Description of the Cortland Tools: Part I Preliminary Notes

Preliminary Notes: 1/30/86

Writer: Willam H. Harris
Apple User Education

Changes Since Last Draft

This is the first draft of this document. The sources used to prepare this document are as follows:

Tool Locator ERS	12/3/85
QuickDraw II ERS	1/15/86
Memory Manager ERS	11/27/85
Event Manager ERS	11/25/85
Miscellaneous Tools	1/10/86

Contents

Preface

About This Manual

This manual describes the Apple Cortland Tools available in ROM for the application developer. Under most circumstances, you don't need to know whether a tool is in ROM or RAM; we describe the ROM-based tools in this book simply as a matter of convenience.

For the description of the RAM-based Apple tools, refer to *Description of the Cortland Tools, Part II* (that document is not yet available). For more detailed information about how to write tools, refer to *Using and Writing Cortland Tools*.

Please note that the information presented in this manual is preliminary. It may change before final release of the product and the manual.

Conventions in the Function Descriptions

The description of each function in this book is presented in the following format:

ToolCall Brief description of the function of the tool.

input	*Param1*	TYPE
input	*Param2*	TYPE
output	*Param3*	TYPE

Further description of the function and the *Params*, if necessary.

The TYPE indicates the data type for the element and can be any one of the following:

- BYTE assembles a byte containing the given expression value.
- WORD assembles a 2-byte word containing the given value.
- LONG assembles a 4-byte location containing the given value.
- BLOCK reserves a block of storage consisting of a specified number of bytes.
- INTEGER.
- LONGINT is a long integer.
- POINTER.
- HANDLE is a pointer to a pointer.

Chapter 1

ROM Tool Overview

This chapter will eventually present an overview of the Cortland tools always present in ROM.

Chapter 2

Tool Locator

Introduction

This chapter describes the tool whose job it is to allow tools and applications to communicate among themselves; this tool is called the Tool Locator. If you are simply using the Cortland tools that Apple provides, you won't need to call any functions in the Tool Locator, nor will you see any evidence of it under normal circumstances. You only need to know how the Tool Locator works if you are writing your own tool set.

Addressing Tool Sets and Functions

Each tool is assigned a permanent tool number. Assignment starts at one and continues with each successive integer. Each function within a tool is assigned a permanent function number. For the functions within each tool, assignment starts at one and continues with each successive integer. Thus, each function has a unique, permanent identifier of the form (TSNum,FuncNum). Both the TSNum and FuncNum are 8-bit numbers.

So far, the following numbers have been assigned:

TSNum	**Descriptions**
1	Tool Locator (ROM resident)
2	Memory Manager (ROM resident)
3	Misc. Tools (ROM resident)
4	Graphics Core Routines (ROM resident)
5	Event Manger (ROM resident)
6	ProDOS-16 (RAM resident)

For each tool, the following calls must be present:

FuncNum	**Descriptions**
1	boot initialization function for each tool
2	application startup function for each tool
3	application shutdown function for each tool
4	version information

Each tool has a boot initialization function that is executed at boot time by either the ROM startup code or the ProDOS startup code. In addition, each tool has an application startup function, an application shutdown function to allow an application to turn each tool "on" and "off", and a version call that returns information about the version of the tool.

All tools return version information in the form of a word. The high byte of the word indicates the major release number (starting with 1). The low byte of the word indicates the minor release number (starting with 0). The most significant bit of the word indicates whether the code is an official release or a prototype (no distinction between alpha, beta, or other prototype releases is made).

```
   P    Major       Minor
|  |          |              |
```

Structure of the Tool Locator

The Tool Locator requires no fixed ROM locations and a few fixed RAM locations. All functions are accessed through the tool locator via their tool set number and function number. The Tool Locator uses the tool set number to find an entry in the Tool Pointer Table (TPT). This table contains pointers to Function Pointer Tables (FPT). Each tool set has an FPT containing pointers to the individual functions in the tool. The Tool Locator uses the function number to find the address of the function being called.

Each tool in ROM has an FPT in ROM. There is also a TPT in ROM pointing to all the FPT's in ROM. One fixed RAM location is used to point to this TPT in ROM. This location is initialized at power up and warm boot by the firmware. In this way the address of the TPT in ROM does not ever have to be fixed.

The TPT has the following form:

```
Count  (4 bytes)
Pointer to TS 1        (4 bytes)
Pointer to TS 2        (4 bytes)
...
```

An FPT has the following form:

```
Count  (4 bytes)
(Pointer to F1) - 1    (4 bytes)
(Pointer to F2) - 1    (4 bytes)
...
```

In both tables, the count is the number of entries plus 1.

Tools are to obtain any memory they need dynamically (using as little fixed memory as possible). To use memory obtained through a memory manager, a tool needs some way to find out where its data structures are. The tool locator system maintains a table of work area pointers for the individual tools. The Work Area Pointer Table (WAPT) is a table of pointers to the work areas of individual tools. Each tool will have an entry in the WAPT for its own use. Entries are assigned by tool number (tool four has entry four and so on). A pointer to the WAPT must be kept in RAM at a fixed memory location so that space for the table can be allocated dynamically. At firmware initialization time, the pointer to the WAPT is set to zero.

The tool locator system permanently reserves some space in bank $E1). It is used as follows:

(4 bytes) Pointer to the active TPT. The pointer is to the ROM-based TPT if there are no RAM-based tool sets and no RAM-based ROM patches. Otherwise, it will point to a RAM-based TPT.

(4 bytes) Pointer to the active user's TPT. This pointer is zero initially, indicating that no user tools are present.

(4 bytes) Pointer to the Work Area Pointer Table (WAPT). The WAPT parallels the TPT. Each WAPT entry is a pointer to a work area assigned to the corresponding tool set. At startup time, each WAPT entry is set to zero, indicating no assigned work area.

(4 bytes) Pointer to the user's Work Area Pointer Table (WAPT).

(16 bytes) Entry points to the dispatcher.

This is the only RAM permanently reserved by the tool locator system.

Tool Locator System Initialization

Each tool set must be initialized before use by application programs. Two types of initialization are needed: boot initialization and application initialization. Boot initialization occurs at system startup time (boot time); regardless of the applications to be executed, the system calls the boot initialization function of every tool set. Thus, each tool set must have a boot initialization routine (FuncNum = 1), even if it does nothing. This function has no input or output parameters.

Application initialization occurs during application execution. The application calls the application startup function (FuncNum=2) of each tool set that it will use. The application startup function performs the chores needed to start up the tool set so the application can use it. This function may have inputs and outputs. Each tool set will define what they are. A common input will be the address of space in bank zero that the tool can use.

The application shutdown function (FuncNum=3) should be executed as soon as the application no longer needs to use the tool. The shutdown releases the resources used by the tool. As a precaution against applications that forget to execute the shutdown function, the startup function should either execute the shutdown function itself or do something else to assure a reasonable startup state. This function may have inputs and outputs as well. Again they are defined by the individual tools.

The provision of two initialization times reflects the needs of currently envisioned tools. For example, the Memory Manager requires boot time initialization because it must operate properly even before any application has been loaded. On the other hand, SANE needs to be initialized only if the system executes some application or desk accessory that uses it. Initializing only the tool sets that will be used saves resources, particularly RAM.

Disk and RAM Structure of Tools

This section will eventually discuss additional details of dynamically loaded, RAM-based tool sets. The exact form of tools on disk is undecided at this time.

The Tool Locator Calls

BootInit Initializes the Tool Locator and all other ROM-based Tool Sets.

AppInit Does nothing.

AppEnd Does nothing.

Version Returns the version of the Tool Locator.

output	*Version*	WORD

GetTsPtr Returns pointer to the Function Pointer Table of the specified tool set.

input	*UserOrSystem*	WORD
input	*TSNum*	WORD
input	*Pointer*	POINTER

SetTSPtr Installs the pointer to a Function Pointer Table in the appropriate Tool Pointer Table.

input	*UserOrSystem*	WORD
input	*TSNum*	WORD
input	*Pointer*	POINTER

If the TPT is not yet in RAM, this tool copies the TPT to RAM. (Memory for the TPT is obtained from the Memory Manager.) If there is not enough room in the TPT for the new entry, the TPT is moved to a bigger chunk of memory. Likewise, the WAPT table is expanded if necessary (memory for the expansions is obtained from the Memory Manager). If the new pointer table has any zero entries, old entries are moved from the old pointer table to the new pointer table.

The call can be used to patch a portion of a Tool Set, rather than replacing the Tool Set entirely.

GetFuncPtr

Returns pointer to the specified function in the specified Tool Set.

input	*TSNum*	WORD
input	*FuncNum*	WORD
output	*Pointer*	POINTER

GetWAP

Gets the pointer to the work area for the specified module.

input	*UserOrSystem*	WORD
input	*TSNum*	WORD
output	*Pointer*	POINTER

SetWAP

Sets the pointer to the work area for the specified module.

input	*UserOrSystem*	WORD
input	*TSNum*	WORD
output	*Pointer*	POINTER

Chapter 3

QuickDraw II

Overview

QuickDraw II includes calls for manipulating the graphics environment and drawing primitive graphic objects. Included in the graphics environment is information about: drawing location, the coordinate system, and clipping.

The primitive objects supported are horizontal lines and pixel images. Additionally, lines, rectangles, and regions are supported as higher-level graphics objects. All higher-level objects are drawn using the lower-level horizontal lines.

The horizontal line-drawing routines draw with patterns. A pattern is a 64-pixel image organized as an 8x8 pixel square that can define a repeating design. When a pattern is drawn, it is aligned so that adjacent areas of the same pattern in the same graphics port will blend with it into a continuous, coordinated pattern.

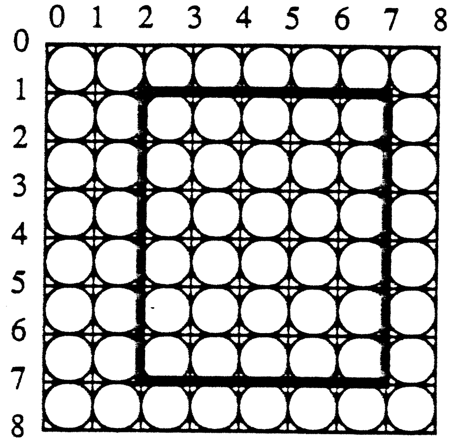
Basic Concepts and Terminology

A pixel map is an area of memory containing a graphic image (the analogous QuickDraw term is BitImage). This image is organized as a rectangular grid of dots called picture elements, or pixels. Each pixel has an assigned value or color. The number of colors a pixel may have depends on its size or chunkiness. Two sizes are possible: four-color and sixteen-color. Exactly which colors map into the various pixel values is determined by a color table, as described under *Color Table* later in this chapter.

