0513 PMReset

Resets the Print Manager; called only when the system is reset.

Warning
An application must never make this call.

Parameters The stack is not affected by this call. There are no input or output parameters.

Errors None

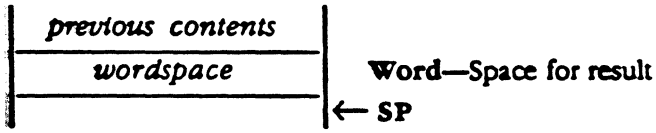
C Call must not be made by an application.

$0613 PMStatus

Indicates whether the Print Manager is active.

Parameters

Stack before call



Stack after call

Errors None

C

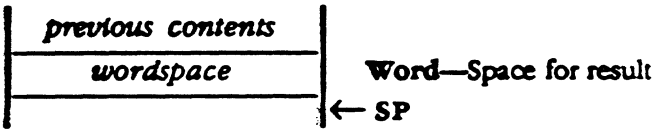
```
extern pascal Boolean PMStatus()
```

$1613 PrChoosePrinter

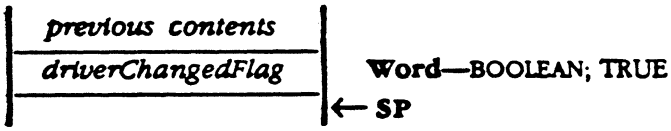
Conducts a chooser dialog with the user to determine what printer and port driver to use. The *driverChangedFlag*, if TRUE, signals that the user has selected a different driver than the one currently being used.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal Boolean PrChoosePrinter()
```

$0F13 PrCloseDoc

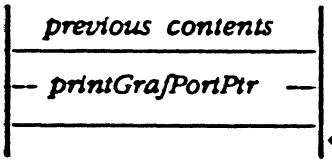
Closes the GrafPort being used for printing. For draft printing, PrCloseDoc ends the printing job.

For spool printing, PrCloseDoc ends the spooling process; the spooled document must now be printed. Before printing it, your application should call the PrError routine to find out whether spooling succeeded. If spooling did succeed, the application must make sure that a 10K block of memory is available to allow for the printing buffer and then call the PrPicFile routine.

◆ *Note:* Your application can use the Memory Manager routine MaxBlock to ensure that a 10K block is available. See the section "MaxBlock" in Chapter 12, "Memory Manager."

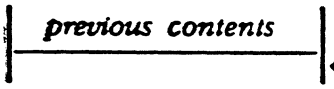
Parameters

Stack before call



Long—POINTER to GrafPort to be used for printing

Stack after call



Errors

$1302	portNotOn	Specified port not selected in Control Panel

C

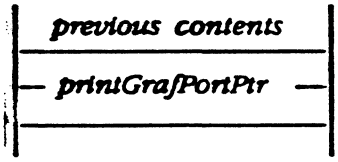
```
extern pascal void PrCloseDoc(printGrafPortPtr)
GrafPortPtr     printGrafPortPtr;
```

$1113 PrClosePage

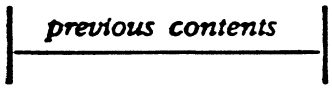
Ends the printing of the current page. The routine signals the Print Manager that your application is finished with this page, so that the Print Manager can perform whatever close operations are required for the current printer and printing method.

Parameters

Stack before call



Stack after call



Errors

$1302	portNotOn	Specified port not selected in Control Panel

C

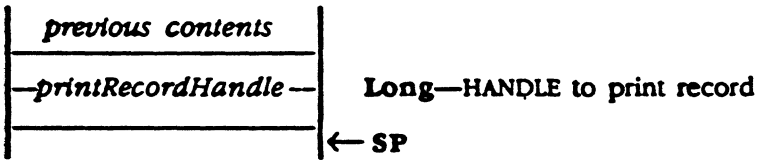
```
extern pascal void PrClosePage(printGrafPortPtr)
GrafPortPtr     printGrafPortPtr;
```

$0913 PrDefault

Fills the fields of the specified print record with default values for the current printer. The record with the *printRecordHandle* may be a new or an existing print record.

Parameters

Stack before call



Stack after call

Errors

$1303	noPrintRecord	No print record was specified
Memory Manager errors		Returned unchanged

C

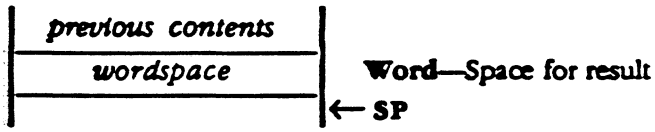
```
extern pascal void PrDefault(printRecordHandle)
PrRecHndl     printRecordHandle;
```

$2313 PrDriverVer

Returns the version number of the currently installed printer driver.

Parameters

Stack before call



Stack after call

Errors None

C

```
extern pascal Word PrDriverVer()
```

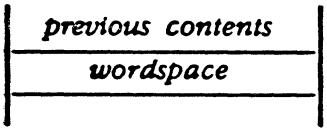
$1413 PrError

Returns the last printer error code left during the printing loop by Print Manager routines. In addition to the tool set error codes (see Table 15-7 in the section "Print Manager Summary" at the end of this chapter), other possible error codes are as follows:

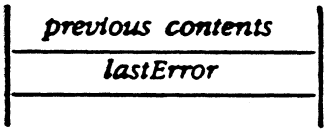
No error	$00
prAbort	$80

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal Word PrError()
```

$0C13 PrJobDialog

Conducts a job dialog with the user to determine the print quality, range of pages to print, and so on. The initial settings displayed in the dialog box are taken from the printer driver, where they were retained from the previous job (with the exception of the page range, which is set to All, and the number of copies, which is set to 1).

If the user confirms the dialog, the print record is updated, the PrValidate routine is automatically called, and the routine returns TRUE. Otherwise, the print record is left unchanged, and the routine returns FALSE.

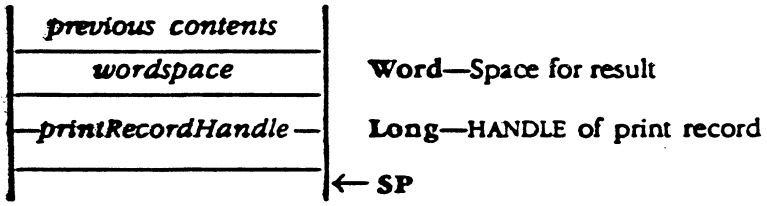
Important

Never call PrJobDialog between the PrOpenPage and PrClosePage calls.

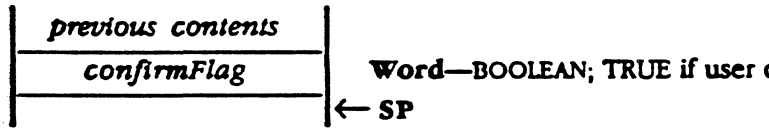
✥ *Note:* Since the job dialog is associated with the Print command, you should proceed with the requested printing operation if *confirmFlag* is TRUE.

Parameters

Stack before call



Stack after call



Errors Memory Manager errors Returned unchanged

C

```
extern pascal Boolean PrJobDialog(printRecordHandle)
PrRecHndl     printRecordHandle;
```