Pixel size in the display is controlled independently for each scan line. Each scan line has a scan line control byte (SCB) which determines the scan line's properties. See Appendix B for more details.

Pixels are frequently thought of as points in the Cartesian coordinate system, with each pixel assigned a horizontal and vertical coordinate. Following the QuickDraw standard as established for the Macintosh, the coordinate grid falls between, rather than on pixels. (See Figure 1.) Each pixel is associated with the point that is above and to the left of it.

Figure 1
Pixels, Points and Rectangles



The rectangle is defined by the points (2,1) and (7,7).

It encloses 30 pixels.

○ A pixel

+ A Point

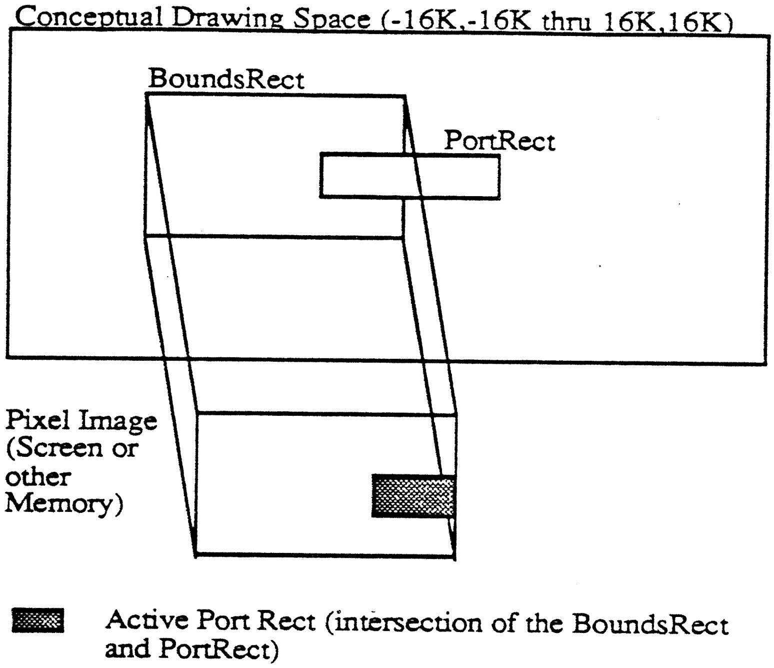
□ A Rectangle

This scheme allows a rectangle to divide pixels into two classes: those that fall within the rectangle and those which fall outside the rectangle.

A pixel map need not be the area of memory associated with the graphics screen. QuickDraw II can treat other memory as pixel map memory and draw into it as easily as into the screen memory.

Drawing can be done in coordinates appropriate to the data being used. Data is mapped from drawing space to the pixel map according to the information kept in two rectangles; the Bounds Rectangle (BoundsRect) and the Port Rectangle (PortRect). Figure 2 illustrates the Bounds and Port Rectangles.

Figure 2. The Bounds and Port Rectangles



The BoundsRect is a rectangle that encompasses the entire pixel map. The upper left corner of the BoundsRect is the point that is above and to the left of the first pixel in the pixel map.

The PortRect is a rectangle that describes the "active" region of the pixel map. The intersection of these two rectangles is the only place that pixels in the pixel map will change (ignoring the VisRgn and ClipRgn, discussed in the following paragraphs).

A **SetOrigin** call allows you to change both these rectangles. Their points remain in the same relative location but the upper left corner (the origin) of the PortRect is set to the point passed by **SetOrigin**.

Drawing is the process by which pixels are altered in a pixel map. You may imagine a pen drawing the image by placing dots of the appropriate color at each pixel that falls under its path.

Drawings are clipped when instructions to draw in inactive parts of the drawing space are ignored. For example, if you are clipping to a rectangle defined by (100,100) and (200,200) and I try to draw a line from (0,0) to (1000,1000), only the pixels that fall inside the (100,100) through (200,200) range are affected.

QuickDraw II also provides for clipping to arbitrary regions. Drawings are clipped to the intersection of two regions: the ClipRgn (a user-maintained clipping region) and the VisRgn (a system-maintained clipping region). This clipping works on the Cortland in the same manner as it does on the Macintosh.

Slabs and Slices

Graphics objects are drawn one scan line at a time. For objects drawn with patterns, the part of the object drawn on a scan line is a "Slab". For objects drawn from other pixel maps, the part of the object drawn on a scan line is a "Slice". The routines that draw slabs and slices can be accessed outside the ROM.

The Drawing Environment

The drawing environment is a set of rules that explain how drawing actions behave. The environment includes information about where drawing will occur (what part of memory, its chunkiness), in what coordinate system, how it will be clipped, the pen state, the font state, and some pointer information. The various parts of the drawing environment are described in this section.

Drawing Location

QuickDraw II allows drawing anywhere in memory. The most common location may be the super hi-res screen, but a pixel map anywhere in memory and of almost any size is acceptable as long as the entire destination pixel map is in a single bank.

PortSCB — Flag to indicate chunkiness of pixel map and master color palette.

Pointer to the pixel map — Points to the first byte in the pixel map.

Width — Number of bytes in a row of pixels (QuickDraw term is RowBytes).

BoundsRect — Rectangle that describes the extent of the pixel map and imposes a coordinate system on it.

PortRect — Rectangle that describes the active area of Data space.

Pen State

QuickDraw II maintains a graphics pen (position and size). Its position is used for drawing text, and its size is used for determining the size of a frame. Quickdraw II does two kinds of drawing; normal drawing and erasing. In normal drawing, the destination pixel map depends on what it was to start with, the original fill pattern or pixel image and the drawing mode. Erasing just fills the affected pixels with the background pattern.

Pen Location -- A point in data space.

Pen Size -- A point describing the width and height of the pen.

Pattern Transfer Mode -- One of the eight transfer modes supported by the Primitives. This mode is used when drawing horizontal lines with the fill pattern.

Fill Pattern -- The fill pattern is used when drawing horizontal lines. When any routine uses the horizontal line drawing routine to draw an object, the object will appear in this pattern.

Background Pattern -- The background pattern is used when erasing horizontal lines. When any routine erases horizontal lines in the shape of an object, that object will appear in this pattern.

Pen Modes

There are eight different pen modes. These modes are used to derive the color of a pixel when it is being drawn to. Each pixel is made up of a series of bits. The pen operates on the individual bits in a pixel as single units. In this way logical binary operations are well defined.

The following pen modes are available. (Each 1 and 0 is the value of a bit in a pixel.)

Mode 0 (pencopy) Copy pen to destination. This is the typical drawing mode.

pencopy		Pen 0	Pen 1
Dest.	0	0	1
	1	0	1

Mode 1 (penOR) Overlay (OR) pen and destination. You can use this mode to non-destructively overlay new images on top of existing images.

penOR		Pen 0	Pen 1
Dest.	0	0	1
	1	1	1

Mode 2 (penXOR) Exclusive or (XOR) pen with destination. You can use this mode for cursor drawing and rubber-banding. If an image is drawn in penXOR mode, the appearance of the destination at the image location can be restored merely by drawing the image again in penXOR mode.

penXOR		Pen 0	Pen 1
Dest.	0	0	1
	1	1	0

Mode 3 (penBIC)

Bit Clear (BIC) pen with destination ((NOT pen) AND destination). You can use this mode to explicitly erase (turn off) pixels, often prior to overlaying another image.

penBIC		Pen 0	Pen 1
Dest.	0	0	0
	1	1	0

The following modes are inverses of the above modes; that is, the pen color is inverted prior to performing the associated operation.

Mode 4 (notpencopy)

Copy inverted pen to destination. You can use this mode to draw inverted images.

notpenCOPY		Pen 0	Pen 1
Dest.	0	1	0
	1	1	0

Mode 5 (notpenOR)

Overlay (OR) inverted pen with destination. You can use this mode to overlay inverted images.

notpenOR		Pen 0	Pen 1
Dest.	0	1	0
	1	1	1

Mode 6 (notpenXOR)

Exclusive or (XOR) inverted pen with destination. This mode behaves similarly to penXOR mode.

notpenXOR		Pen 0	Pen 1
Dest.	0	1	0
	1	0	1

Mode 7 (notpenBIC)

Bit Clear (BIC) inverted pen with destination (pen AND destination). You can use this mode to display the intersection of two images.

notpenBic		Pen 0	Pen 1
Dest.	0	0	0
	1	1	0

Clipping

As stated earlier, a drawing may be clipped to a variety of rectangles and regions.

GrafProcs and GrafPort

QuickDraw II's local environment includes clipping information, handles to pictures, regions, and polygons, as well as a pointer to the GrafProcs record. The GrafProcs record holds pointers to all the standard drawing functions. A programmer may change the pointers in this record and cause QuickDraw II to use a different drawing routine.

An entire drawing environment is kept in a single record (called the GrafPort), which can be saved and restored with a single call. This allows for simple context switching. The programmer has two ways of changing the drawing environment. First, he or she can change the contents of the GrafPort directly and have these changes apply to the drawing environment without making any other calls. Or, he or she can use some of the many calls to set the individual fields in the GrafPort.

Data Structures

Pointer

P	4 bytes

Point

V	2 bytes
H	2 bytes

Rect

V1	2 bytes
H1	2 bytes
V2	2 bytes
H2	2 bytes

String

Standard ProDOS string starting with a length byte followed by up to 255 characters of data.