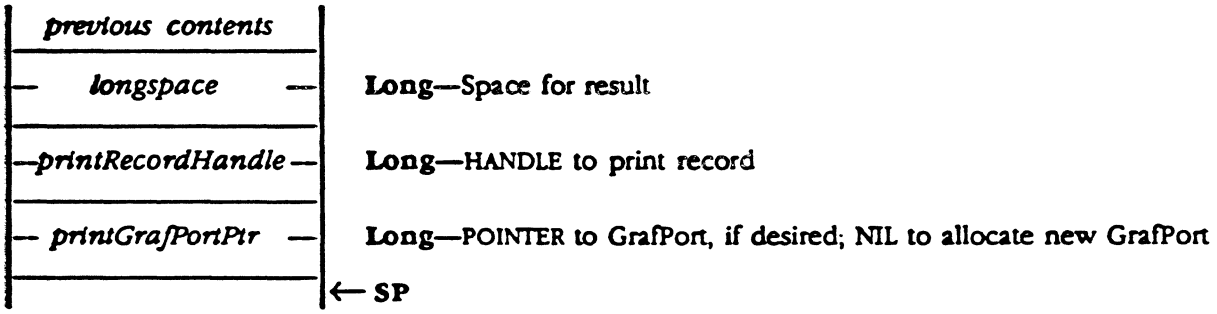
$0E13 PrOpenDoc

Initializes a GrafPort for use in printing a document, makes it the current port, and returns a pointer to the port.

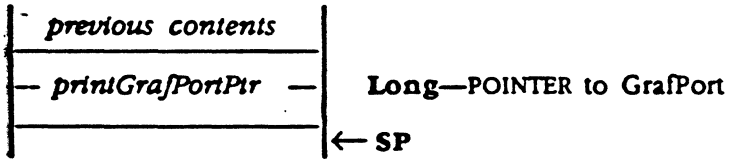
Important
You must balance every call to PrOpenDoc with a call to PrCloseDoc.

Parameters

Stack before call



Stack after call



Errors			
	$1302	`portNotOn`	Specified port not selected in Control Panel
	$1304	`badLaserPrep`	The version of the LaserPrep file in the LaserWriter is not compatible with this version of the Print Manager
	$1305	`badLPFile`	The version of the LaserPrep file in the DRIVERS subdirectory of the SYSTEM subdirectory is not compatible with this version of the Print Manager
	$1306	`papConnNotOpen`	Connection can't be established with the LaserWriter
	$1307	`papReadWriteErr`	Read-write error on the LaserWriter
	Memory Manager errors		Returned unchanged
	ProDOS errors		Returned unchanged

C

```
extern pascal GrafPortPtr PrOpenDoc(printRecordHandle,printGrafPortPtr)
PrRecHndl     printRecordHandle;
GrafPortPtr     printGrafPortPtr;
```

More about PrOpenDoc parameters

The print record with the *printRecordHandle* will be used for this printing operation; you should already have validated this print record.

Depending on the setting of the *bJDocLoop* field in the job subrecord, the GrafPort to be used for printing will be set up for draft or spool printing. For spool printing, the spool file name, volume reference number, and version number are taken from the job subrecord.

The *printGrafPortPtr* parameter is normally NIL. If the application passes a pointer to its own Grafport, the Print Manager will use it; otherwise, the routine allocates a new GrafPort and returns a pointer to that GrafPort.

$1013 PrOpenPage

Begins a new page. The page is printed only if it falls within the page range given in the job subrecord.

Important

You must balance every call to PrOpenPage with a call to PrClosePage.

For spool printing, *pageFramePtr* points to a rectangle to be used as the QuickDraw II picture frame for this page. This rectangle can be used for scaling. When you print the spooled document, this rectangle will be scaled (with the QuickDraw II Auxiliary DrawPicture routine) to coincide with the *rPage* rectangle in the printer information subrecord. Unless you want the printout to be scaled, you should set *pageFramePtr* to NIL. This uses the *rPage* rectangle as the picture frame, so that the page will be printed with no scaling.

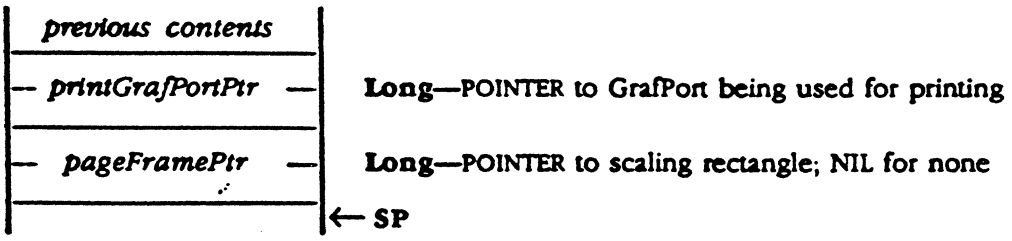
Important

Don't call the QuickDraw II Auxiliary routine OpenPicture while a page is open (after a PrOpenPage call and before the following PrClosePage call). You can, however, call the QuickDraw II Auxiliary routine DrawPicture at any time.