An_SCB_Byte

Bits	Meaning
0-3	Color Table
4	Reserved
5	Fill 0=off 1=on
6	Interrupt 0 = off 1 = on
7	Color Mode 0=320 1=640

LocInfo
MasterSCB : an_scb_byte
reserved : byte
PointerToPixelImage : pointer
Width : word
BoundsRect : rect

nibble = 0..15
twobit = 0..3

Pattern
case mode of
mode320:
(packed array [0..63] of nibble);
mode640:
(packed array [0..63] of twobit);

PenState
PnLoc : point
PnSize : point
PnMode : integer
PnPat : pattern

GrafPort
PortInfo : LocInfo
PortRect : rect
BkPat : Pattern
PnLoc : Point
PnSize : Point
PnMode : integer
PnPat : pattern
PnVis : integer
FontPtr : Pointer
Txface : Style
TxMode : integer
TxSize : integer
SpExtra : integer
FGColor integer
BGColor : integer

PicSave : pointer
RgnSave : pointer
PolySave : pointer
GrafProcs : pointer

Hardware and the Drawing Environment

The Super Hi-Res Graphics hardware can display 200 scan lines and many colors. The following four features are controlled independently for each scan line:

Color Table	One of 16
Fill Mode	On or Off
Interrupt	On or Off
Color Mode	320 vs 640 pixels per scan line

The scan line control byte (SCB) controls these four features for each scan line. The low nibble of the SCB identifies the color table to be used for this scan line. Bit 4 is reserved. Bit 5 of the SCB controls fill mode: 1 is on, 0 is off. Bit 6 of the SCB controls interrupts: if the bit is set then an interrupt will be generated when the scan line is refreshed. Bit 7 of the SCB controls the mode: 0 is 320, 1 is 640.

```
 7   6   5   4    3   2   1   0
|---|---|---|---||-----------------
 M   I   F   R      Color Table
```

Color Table

A color table is a table of 16 2-byte entries. The low nibble of the low byte is the intensity of the color blue. The high nibble of the low byte is the intensity of the color green. The low nibble of the high byte is the intensity of the color red. The high nibble of the high byte is not used. Pixels in 320 mode are 4 bits wide and their numeric representation identifies a color in the color table. Pixels in 640 mode are 2 bits wide and their numeric representation identifies a color in a subset of the full color table. The first pixel in the byte (bits 0 and 1) selects one of four colors in the table from 0 through 3. The second pixel in the byte (bits 2 and 3) selects one of four colors in the table from 4 through 7. The third pixel in the byte (bits 4 and 5) selects one of four colors in the table from 8 through 11. The fourth pixel in the byte (bits 6 and 7) selects one of four colors in the table from 12 through 15.

HighByte		LowByte	
High Nibble	Low Nibble	High Nibble	Low Nibble
Reserved	Red	Green	Blue

Fill Mode

When fill mode is active, the zeroth color in the color table becomes inactive. A pixel with a numeric value of zero serves as a place holder, indicating that the pixel should be displayed as the same color last displayed.

Scan Line Values

1 0 0 0 0 2 0 0 0 0 0 1 0 0 0 0

Colors Shown

B B B B B W W W W W W B B B B B

Interrupts

Interrupts can be used to synchronize drawing with vertical blanking so pixels are not changed as they are being drawn (a pixel is drawn once every 1/60 of a second). Interrupts can also be used to change the color table before a screen is completely drawn. This will allow a program to show more than 256 colors on the screen at once (but at the cost of servicing the interrupt).

Housekeeping Functions

QDBootInit

Initializes QuickDraw II at boot time. The function puts the address of the cursor update routine into the bank E1 vectors.

QDApInit

Initializes Quickdraw II, sets the current port to the standard port, and clears the screen.

input	*ZeroPageLoc*	WORD
input	*MasterSCB*	WORD
input	*MaxWidth*	WORD
input	*ProgramID*	WORD

The *MasterSCB* is used to set all SCB's in the super hi-res graphics screen. *MaxWidth* is a number that tells QuickDraw II the size in bytes of the largest pixel map that will be drawn to. This allows QuickDraw II to allocate certain buffers it needs only once and keep them throughout the life of the application. *ProgramID* is the ID QuickDraw II will use when getting memory from the Memory Manager. All memory is reserved in the name of this ID.

QDQuit — Frees up any buffers that were allocated.

QDVersion — Returns the version of QuickDraw II.

output	*VersionInfo*	WORD

Global Environment Calls

GetStandardSCB — Returns a copy of the standard SCB in the low byte of the word.

output	*TheStandardSCB*	WORD

This corresponds to:

Bits	Meaning
0-3	Color Table 0
4	Reserved
5	Fill off
6	Interrupt off
7	Color Mode = 320

SetMasterSCB — Sets the master SCB to the specified value (only the low byte is used).

input	*AnSCB*	WORD

The master SCB is the global mode byte used throughout QuickDraw II. The master SCB is used by routines like InitPort to decide what standard values should be put into the GrafPort.

GetMasterSCB — Returns a copy of the master SCB (only the low byte is valid).

output	*AnSCB*	WORD

InitColorTable — Returns a copy of the standard color table for the current mode.

input	*TablePtr*	POINTER

The entries are as follows for 320 mode:

Pixel Value	Name	Master Color	
0	Black	0 0 0	Opposite of White
1	Red	F 0 0	
2	Green	0 F 0	
3	Blue	0 0 F	
4	Teal	0 8 8	
5	??	8 0 8	

6	Brown	0 6 6	
7	Dark Gray	5 5 5	
8	Light Gray	A A A	
9	Orange	F 8 0	
10	???	8 F 8	
11	???	F 8 8	
12	Yellow	F F 0	
13	Magenta	F 0 F	
14	Cyan	0 F F	
15	White	F F F	Opposite of Black

The entries are as follows for 640 mode:

Pixel Value	Name	Master Color	
0	Black	0 0 0	Opposite of White
1	Red	F 0 0	
2	Green	0 F 0	
3	Blue	F F F	

SetColorTable

Sets a color table to specified values.

input	*TableNumber*	WORD
input	*TablePtr*	POINTER

Tablenumber identifies the table to be set to the values specified in the table pointed to. The 16 color tables are stored starting at $9E00. Each table takes $20 bytes. Each word in the table represents one of 4096 colors. The high nibble of the high byte is ignored.

GetColorTable

Fills a color table with the contents of another color table.

input	*TableNumber*	WORD
input	*TablePtr*	POINTER

Tablenumber specifies the number of the color table whose contents are to be copied; *TablePtr* points to the color table which is to receive the contents.

SetColorEntry

Sets the value of a color in a specified color table.

input	*TableNumber*	WORD
input	*EntryNumber*	WORD
input	*Value*	WORD

Tablenumber specifies the number of the color table; *EntryNumber* specifies the number of the color to be changed; *Value* sets the color.

GetColorEntry

Returns the value of a color in a specified color table .

input	*TableNumber*	WORD
input	*EntryNumber*	WORD
output	*Value*	WORD

Tablenumber specifies the number of the color table; *EntryNumber* specifies the number of the color to be examined; *Value* returns the color.

SetSCB

Sets the scan line control byte (SCB) to a specified value.

input	*ScanLine*	WORD
input	*Value*	WORD

Scanline identifies the scan line whose SCB is to be set; *Value* sets the SCB.

GetSCB

Returns the value of a specified scan line control byte (SCB).

input	*ScanLine*	WORD
output	*Value*	WORD

Scanline identifies the scan line whose SCB is to be examined; *Value* returns the value of the SCB.

SetAllSCBs

Sets all scan line control bytes (SCBs) to a specified value.

input	*Value*	WORD

GrafPort Calls

OpenPort

Initializes specified memory locations as a standard port and allocates new VisRgn and ClipRgn.

input	*PortPtr*	LONG

InitPort

Initializes specified memory locations as a standard port.

input	*PortPtr*	LONG

InitPort, unlike **OpenPort**, assumes that the region handles are valid and does not allocate new handles. Otherwise, **InitPort** performs the same functions.

ClosePort — Deallocates the memory associated with a port.

input	*PortPtr*	LONG

All handles are discarded. If the application disposes of the memory containing the port without first calling **ClosePort**, the memory associated with the handles is lost and cannot be claimed.

SetPort — Makes the specified port the current port.

input	*PortPtr*	LONG

GetPort — Returns the handle to the current port.

output	*PortPtr*	LONG

SetPortInfo — Sets the current port's map information structure to the specified location information.

input	*LocInfo*	LONG

SetPortSize — Changes the size of the current GrafPort's PortRect.

input	*Width*	WORD
input	*Height*	WORD

This does not affect the pixel map, but just changes the active area of the GrafPort. The call is normally used by the Window Manager.

MovePortTo — Changes the location of the current GrafPort's PortRect.

input	*Width*	WORD
input	*Height*	WORD

This does not affect the pixel map, but just changes the active area of the GrafPort. The call is normally used by the Window Manager.

SetOrigin — Adjusts the contents of PortRect and BoundsRect so that the upper left corner of PortRect is set to the specified point.

input	*H*	WORD
input	*V*	WORD

VisRgn is also affected, but ClipRgn is not. The pen position does not change.

SetClip Sets the clip region to the region passed by using **CopyRgn**.

input	*RgnHandle*	LONG

GetClip Returns a handle to the current clip region.

output	*RgnHandle*	LONG

ClipRect Changes the clip region of the current GrafPort to a rectangle equivalent to a given rectangle.

input	*RectPtr*	LONG

This does not change the region handle, but affects the region itself.

Cursor-Handling Routines

SetCursor Sets the cursor to the image passed in the cursor record.

input	*CursorPtr*	LONG

If the cursor is hidden, it remains hidden and appears in the new form when it becomes visible again. If the cursor is visible, it appears in the new form immediately.

GetCursorAdr Returns a pointer to the current cursor record.

input	*CursorPtr*	LONG

HideCursor Decrements the cursor level. A cursor level of zero indicates the cursor is visible; a cursor level less than zero indicates the cursor is not visible.

ShowCursor Increments the cursor level unless it is already zero. A cursor level of zero indicates the cursor is visible; a cursor level less than zero indicates the cursor is not visible.

ObscureCursor Hides the cursor until the mouse moves. This tool is used to get the cursor out of the way of typing.

Pen, Pattern, and Drawing Mode Calls

HidePen Decrements the pen level. A pen level of zero indicates drawing will occur; a pen level less than zero indicates drawing will not occur.

ShowPen Increments the pen level unless it is already zero. A pen level of zero indicates that drawing will occur; a pen level less than zero indicates drawing will not occur.

GetPen Returns the pen location.

output	*PointPr*	LONG

SetPenState Sets the pen state in the GrafPort to the values passed.

input	*PenStatePtr*	LONG

GetPenState Returns the pen state from the GrafPort.

output	*PenStatePtr*	LONG

PenSize Sets the current pen size to the specified pen size.

input	*Width*	LONG
input	*Height*	LONG

PenMode Sets the current pen mode to the specified pen mode.

input	*PenMode*	LONG

PenPat

Sets the current pen pattern to the specified pen pattern.

input	*PatternPtr*	LONG

BackPat

Sets the background pattern to the specified pattern.

input	*PatternPtr*	LONG

PenNormal

Sets the pen state to the standard state (**PenSize** = 1,1; **PenMode** = copy; **PenPat** = Black). The pen location is not changed.

MoveTo

Moves the current pen location to the specified point.

input	*H*	WORD
input	*V*	WORD

Move

Moves the current pen location by the specified horizontal and vertical displacements.

input	*dh*	WORD
input	*dv*	WORD

LineTo

Draws a line from the current pen location to the specified point.

Line

Draws a line from the current pen location to a new point specified by the horizontal and vertical displacements.

input	*dh*	WORD
input	*dv*	WORD

Calculations With Rectangles

SetRect

Sets the rectangle pointed to by RectPtr to the specified values.

input	*RectPtr*	LONG
input	*Left*	WORD
input	*Top*	WORD
input	*Right*	WORD
input	*Bottom*	WORD

OffsetRect

Offsets the rectangle pointed to by RectPtr by the specified displacements.

input	*RectPtr*	LONG
input	*dh*	WORD
input	*dv*	WORD

dv is added to the top and bottom; *dh* is added to the left and right.

InsetRect

Insets the rectangle pointed to by RectPtr by the specified displacements.

input	*RectPtr*	LONG
input	*dh*	WORD
input	*dv*	WORD

dv is added to the top and subtracted from the bottom; *dh* is added to the left and subtracted from the right.

SectRect

Calculates the intersection of two rectangles and places the intersection in a third rectangle.

input	*SrcRectAPtr*	LONG
input	*SrcRectBPtr*	LONG
input	*DestRectPtr*	LONG
output	*Boolean*	WORD

If the result is non-empty, the output is TRUE; if the result is empty, the output is FALSE.

UnionRect

Calculates the union of two rectangles and places the union in a third rectangle.

input	*SrcRectAPtr*	LONG
input	*SrcRectBPtr*	LONG
input	*DestRectPtr*	LONG
output	*Boolean*	WORD

If the result is non-empty, the output is TRUE; if the result is empty, the output is FALSE.

PtInRect

Detects whether a specified point is in a specified rectangle.

input	*PtPtr*	LONG
input	*RectPtr*	LONG
output	*Boolean*	WORD

For example, PtInRect((10,10)),((10,10,20)) is TRUE but PtInRect((20,20)),((10,10,20)) is FALSE.

Pt2Rect Copies one point to the upper left of a specified rectangle and another point to the lower right of the rectangle.

input	*Pt1Ptr*	LONG
input	*Pt2Ptr*	LONG
input	*RectPtr*	LONG

EqualRect Compares two rectangles and returns TRUE or FALSE.

input	*R1Ptr*	LONG
input	*R2Ptr*	LONG
output	*Boolean*	WORD

EmptyRect Returns whether or not a specified rectangle is empty.

input	*RectPtr*	LONG
outpu	*Boolean*	WORD

An empty rectangle has the top greater than or equal to the bottom, or the left greater than or equal to the right.

Rectangle Calls

FrameRect Draws the boundary of the specified rectangle with the current pattern and pen size.

input	*RectPtr*	LONG

Only points entirely within the rectangle are affected.

PaintRect Paints (fills) the interior of the specified rectangle with the current pen pattern.

input	*RectPtr*	LONG

EraseRect Paints (fills) the interior of the specified rectangle with the background pattern.

input	*RectPtr*	LONG

InvertRect Inverts the pixels in the interior of the specified rectangle.

input	*RectPtr*	LONG

FillRect Paints (fills) the interior of the specified rectangle with the specified pattern.

input	*RectPtr*	LONG
input	*Pattern*	LONG

Pixel Transfer Calls

ScrollRect

Shifts the pixels inside the intersection of the specified rectangle, VisRgn, ClipRgn, PortRect, and BoundsRect.

input	*RectPointer*	POINTER
input	*dh*	WORD
input	*dv*	WORD
input	*UpdateRgn*	HANDLE

The pixels are shifted a distance of *dh* horizontally and *dv* vertically. The positive directions are to the right and down. No other pixels are affected. Pixels shifted out of the scroll area are lost. The backgound pattern fills the space created by the scroll. In addition *UpdateRgn* is changed to the area filled with **BackPat**.

Note that this *UpdateRgn* must be an existing region; it is not created by **ScrollRect**.

PaintPixels

Transfers a region of pixels.

input	*PaintParamPtr*	LONG

PaintParamPtr is equal to the following:

PtrToSourceLocInfo	LONG
PtrToDestLocInfo	LONG
PtrToSourceRect	LONG
PtrToDestPoint	LONG
Mode	WORD
MaskHandle (ClipRgn)	LONG

The pixels are transferred without referencing the current GrafPort. The source and destination are described in the input, as is the clipping region.

Calculations With Points

AddPt

Adds two specified points together and leaves the result in the destination point.

input	*SrcPtPtr*	LONG
input	*DestPtPtr*	LONG

SubPt

Subtracts the source point from the destination point and leaves the result in the destination point.

input	*SrcPtPtr*	LONG
input	*DestPtPtr*	LONG

SetPt

Sets a point to specified horizontal and vertical values.

input	*SrcPtPtr*	LONG
input	*h*	WORD
input	*V*	WORD

EqualPt

Returns a boolean result indicating whether two points are equal.

input	*Pt1Ptr*	LONG
input	*Pt2Ptr*	LONG
output	*Boolean*	WORD

LocalToGlobal

Converts a point from local coordinates to global coordinates.

input	*PtPtr*	LONG

Local coordinates are based on the current BoundsRect of the GrafPort. Global coordinates have 0,0 as the upper left corner of the pixel image.

GlobalToLocal

Converts a point from global coordinates to local coordinates.

input	*PtPtr*	LONG

Local coordinates are based on the current BoundsRect of the GrafPort. Global coordinates have 0,0 as the upper left corner of the pixel image.

Calculations With Regions

NewRgn

Allocates space for a new region and initializes it to the empty region. This is the only way to create a new region.

output	*RgnHandle*	LONG

All other calls work with existing regions.

DisposeRgn

Deallocates space for the specified region.

input	*RgnHandle*	LONG

CopyRgn

Copies the contents of a region from one region to another.

input	*SrcRgn*	HANDLE
input	*DestRgn*	HANDLE

If the regions are not the same size to start with, the *DestRgn* is resized. (*DestRgn* must already exist. This call does not allocate it.)

SetEmptyRgn

Destroys the previous region information by setting it to the empty region.

input	*Rgn*	HANDLE

The empty region is a rectangular region with a bounding box of (0,0,0,0). If the original region was not rectangular, the region is resized.

SetRectRgn

Destroys the previous region information by setting it to a rectangle described by the input.

input	*Rgn*	HANDLE
input	*Left*	WORD
input	*Top*	WORD
input	*Right*	WORD
input	*Bottom*	WORD

If the inputs do not describe a valid rectangle, the region is set to the empty region. If the original region was not rectangular, the region is resized.

RectRgn

Destroys the previous region information by setting it to a rectangle described by the input.

input	*RgnHandle*	LONG
input	*RectPtr*	LONG

If the input does not describe a valid rectangle, the region is set to the empty region. If the original region was not rectangular, the region is resized.

OpenRgn

Tells QuickDraw II to allocate temporary space and start saving lines and framed shapes for later processing as a region definition.

While the region is open, all calls to **Line**, **LineTo**, and the procedures that draw framed shapes affect the outline of the region.

CloseRgn

Tells QuickDraw II to stop processing information and to return the region that has been created.

input	*DestRgn*	HANDLE

DestRgn must already exist, and its contents are replaced with the new region.

OffsetRgn

Moves the region on the coordinate plane a distance of *dh* horizontally and *dv* vertically.

input	*Rgn*	HANDLE
input	*dh*	WORD
input	*dv*	WORD

The region retains its size and shape.

InsetRgn

Shrinks or expands a region.

input	*RgnHandle*	LONG
input	*dh*	WORD
input	*dv*	WORD

All points on the region boundary are moved inwards a distance of *dv* vertically and *dh* horizontally. If *dv* or *dh* are negative, the points are moved outwards in that direction. **InsetRgn** leaves the region "centered" on the same position, but moves the outline. **InsetRgn** of a rectangular region works just like **InsetRect**.

SectRgn

Calculates the intersection of two regions and places the intersection in the third region.

input	*SrcRgnA*	HANDLE
input	*SrcRgnB*	HANDLE
input	*DestRgn*	HANDLE

The function does not allocate the third region. You must allocate the third region before the call to **SectRgn**.

If the regions do not intersect, or one of the regions is empty, the destination is set to the empty region.

UnionRgn

Calculates the union of two regions and places the union in the third region.

input	*SrcRgnA*	HANDLE
input	*SrcRgnB*	HANDLE
input	*DestRgn*	HANDLE

The function does not allocate the third region. You must allocate the third region before the call to **UnionRgn**.

If both regions are empty, the destination is set to the empty region.

DiffRgn

Calculates the difference of two regions and places the difference in the third region.

input	*SrcRgnA*	HANDLE
input	*SrcRgnB*	HANDLE
input	*DestRgn*	HANDLE

The function does not allocate the third region. You must allocate the third region before the call to **DiffRgn**.

If the source region is empty, the destination is set to the empty region.

XorRgn

Calculates the difference between the union and the intersection of two regions and places the result in the third region.

input	*SrcRgnA*	HANDLE
input	*SrcRgnB*	HANDLE
input	*DestRgn*	HANDLE

The function does not allocate the third region. You must allocate the third region before the call to **XorRgn**.

If the regions are not coincident, the destination is set to the empty region.

PtInRgn

Checks to see whether the pixel below and to the right of the point is within the specified region.

input	*PointPtr*	POINTER
input	*RgnHandle*	HANDLE
output	*Boolean*	WORD

The function returns TRUE if the pixel is within the region and FALSE if it is not.

RectInRgn

Checks whether a given rectangle intersects a specified region.

input	*RectPtr*	POINTER
input	*RgnHandle*	HANDLE
output	*Boolean*	WORD

The function returns TRUE if the intersection encloses at least one pixel or FALSE if it does not.