Also, the GrafPort is completely reinitialized by PrOpenPage. Therefore, after a PrOpenPage call, you must set GrafPort features such as the font and font size for every page that you draw.

Parameters

Stack before call



Stack after call

Errors	$1302	portNotOn	Specified port not selected in Control Panel
	Memory Manager errors		Returned unchanged

C

```
extern pascal void PrOpenPage(printGrafPortPtr,pageFramePtr)
GrafPortPtr     printGrafPortPtr;
Rect *pageFramePtr;
```

$1213 PrPicFile

Prints a spooled document. If spool printing is being used, your application should normally call PrPicFile after it calls the PrCloseDoc routine.

The print record should be the printer record already used for this job. The spool file name and version number are taken from the job subrecord of this print record. After printing is successfully completed, the Print Manager deletes the spool file from the disk.

You'll normally pass NIL for the *printGrafPortPtr*. The Print Manager will then allocate the memory blocks it needs from the Memory Manager.

Important

Be sure not to pass a pointer to the same GrafPort you received from PrOpenDoc. If that port was allocated by the PrOpenDoc routine itself (that is, if the printGrafPortPtr to PrOpenDoc was NIL), then the PrCloseDoc routine will have disposed of the port, making your pointer to it invalid. If you provided your own memory for PrOpenDoc, you can use the same memory again for PrPicFile.

Parameters

Stack before call

previous contents	
printRecordHandle	**Long**—HANDLE to print record
printGrafPortPtr	**Long**—POINTER to printing GrafPort; NIL for default allocation
statusRecPtr	**Long**—POINTER to printer status record (see Figure 15-13)
	← **SP**

Stack after call

previous contents	
	← **SP**

Errors	$1302	`portNotOn`	Specified port not selected in Control Panel
	Memory Manager errors		Returned unchanged

C

```
extern pascal void PrPicfile(printRecordHandle,printGrafPortPtr,
statusRecPtr)
PrRecHndl     printRecordHandle;
GrafPortPtr     printGrafPortPtr;
PrStatusRecPtr     statusRecPtr;
```

Printer status record

The PrPicFile routine uses the printer status record pointed to by *statusRecPtr* to report on its progress. The record is 28 bytes long and is shown in Figure 15-13. Your background procedures (if any) can use this record to monitor the state of the printing operation.

The *fPgDirty* field is TRUE if anything has already been printed on the current page; otherwise, the field is FALSE.