EqualRgn

Compares the two regions and returns TRUE if they are equal or FALSE if not.

input	*Rgn1*	HANDLE
input	*Rgn2*	HANDLE
output	*Boolean*	WORD

The two regions must have identical sizes, shapes and locations to be considered equal. Any two empty regions are always equal.

EmptyRgn

Checks to see if a region is empty.

input	*RgnHandle*	LONG
output	*Boolean*	WORD

Returns TRUE if the region is empty or FALSE if not.

Graphic Operations on Region Calls

FrameRgn

Draws the boundary of the specified region with the current pattern and current pen size.

input	*RgnHandle*	LONG

Only points entirely inside the region are affected.

If a region is open and being formed, the outside outline of the region being framed is added to that region's boundary.

PaintRgn

Paints (fills) the interior of the specified region with the current pen pattern.

input	*RgnHandle*	LONG

EraseRgn

Fills the interior of the specified region with the background pattern.

input	*RgnHandle*	LONG

InvertRgn

Inverts the pixels in the interior of the specified region.

input	*RgnHandle*	LONG

FillRgn

Fills the interior of the specified region with the specfied pattern.

input	*RgnHandle*	LONG
input	*PatternPtr*	LONG

Miscellaneous Utilities

Random

Returns a pseudorandom number in the range –32768 to 32767.

output	*Integer*	WORD

The number returned is generated based upon calculations performed on *SeedValue*, which can be set with **SetRandSeed**. The result for any particular seed value is always the same.

SetRandSeed

Sets the seed value for the random number generator.

input	*SeedValue*	WORD

GetPixel

Returns the pixel below and to the right of the specifed point.

input	*h*	WORD
input	*v*	WORD
input	*ThePixel*	WORD

ThePixel is returned in the lower bits of the word. If the current drawing location has a chunkiness of 2, then 2 bits of the word are valid. If the current drawing location has a chunkiness of 4, then 4 bits of the word are valid.

There is no guarantee that the point actually belongs to the port.

Chapter 4

Memory Manager

Overview

The Memory Manager on the Cortland is responsible for allocating blocks of memory to programs. The Manager does the bookkeeping of what memory is being used and keeps track of who owns various blocks of memory.

Properties of Memory Blocks

Memory blocks have attributes that determine how they are allocated and maintained. Some attributes are defined at allocation time and can't be changed. Other attributes can be modified after allocation.

Allocation Attributes

When a block is allocated, an attribute byte is specified that determines how the block is allocated. This type of attribute can only be set when the block is allocated. The attributes are as follows:

- Movable
- Fixed Address
- Fixed Bank
- Bank Boundary Limited
- Special Memory Useable
- Page Aligned

These attributes are explained in this section.

Movable

If a block is movable, it can be moved when compacting memory. Code blocks will rarely be movable but data blocks should usually be movable.

Fixed Address

This attribute specifies that the block must be at a specified address when allocated. An example is allocating the graphics screen.

Fixed Bank

This attribute specifies that the block must start in a specified bank. An example is allocating a block to be used as a zero page.

Bank Boundary Limited

This attribute specifies that a block must not cross banks. Code blocks, for example, my never cross banks.

Special Memory Useable

This attribute specifies that the block may be allocated in special memory. This is memory that was used in the Apple IIe. It includes banks 0 and 1 and the video screens.

Page Aligned

For timing reasons, code or data may need to be page aligned.

Modifiable Attributes

The memory manager can move or purge a block while making room for a new block. There are attributes that determine whether a block can be moved or purged; these attributes can be moved or purged. The attributes are as follows:

- Locked
- PurgeLevel

Locked

When a block is locked, it is unmovable and unpurgeable regardless of the setting of Moveable or PurgeLevel. This concept allows a block to be temporarily locked down while it is being executed or referenced.

PurgeLevel

PurgeLevel is a two-bit number defining the purge priority of a block. *0* means the block cannot be purged; *3* means the block will be the first purged.

Housekeeping Functions

MMInit Called at boot time.

MMStartUp Initializes the Memory Manager.

MMShutDown Releases resources.

MMVersion Returns the version of the Memory Manager.

output	*VersionInfo*	WORD

Memory Allocating Functions

NewHandle Creates a new block.

input	*BlockSize*
input	*Owner*
input	*Attributes*
input	*Location*
output	*Handle*

Blocksize is the size of the block to create.

ReAllocHandle Reallocates a block that was purged.

input	*TheHandle*
input	*BlockSize*
input	*Owner*
input	*Attributes*
input	*Location*
output	*Handle*

Blocksize is the size of the block to create.

Memory Freeing Functions

DisposHandle Purges a specified unlocked block and deallocates the handle.

input	*TheHandle*	HANDLE

The block must be unlocked, but is purged regardless of its purge level.

DisposAll Discards all of the handles for a specific owner.

input	*Owner*	HANDLE

PurgeHandle

Purges a specified unlocked block, but does not deallocate the handle.

input	*TheHandle*	HANDLE

The block must be unlocked, but is purged regardless of its purge level. *TheHandle* itself remains allocated but is pointer to NIL.

PurgeAll

Purges all of the purgeable blocks for a specific owner.

input	*Owner*	HANDLE

Block Information Functions

GetHandleSize

Returns the size of a block.

input	*TheHandle*	HANDLE
output	*Size*	LONG

SetHandleSize

Changes the size of a block.

input	*TheHandle*	HANDLE
input	*NewSize*	LONG

The block can be made larger or smaller. If more room is needed to lengthen a block, memory may be compacted or blocks may be purged.

FindHandle

Returns the handle of the block containing a specified address.

input	*Location*	HANDLE
output	*TheHandle*	LONG

Note that, if the block is not locked, it may move.

Locking and Unlocking Functions

HLock

Locks a block specified by a handle.

input	*TheHandle*	HANDLE

A locked block cannot be relocated during memory compaction..

HLockAll — Locks all of the blocks owned by an owner.

input	*Owner*	

HUnLock — Unlocks a block specified by a handle.

input	*TheHandle*	HANDLE

A unlocked block can be relocated during memory compaction..

HUnLockAll — Unlocks all of the blocks owned by an owner.

input	*Owner*	

Purge Level Functions

SetPurge — Sets the purge level of a block specified by a handle.

input	*TheHandle*	HANDLE
input	*NewPLevel*	

SetPurgeAll — Sets the purge level of all blocks owned by a specified owner.

input	*Owner*	HANDLE
input	*NewPLevel*	

Free Space Functions

FreeMem — Returns the total number of free bytes in memory.

output	*Size*	LONG

FreeMem compacts memory space. The function does not count memory that can be freed by purging; it might not be possible (because of memory fragmentation) to allocate a block that large.

MaxBlock — Returns the size of the largest free block in memory.

output	*Size*	LONG

This function does not count memory that can be freed by purging or compacting.

Chapter 5

Event Manager

Overview

The Event Manager allows applications to monitor the user's actions, such as those involving the mouse, keyboard, and keypad. The Event Manager is also used by other parts of the Toolbox; for instance, the Window Manager uses events to coordinate the ordering and display of windows on the screen. There are actually two Event Managers: one in the Operating System and one in the Toolbox.

The Operating System Event Manager detects low-level, hardware-related events such as mouse button presses and keystrokes. It stores information about these events in the event queue and provides routines that access the queue.

The Operating System Event Manager also allows an application to

- post its own events into the event queue
- remove events from the event queue
- set the system event mask, to control which types of events get posted into the queue

The Toolbox Event Manager calls the Operating System Event Manager to retrieve events from the event queue. In addition, it reports window and switch events, which aren't kept in the queue. The Toolbox Event Manager is the application's link to its user. A typical event-driven application decides what to do from moment to moment by asking the Toolbox Event Manager for events and responding to them one by one in whatever way is appropriate.

The Toolbox Event Manager also allows an application to

- restrict some of the routines to apply only to certain event types
- directly read the current state of the mouse button
- monitor the location of the mouse

In general, events are collected from a variety of sources and reported to the application on demand, one at a time. Events aren't necessarily reported in the order they occurred because some have a higher priority than others.

> *Note:* In the remainder of this document, *OSEM* denotes the Operating System Event Manager and *TBEM* denotes the Toolbox Event Manager.

Event Types

Events are of various types. Some report actions by the user; others are generated by the Window Manager, the Control Manager, device drivers, or the application itself for its own purposes. Some events are handled by the system before the application ever sees them; others are left for the application to handle. The event types are as follows:

Mouse Events

Pressing the mouse button generates a **mouse-down event**; releasing the button generates a **mouse-up event**. Movements of the mouse cause the cursor position to be updated but are not reported as events. Whenever an event is posted, the location of the mouse at that time is reported in a field of the event record. The application can obtain the current mouse position if needed by calling the TBEM routine GetMouse. Because relative pointing devices such as joysticks must also be supported, the Event Manager differentiates between button 0 and button 1.

Keyboard Events

The character keys on the keyboard and keypad generate **key-down events** when pressed; this includes all keys except Shift, Caps Lock, Control, Option, and Open-Apple, which are called modifier keys. Modifier keys are treated differently and generate no keyboard events of their own. Whenever an event is posted, the state of the modifier keys is reported in a field of the event record.

The character keys on the keyboard and keypad also generate **auto-key events** when held down. Two different time intervals are associated with auto-key events. The first auto-key event is generated after a certain initial delay has elapsed since the key was originally pressed; this is called the delay to repeat. Subsequent auto-key events are then generated each time a certain repeat interval has elapsed since the last such event; this is called the repeat speed. The user can change these values with the Control Panel.

Window Events

The Window Manager generates events to coordinate the display of windows on the screen. **Activate events** are generated whenever an inactive window becomes active or an active window becomes inactive. They generally occur in pairs (that is, one window is deactivated and then another is activated).

Update events occur when all or part of a window's contents need to be drawn or redrawn, usually as a result of the user opening, closing, activating, or moving a window.

Other Events

A **device driver event** may be generated by device drivers in certain situations; for example, a driver might be set up to report an event when its transmission of data is interrupted. Device driver events are placed in the event queue with the OSEM procedure PostEvent.

An application can define as many as four **application events** of its own and use them for any desired purpose. Application-defined events are placed in the event queue with the OSEM procedure PostEvent.

A **switch event** is generated by the Control Manager whenever a button-down event has occurred on the switch control.

A **desk accessory event** is generated whenever the user enters the special keystoke to invoke a "classic" desk accessory (currently CONTROL-OPEN APPLE-ESCAPE).

A **null event** is returned by the Event Manager if it has no other events to report.

Event Priority

Events are retrieved from the event queue in the order they were originally posted. However, the way that various types of events are generated and detected causes some events to have higher priority than others. Also, not all events are kept in the event queue. Furthermore, when an application asks the TBEM for an event, it can specify particular types that are of interest. Specifying such events can cause some events to be passed over in favor of others that were actually posted later.

The TBEM always returns the highest-priority event available of the requested types. The priority ranking is as follows:

1. Activate (window becoming inactive before window becoming active).
2. Switch.
3. Mouse-down, mouse-up, key-down, auto-key, device driver, application-defined, desk accessory (all in FIFO order).
4. Update (in front-to-back order of windows).

Activate events take priority over all others; they're detected in a special way, and are never actually placed in the event queue. The TBEM checks for pending activate events before looking in the event queue, so it will always return such an event if one is available. Because of the special way activate events are detected, there can never be more than two such events pending at the same time; at most there will be one for a window becoming inactive followed by another for a window becoming active.

Next in priority are switch events, which are generated by the Control Manager and are also not placed in the event queue. If no activate events are pending, the TBEM checks for a switch event before looking in the event queue. If a switch event is available, the TBEM then checks to see if any update events are pending, and if so, it returns the update event to the application. The switch event is not returned to the application until there are no pending update events. This is to ensure that all of the windows are updated before the application is switched.

Category 3 includes most of the event types. Within this category, events are retrieved from the queue in the order they were posted.

Next in priority are update events. Like activate and switch events, these are not placed in the event queue, but are detected in another way. If no higher-priority event is available, the TBEM checks for windows whose contents need to be drawn. If it finds one, it returns an update event for that window. Windows are checked in the order in which they're displayed on the screen, from front to back, so if two or more windows need to be updated, an update event will be returned for the frontmost such window.

Finally, if no other event is available, the TBEM returns a null event.

> *Note:* If the queue should become full, the OSEM will begin discarding old events to make room for new ones as they're posted. The events discarded are always the oldest ones in the queue.

Event Records

Every event is represented internally by an event record containing all pertinent information about that event. The event record includes the following information:

- the type of event
- the time the event was posted (in ticks since system startup)
- the location of the mouse at the time the event was posted (in global coordinates)
- the state of the mouse buttons and modifier keys at the time the event was posted
- any additional information required for a particular type of event, such as which key the user pressed or which window is being activated

Every event, including null events, has an event record containing this information.

Event records are defined as follows:

what	INTEGER	{event code}
message	LONGINT	{event message}
when	LONGINT	{ticks since startup}
where	Point	{mouse location}
modifiers	INTEGER	{modifier flags}

The when field contains the number of ticks since the system last started up, and the where field gives the location of the mouse, in global coordinates, at the time the event was posted. The other three fields are described in the following sections.

Event Code

The what field of an event record contains an event code identifying the type of the event. The event codes are assigned as follows:

0 - null event
1 - mouse down event
2 - mouse up event
3 - key down event
4 - undefined
5 - auto-key event
6 - update event
7 - undefined
8 - activate event
9 - switch event
10 - desk accessory event
11 - device driver event
12 - application-defined event
13 - application-defined event
14 - application-defined event
15 - application-defined event

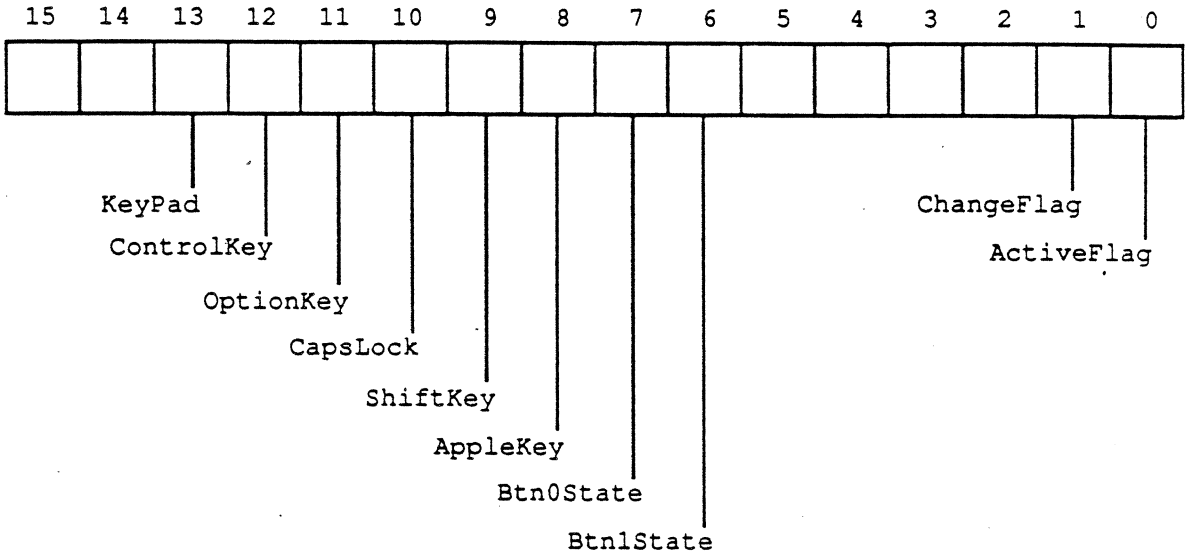
Event Message

The message field of an event record contains the event message, which conveys additional information about the event. The nature of this information depends on the event type, as shown in the following table.

Event type	Event message
Key-down	ASCII character code in low-order byte
Auto-key	ASCII character code in low-order byte
Activate	Pointer to window
Update	Pointer to window
Mouse-down	Button number (0 or 1) in low-order word
Mouse-up	Button number (0 or 1) in low-order word
Device driver	Defined by the device driver
Application	Defined by the application
Switch	Undefined
Desk Accessory	Undefined
Null	Undefined

Modifier Flags

The modifiers field of an event record contains further information about activate events and the state of the modifier keys and mouse buttons at the time the event was posted, as shown below. The application might look at this field to find out, for instance, whether the OPEN-APPLE key was down when a mouse-down event was posted (which could affect the way objects are selected) or when a key-down event was posted (which could mean the user is choosing a menu item by typing its keyboard equivalent).

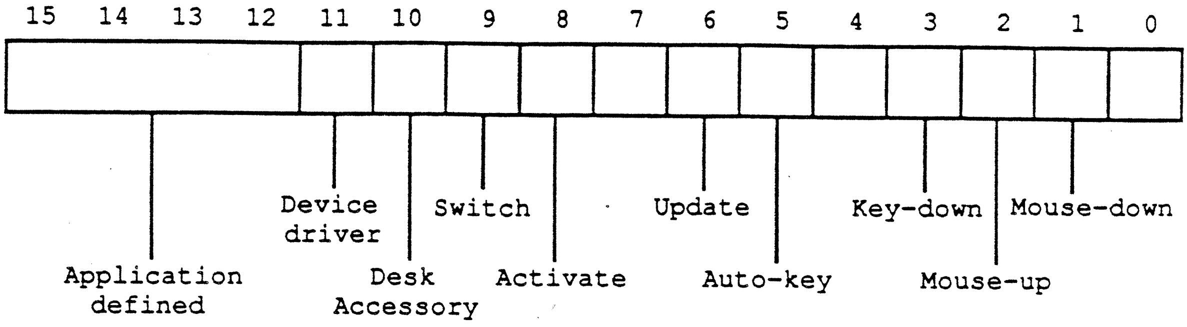


The ActiveFlag and ChangeFlag bits give further information about activate events. The ActiveFlag bit is set to 1 if the window pointed to by the event message is being activated, or 0 if the window is being deactivated. The ChangeFlag bit is set to 1 if the active window is changing from an application window to a system window or vice versa. Otherwise, it's set to 0. The KeyPad bit gives further information about key-down events; it's set to 1 if the key pressed was on the keypad, or 0 if the key pressed was on the keyboard. The remaining bits indicate the state of the mouse button and modifier keys. Note that the Btn0State and Btn1State bits are set to 1 if the corresponding mouse button is *up*, whereas the bits for the five modifier keys are set to 1 if their corresponding keys are *down*.

Event Masks

Some of the TBEM and OSEM routines can be restricted to operate on a specific event type or group of types; in other words, the specified event types are enabled while all others are disabled. For instance, instead of just requesting the next available event, the application can specifically ask for the next keyboard event.

An application can specify which event types a particular call applies to by supplying an **event mask** as a parameter. This is an integer in which there's one bit position for each event type, as shown below. The bit position representing a given type corresponds to the event code for that type—for example, update events (event code 6) are specified by bit 6 of the mask. A 1 in bit 6 means that this call applies to update events; a 0 means that it doesn't.



Note: Null events can't be disabled; a null event will always be reported when none of the enabled types of events are available.

There's also a global **system event mask** that controls which event types get posted into the event queue by the OSEM. Only event types corresponding to bits set in the system event mask are posted; all others are ignored. When the system starts up, the system event mask is set to post all events.

Using the Event Managers

If an application will be using the Event Managers and the Window Manager, it must initialize the Event Managers **before** initializing the Window Manager. The TBEM and OSEM are initialized by calling the TBEM routine **EMStartUp**. Because the TBEM needs to share data with the Window Manager, they must both use the same zero-page work area. When the Window Manager is initialized, it must call the TBEM routine **DoWindows** to obtain the address of the zero-page work area that has been assigned to the Event Managers. If **DoWindows** is not called, the TBEM will assume that windows are not being used and will not attempt to return window events.

Event-driven applications have a main loop that repeatedly calls **GetNextEvent** to retrieve the next available event, and then takes whatever action is appropriate for each type of event. Some typical responses to commonly occurring events are described in the next section. The program is expected to respond only to those events that are directly related to its own operations. After calling **GetNextEvent**, it should test the Boolean result to find out whether it needs to respond to the event: TRUE means the event may be of interest to the application; FALSE usually means it will not be of interest.

In some cases, the application may simply want to look at a pending event while leaving it available for subsequent retrieval by **GetNextEvent**. It can do this with the **EventAvail** function.