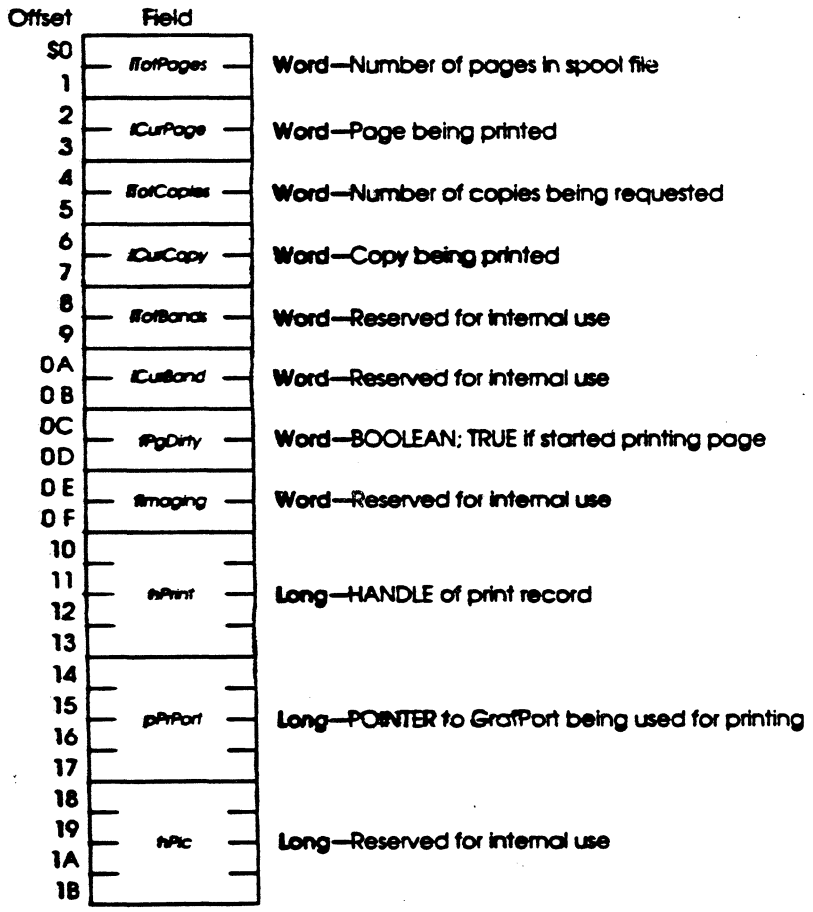


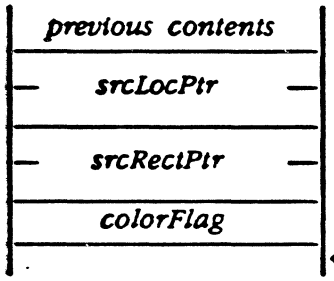
Figure 15-13
Printer status record

$0D13 PrPixelMap

Prints all or part of a specified pixel map.

Parameters

Stack before call

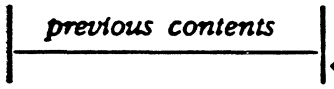


Long—POINTER to source LocInfo containing POINTER to pixel map

Long—POINTER to rectangle enclosing portion of pixel map to print

Word—BOOLEAN; TRUE if printing in color, FALSE if in black and white

Stack after call



Errors

$1302	portNotOn	Specified port not selected in Control Panel
$1304	badLaserPrep	The version of the LaserPrep file in the LaserWriter is not compatible with this version of the Print Manager
$1305	badLPFile	The version of the LaserPrep file in the DRIVERS subdirectory of the SYSTEM subdirectory is not compatible with this version of the Print Manager
$1306	papConnNotOpen	Connection can't be established with the LaserWriter
$1307	papReadWriteErr	Read-write error on the LaserWriter
Memory Manager errors		Returned unchanged
QuickDraw II errors		Returned unchanged

C

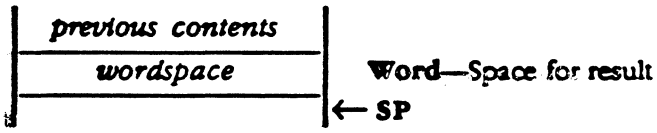
```
extern pascal void PrPixelMap(srcLocPtr,srcRectPtr,colorFlag)
LocInfoPtr     srcLocPtr;
RectPtr     srcRectPtr;
Boolean             colorFlag;
```

$2413 PrPortVer

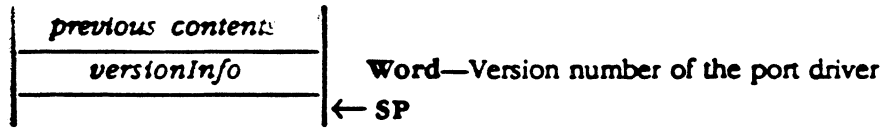
Returns the version number of the currently installed port driver.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal Word PrPortVer()
```

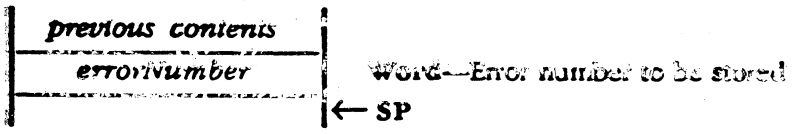
$1513 PrSetError

Stores a specified value into the global variable where the Print Manager keeps its printer error code.

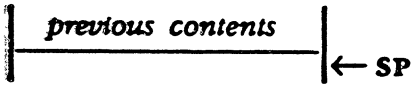
For example, you can use this procedure to abort a printing operation in progress by setting the error code to `prAbort`.

Parameters

Stack before call



Stack after call



Errors None

C