Responding to Mouse Events

On receiving a mouse-down event, an application should first call the Window Manager to find out where on the screen the mouse button was pressed, and then respond in whatever way is appropriate. Depending on the part of the screen in which the button was pressed, the application may have to call Toolbox routines in the Menu Manager, the Desk Manager, the Window Manager, or the Control Manager.

If the application attaches some special significance to pressing a modifier key along with the mouse button, it can discover the state of that modifier key when the mouse button was down by examining the appropriate flag in the modifiers field of the event record.

If the application wishes to respond to mouse double-clicks, it will have to detect them itself. It can do so by comparing the time and location of a mouse-up event with those of the immediately following mouse-down event. It should assume a double-click has occurred if both of the following are true:

- The times of the mouse-up event and the mouse-down event differ by a number of ticks less than or equal to the value returned by the TBEM function GetDblTime.
- The locations of the two mouse-down events separated by the mouse-up event are sufficiently close to each other. Exactly what this means depends on the particular application. For instance, in a word-processing application, two locations might be considered essentially the same if they fall on the same character, whereas in a graphics application they might be considered essentially the same if the sum of the horizontal and vertical changes in position is no more than five pixels.

Mouse-up events may be significant in other ways; for example, they might signal the end of dragging to select more than one object. Most simple applications, however, will ignore mouse-up events.

Responding to Keyboard Events

For a key-down event, the application should first check the modifiers field to see whether the character was typed with the Open-Apple key held down; if so, the user may have been choosing a menu item by typing its keyboard equivalent.

If the key-down event was not a menu command, the application should then respond to the event in whatever way is appropriate. For example, if one of the windows is active, it might want to insert the typed character into the active document; if none of the windows is active, it might want to ignore the event.

Usually the application can handle auto-key events the same way as key-down events. You may, however, want it to ignore auto-key events that invoke commands that shouldn't be continually repeated.

Responding to Window Events

When the application receives an activate event for one of its own windows, the Window Manager will already have done all of the normal "housekeeping" associated with the event, such as highlighting or unhighlighting the window. The application can then take any further action that it may require, such as showing or hiding a scroll bar or highlighting or unhighlighting a selection.

On receiving an update event for one of its own windows, the application should usually update the contents of the window.

Responding to Other Events

An application will never receive a desk accessory event because these are intercepted and handled by the Desk Manager.

If the application receives a switch event, it should call a (currently unnamed) routine in the Switcher that will save the current state and switch to the next application.

Posting and Removing Events

If an application is using application-defined events, it will need to call the OSEM function PostEvent to post them into the event queue. Device drivers can post events the same way. This function is sometimes also useful for reposting events that have been removed from the event queue with **GetNextEvent**.

In some situations, you may want your application to remove from the event queue some or all events of a certain type or types. It can do this with the OSEM procedure **FlushEvents**.

Other Operations

In addition to receiving the user's mouse and keyboard actions in the form of events, applications can directly read the mouse location and state of the mouse buttons by calling the TBEM routines **GetMouse** and **Button**, respectively. To follow the mouse when the user moves it with the button down, the application can use the TBEM routines **StillDown** or **WaitMouseUp**.

Finally, the TBEM function **GetCaretTime** returns the number of ticks between blinks of the "caret" (usually a vertical bar) marking the insertion point in editable text. An application should call **GetCaretTime** if it is causing the caret to blink itself. The application would check this value each time through the main event loop to ensure a constant frequency of blinking.

Applications should never call the TBEM routines **DoWindows** and **SetSwitch**, and will probably never call the OSEM routines **GetOSEvent**, **OSEventAvail**, **SetEventMask**, and **GetEvQHdr**.

The Journaling Mechanism

The Event Manager has a journaling mechanism that can be accessed through assembly language. The journaling mechanism "decouples" the Event Manager from the user and feeds it events from a file that contains a recording of all the events that occurred during some portion of a user's session. Specifically, this file is a recording of all calls to the TBEM routines **GetNextEvent**, **EventAvail**, **GetMouse**, and **Button**. When a journal is being recorded, every call to any of these routines is sent to a journaling device driver, which records the call (and the results of the call) in a file. When the journal is played back, these recorded TBEM calls are taken from the journal file and sent directly to the TBEM. The result is that the recorded sequence of user-generated events is reproduced when the journal is played back.

Note: The journaling mechanism may not be supported in the first release due to time contraints.

Housekeeping Functions

EMBootInit

Called at boot time. Does nothing.

EMStartUp

Initializes the Toolbox and Operating System Event Managers.

input	*ZeroPageAdrs*	INTEGER
input	*QueueSize*	INTEGER
input	*XMinClamp*	INTEGER
input	*XMaxClamp*	INTEGER
input	*YMinClamp*	INTEGER
input	*YMaxClamp*	INTEGER

QueueSize specifies the maximum number of event records the queue can hold. If *QueueSize* is equal to zero, a default size of ?? will be used. The Clamp inputs specify the minimum and maximum X and Y clamps for the mouse.

EMShutDown

Turns off the Toolbox and Operating System Event Managers.

EMVersion

Returns the version of the Toolbox and Operating System Event Managers.

output	*VersionInfo*	WORD

DoWindows

Returns the address of the zero-page work area used by the Toolbox and Operating System Event Managers.

output	*ZeroPageAdrs*	INTEGER

Accessing Events Through the Toolbox Event Manager

GetNextEvent

Returns the next available event of a specified type or types and, if the event is in the event queue, removes it from the queue.

input	*EventMask*	INTEGER
input	*EventPtr*	POINTER to EventRecord
output	*Boolean*	WORD

The event is returned in the event record pointed to by *EventPtr*. *EventMask* specifies which event types are of interest.

GetNextEvent returns the next available event of any type designated by the mask, subject to the following priority order:

1. Activate (window becoming inactive before window becoming active).

2. Switch.

3. Mouse-down, mouse-up, key-down, auto-key, device driver, application-defined, desk accessory, all in FIFO order.

4. Update (in front-to-back order of windows).

If no event of any of the designated types is available, **GetNextEvent** returns a null event. This priority order is further discussed in "?????????".

Events in the queue that aren't designated in the mask are left in the queue. The events can be removed by calling the **FlushEvents** tool.

Before reporting an event to the application, **GetNextEvent** first calls the Desk Manager tool **SystemEvent** to see whether the system wants to intercept and respond to the event. If so, or if the event being reported is a null event, **GetNextEvent** returns a *Boolean* result of FALSE; a *Boolean* result of TRUE means that the application should handle the event itself. The Desk Manager intercepts the following events:

- desk accessory events
- activate and update events directed to a desk accessory
- mouse-up and keyboard events, if the currently active window belongs to a desk accessory

In each case, the event is intercepted by the Desk Manager only if the desk accessory can handle that type of event. As a rule, all desk accessories should be set up to handle activate, update, and keyboard events and should not handle mouse-up events.

GetNextEvent also handles the Alarm Clock desk accessory. If the "alarm" is set and the current time is the alarm time, the alarm goes off. The user can set the alarm with the Alarm Clock desk accessory.

EventAvail

This tool works the same way as **GetNextEvent**, except that **EventAvail** leaves the event in the event queue (if the event was there in the first place).

input	*EventMask*	INTEGER
input	*EventPtr*	POINTER to EventRecord
output	*Boolean*	WORD

An event returned by **EventAvail** cannot be accessed if, in the meantime, the queue becomes full and the event is discarded. However, because the oldest events are the ones discarded, useful events will be discarded only in an unusually busy environment.

Reading the Mouse

GetMouse

Returns the current mouse location.

output	*MouseLocPtr*	POINTER to a Point

The location is given in the local coordinate system of the current GrafPort (for example, the currently active window). This differs from the mouse location stored in the "where" field of an event record; that location is always in global coordinates.

Button

Returns the current state of the mouse button.

input	*ButtonNum*	INTEGER
output	*Boolean*	WORD

ButtonNum contains the number (0 or 1) of the mouse button to check. *Boolean* returns TRUE if the mouse button is currently down, or FALSE if it isn't.

StillDown

Tests whether the mouse button is still down.

input	*ButtonNum*	INTEGER
output	*Boolean*	WORD

ButtonNum contains the number (0 or 1) of the mouse button to check. *Boolean* returns TRUE if the mouse button is currently down and there are no more mouse events pending in the event queue. Usually called after a mouse-down event, **StillDown** is a true test of whether the mouse button is still down from the original press. (**Button** is not a true test, because it returns TRUE whenever the mouse button is currently down, even if the button was released and pressed again since the original mouse-down event.)

WaitMouseUp

Tests whether the mouse button is still down, and, if the button is not still down from the original press, removes the preceding mouse-up event before returning FALSE.

input	*ButtonNum*	INTEGER
output	*Boolean*	WORD

ButtonNum contains the number (0 or 1) of the mouse button to check. *Boolean* returns TRUE if the mouse button is currently down and there are no more mouse events pending in the event queue.

WaitMouseUp could be used, for example, if an application attached some special significance to mouse double-clicks and to mouse-up events. **WaitMouseUp** would allow the application to recognize a double-click without being confused by the intervening mouse-up.

Miscellaneous Toolbox Event Manager Routines

GetDblTime

Returns the suggested maximum difference (in ticks) between mouse-up and mouse-down events in order for the mouse clicks to be considered a double click.

output	*MaxTicks*	LONGINT

The user can adjust this value by using the Control Panel.

GetCaretTime

Returns the time (in ticks) between blinks of the "caret" (usually a vertical bar) marking the insertion point in text.

output	*NumTicks*	LONGINT

If an application is not using TextEdit, the application must cause the caret to blink. On every pass through the program's main event loop, the application should check *NumTicks* against the elapsed time since the last blink of the caret .

The user can adjust this value by using the Control Panel.

SetSwitch

Informs the Toolbox Event Manager of a pending switch event. **SetSwitch** is called by the Control Manager and should not be called by an application.

Posting and Removing Events

PostEvent

Places an event in the event queue.

input	*EventCode*	INTEGER
input	*EventMsg*	LONGINT
output	*Result*	INTEGER

EventCode designates the type of event to be placed in the queue. *EventMsg* specifies the event message, with the current state of the modifier keys and mouse buttons supplied in the high-order word of the message. In addition, the current time and mouse location is recorded in the message.