```
extern pascal void PrSetError(errorNumber)
Word     errorNumber;
```

$0B13 PrStlDialog

Conducts a style dialog with the user to determine the page dimensions and other information needed for page setup. The initial settings displayed in the dialog box are taken from the print record. If the user confirms the dialog, the results of the dialog are saved in the specified print record, the PrValidate routine is automatically called, and the routine returns TRUE. Otherwise, the print record is left unchanged, and the routine returns FALSE.

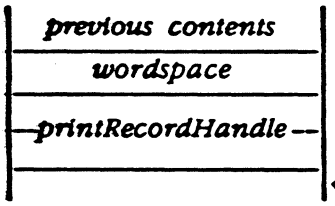
Important

Never call PrStlDialog between PrOpenPage and PrClosePage calls.

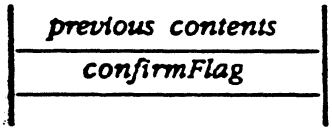
◆ *Note:* If the print record was taken from a document, you should update its contents in the document if confirmFlag is TRUE. This causes the results of the style dialog to remain with the document.

Parameters

Stack before call



Stack after call



Errors Memory Manager errors Returned unchanged

C

```
extern pascal Boolean PrStlDialog(printRecordHandle)
PrRecHndl     printRecordHandle;
```

$0A13 PrValidate

Checks the contents of the specified print record for compatibility with the current version number of the Print Manager and the currently installed printer. If the record is valid, the routine returns FALSE (no change). If the record is invalid, the record is adjusted to the default values for the current printer, and the routine returns TRUE.

Important

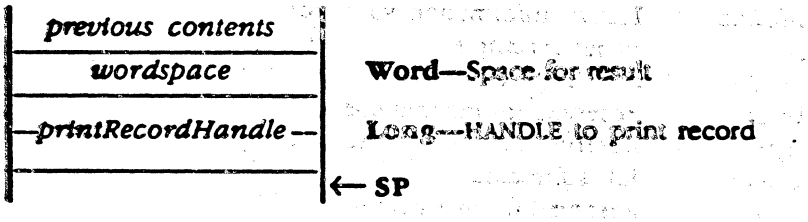
Never call PrValidate (or PrStlDialog or PrJobDialog, which call it) between PrOpenPage and PrClosePage calls.

If the current print record is set for an ImageWriter while a LaserWriter is selected or vice versa, PrValidate calls PrDefault to set the print record for the appropriate printer.

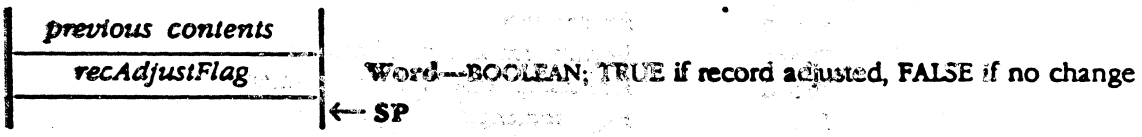
PrValidate also makes sure all the information in the print record is internally self-consistent and updates the print record as necessary. These changes do not affect the routine's Boolean result.

Parameters

Stack before call



Stack after call



Errors Memory Manager errors Returned unchanged

C

```
extern pascal Boolean PrValidate(printRecordHandle)
PrRecHndl     printRecordHandle;
```

Print Manager summary

This section briefly summarizes the constants, data structures, and tool set error codes contained in the Print Manager.

Important

These definitions are provided in the appropriate interface file.

Table 15-5
Print Manager constants

Name	Value	Description
Printer error codes		
prAbort	$80	Abort printing

Table 15-6
Print Manager data structures

Name	Offset	Type	Definition
Print record (PrRec)			
prVersion	$0	Word	Version number of the printer driver
prInfo	$2	PrInfoRec	Printer information subrecord
rPaper	$10	Rect	Paper rectangle
prStl	$18	PrStyleRec	Style subrecord
prInfoPT	$2A	14 bytes	Reserved for internal use
prXInfo	$38	24 bytes	Reserved for internal use
prJob	$50	PrJobRec	Job subrecord
printX	$64	38 bytes	Reserved for internal use
iReserved	$8A	Word	Reserved for internal use
Printer information subrecord (PrInfoRec)			
iDev	$0	Word	Printer type
iVRes	$2	Word	Vertical resolution of printer
iHRes	$4	Word	Horizontal resolution of printer
rPage	$6	Rect	Page rectangle

Table 15-6 (continued)
Print Manager data structures

Name	Offset	Type	Definition
Printer style subrecord (PrStyleRec)			
wDev	$0	Word	Output-quality information
internA	$2	3 Words	Reserved for internal use
feed	$8	Word	Paper feed type
paperType	$A	Word	Paper type
crWidth	$C	Word	Carriage width for ImageWriter
vSizing	$C	Word	Vertical sizing for LaserWriter
reduction	$E	Word	Percent reduction, LaserWriter only
internB	$10	Word	Reserved for internal use
Job information subrecord (PrJobRec)			
iFstPage	$0	Word	First page to print
iLstPage	$2	Word	Last page to print
iCopies	$4	Word	Number of copies
bJDocLoop	$6	Byte	Printing method
fFromUser	$7	Byte	Reserved for internal use
pIdleProc	$9	Pointer	Pointer to background procedure
pFileName	$D	Pointer	Spool file pathname
iFileVol	$11	Word	Spool file volume reference number
bFileVers	$13	Byte	Spool file version number
bJobX	$14	Byte	Reserved for internal use
Printer status subrecord (PrStatusRec)			
iTotPages	$0	Word	Number of pages in spool file
iCurPage	$2	Word	Page being printed
iTotCopies	$4	Word	Number of copies requested
iCurCopy	$6	Word	Copy being printed
iTotBands	$8	Word	Reserved for internal use
iCurBand	$A	Word	Reserved for internal use
fPgDirty	$C	Boolean	TRUE if started printing page
fImaging	$E	Word	Reserved for internal use
hPrint	$10	PrRecHndl	Handle of print record
pPrPort	$14	GrafPortPtr	Pointer to GrafPort being use for printing
hPic	$18	Long	Reserved for internal use

Note: The actual assembly-language equates have a lowercase *o* (the letter) in front of all of the names given in this table.

Table 15-7
Print Manager error codes

Code	Name	Description
$1301	missingDriver	Specified driver not in the DRIVERS subdirectory of the SYSTEM subdirectory
$1302	portNotOn	Specified port not selected in the control panel
$1303	noPrintRecord	No print record specified
$1304	badLaserPrep	The version of the LaserPrep file in the LaserWriter is not compatible with this version of the Print Manager
$1305	badLPFile	The version of the LaserPrep file in the DRIVERS subdirectory of the SYSTEM subdirectory is not compatible with this version of the Print Manager
$1306	papConnNotOpen	Connection can't be established with the LaserWriter
$1307	papReadWriteErr	Read-write error on the LaserWriter
$1321	startUpAlreadyMade	LLDStartUp call already made
$1322	invalidCtlVal	Invalid control value specified

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using the Apple Macintosh™ Plus and Microsoft® Word. Proof and final pages were created on the Apple LaserWriter® Plus. POSTSCRIPT®, the LaserWriter page-description language, was developed by Adobe Systems Incorporated. Line drawings were created with Adobe Illustrator™.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic®. Bullets are ITC Zapf Dingbats®. Program listings are set in Apple Courier, a monospaced font.

Last page of text (p. 776 — p. 775 is blank)