Result returns a result code equal to one of the following values:

0 - no error (event posted)
1 - event type not designated in system event mask

An application must be careful when it posts any events other than its own application-defined events into the queue. Attempting to post an activate or update event (which aren't normally placed in the queue), for example, will interfere with the normal operation of the Toolbox Event Manager.

If **PostEvent** is used to repost an event, the event time, mouse location, state of the modifier keys, and state of the mouse buttons will all be changed from the originally posted event. This can alter the meaning of the event.

FlushEvents

Removes events from the event queue.

input	*EventMask*	INTEGER
input	*StopMask*	INTEGER
output	*Result*	INTEGER

EventMask specifies the type or types of the events to be removed from the queue. **FlushEvents** removes all events of the type or types specified up to but not including the first event of any type specified by *StopMask*. To remove all events specified by *EventMask*, specify 0 as the value of *StopMask*.

If the event queue doesn't contain any event of the types specified by *EventMask*, **FlushEvents** does nothing.

When the tool finishes, *Result* contains 0 if all events were removed from the queue, or an event code specifying the type of event that caused the process to stop.

Accessing Events Through the OS Event Manager

GetOSEvent

Returns the next available event of a specified type or types and, if the event is in the event queue, removes it from the queue.

input	*EventMask*	INTEGER
input	*EventPtr*	POINTER to EventRecord
input	*Boolean*	WORD

The event is returned in the event record pointed to by *EventPtr*. *EventMask* specifies which event types are of interest.

GetOSEvent returns the next available event of any type designated by the mask, subject to the following priority order:

1. Activate (window becoming inactive before window becoming active).

2. Switch.

3. Mouse-down, mouse-up, key-down, auto-key, device driver, application-defined, desk accessory, all in FIFO order.

4. Update (in front-to-back order of windows).

If no event of any of the designated types is available, **GetNextEvent** returns a null event and a *Boolean* of FALSE; otherwise *Boolean* is TRUE. This priority order is further discussed in "?????????".

Events in the queue that aren't designated in the mask are left in the queue. The events can be removed by calling the **FlushEvents** tool.

OSEventAvail

This tool works the same way as **GetOSEvent**, except that **OSEventAvail** leaves the event in the event queue (if the event was there in the first place).

input	*EventMask*	INTEGER
input	*EventPtr*	POINTER to EventRecord
output	*Boolean*	WORD

An event returned by **OSEventAvail** cannot be accessed if, in the meantime, the queue becomes full and the event is discarded. However, because the oldest events are the ones discarded, useful events will be discarded only in an unusually busy environment.

Miscellaneous OS Event Manager Routines

SetEventMask

Sets the system event mask to the specified event mask.

input	*TheMask*	INTEGER

The Operating System Event Manager will post only those event types that correspond to bits set in the mask. It will not post activate, update, or switch events, because those events are not stored in the event queue.

The system event mask is initially set to post all events. An application should not change the system event mask, because desk accessories may depend upon receiving certain types of events.

GetEvQHdr

Returns a pointer to the header of the event queue.

output	*QHdrPtr*	POINTER

Chapter 6

Other ROM Tools

SANE

The ROM Tools for the Cortland will provide all of the functions found in the Standard Apple Numeric Environment (SANE). The SANE Tools can be called using the normal Cortland call mechanism.

The SANE Tools for the Cortland work in the same manner as they do in other Apple environments, except for minor differences in the halt mechanism. For more information regarding that mechanism, refer to the *Cortland SANE Tool Set Preliminary Notes*. For more information regarding the capabilities of SANE, refer to the *Apple Numerics Manual*.

Desk Manager

No information available at this time.

Sound Manager

No information available at this time.

Chapter 7

Miscellaneous ROM Tools

Overview

There are a number of tools that do not fall easily into one logical category. We have grouped them under the name of "Miscellaneous ROM Tools". Those tools are explained in this chapter.

The error codes for the miscellaneous tools are as follows:

$0000	No Error
$0001	Bad Input Parameter
$0002	No Device for Input Parameter

Housekeeping Functions

PowerUpInit Called at boot time. Initializes HeartBeat interrupt chain link pointer to $00000000.

StartUp Does nothing.

ShutDown Does nothing.

Version Returns the version of the miscellaneous tools.

output	*VersionInfo*	WORD

Math Functions

Multiply Multiplies two 16-bit inputs and produces a 32-bit result.

input	*ResultSpace*	LONG
input	*M1*	WORD
input	*M2*	WORD
output	*Result*	LONG

If the inputs were unsigned, the 32-bit *Result* is unsigned. If the inputs were signed, the low word of the 32-bit *Result* indicates the sign.

SDivide

Divides two 16-bit inputs and produces two 16-bit signed results.

input	*ResultSpace*	LONG
input	*Numerator*	WORD
input	*Denominator*	WORD
output	*Quotient*	LONG
output	*Remainder*	LONG

UDivide

Divides two 16-bit inputs and produces two 16-bit unsigned results.

input	*ResultSpace*	LONG
input	*Numerator*	WORD
input	*Denominator*	WORD
output	*Quotient*	LONG
output	*Remainder*	LONG

LongMul

Multiplies two 32-bit inputs and produces a 64-bit result.

input	*ResultSpace*	LONG
input	*ResultSpace*	LONG
input	*M1*	LONG
input	*M2*	LONG
output	*Result*	LONG
output	*Result*	LONG

If the inputs were unsigned, the 64-bit *Result* is unsigned. If the inputs were signed, the low two words of the 64-bit *Result* indicate the sign.

LongDivide

Divides two 32-bit inputs and produces two 32-bit unsigned results.

input	*ResultSpace*	LONG
input	*ResultSpace*	LONG
input	*Numerator*	LONG
input	*Denominator*	LONG
output	*Quotient*	LONG
output	*Remainder*	LONG

FixRatio

Takes two signed integer inputs and produces a two-word fixed-point number as a ratio of the numerator and denominator.

input	*ResultSpace*	LONG
input	*Numerator*	LONG
input	*Denominator*	LONG
output	*Result* (least significant)	LONG
output	*Result*(most significant)	LONG

FixMul	Multiplies two fixed-point inputs and produces a two-word fixed-point result.

input	*ResultSpace*	LONG
input	*M1*	LONG
input	*M2*	LONG
output	*Result* (least significant)	LONG
output	*Result*(most significant)	LONG

FracMul	Multiplies two Frac inputs and produces a Frac result.
FixDiv	Divides two fixed-point inputs and produces a fixed-point result.
FracDiv	Divides two Frac inputs and produces a Frac result.
FixRound	Takes a fixed-point input and produces a rounded integer result.
FracSqrt	Takes a Frac input and produces a Frac square root.
FracCos	Takes a Frac input and produces its cosine.
FracSine	Takes a Frac input and produces its sine.
FixATan2	Takes two inputs and produces a fixed point arc tangent of their ration. The inputs can be long integer, fixed, or Frac.
HiWord	Returns high word of input.
LoWord	Returns low word of input.
Long2Fix	Converts long integer to fixed.
Fix2Long	Converts fixed to long integer.
Fix2Frac	Converts fixed to Frac.
Frac2Fix	Converts Frac to fixed.
Fix2X	Converts fixed to extended.

Frac2X	Converts Frac to extended.
X2Fix	Converts extended to fixed.
X2Frac	Converts extended to Frac.

Battery RAM Functions

WriteBRam

Writes 256 bytes of data from a specified address to the battery RAM.

input	*BufferAddress*	LONG

ReadBRam

Reads 256 bytes of data from the battery RAM and transfers it to a specified address.

input	*BufferAddress*	LONG

WriteBParam

Writes data to a specified parameter in battery RAM.

input	*Data*	WORD
input	*ParamRef*	WORD

ParamRef is from 0-255, and is defined as below for **ReadBParam**.

ReadBParam

Reads one byte of data from battery RAM at a specified parameter address.

input	*ParamRef*	WORD
output	*Data*	WORD

ParamRef is from 0-255, and is defined as follows:

$0B	Port2 Printer/Modem
$0C	Port2 Line Length
$0D	Port2 delete if after cr
$0E	Port2 add lf after cr
$0F	Port2 Echo
$10	Port2 Buffer
$11	Port2 Baud
$12	Port2 Data Bits
$13	Port2 Stop Bits
$14	Port2 Parity
$15	Port2 DCD Handshake
$16	Display Color/Monochrome

$17	Display 40/80 Column
$18	50/60 Hz
$19	Display Text Color
$1A	Display Background Color
$1B	Display Border Color
$1C	User Volume
$1D	Bell Volume
$1E	System Speed
$1F	Slot1 Internal/External
$20	Slot2 Internal/External
$21	Slot3 Internal/External
$22	Slot4 Internal/External
$23	Slot5 Internal/External
$24	Slot6 Internal/External
$25	Slot7 Internal/External
$26	Startup Slot
$27	Text Display Language
$28	Keyboard Language
$29	Keyboard Buffering
$2A	Keyboard Repeat Speed
$2B	Keyboard Repeat Delay
$2C	Double Click Time
$2D	Flash Rate
$2E	Shift Caps/Lower Case
$2F	Fast Space/Delete Keys
$30	Dual Speed
$31	High Mouse Resolution
$32	Month/Day/Year Format
$33	24/am-pm Format
$34-35	Minimum Ram for RAMDISK
$36-37	Maximum Ram for RAMDISK
$38-39	Free space for RAMDISK
$3A-42	Number of Languages
$43-54	Number of Layouts
$55-7F	Reserved
$80	AppleTalk node number

Clock Routines

These routines allow the clock to be set or read. Setting the clock requires that the time be passed as an input parameter in a hex format.

Two tools are provided for reading the clock. One returns time in a hex format; the other returns time in an ASCII format.

ReadTimeHex Returns current time in Hex format.

input	*ResultSpace*	WORD
input	*ResultSpace*	WORD
input	*ResultSpace*	WORD
output	*Year/Day*	WORD
output	*Month/Seconds*	WORD
output	*Minute/Hour*	WORD

WriteTimeHex Sets clock to time specified in Hex format.

input	*Minute/Hour*	WORD
input	*Month/Seconds*	WORD
input	*Year/Day*	WORD
input	*Status*	WORD

Status indicates which parameters have changed, as follows:

Bit 6-15	Reserved
Bit 5	1 if Year not changed
Bit 4	1 if Day not changed
Bit 3	1 if Month not changed
Bit 2	1 if Second not changed
Bit 1	1 if Minute not changed
Bit 0	1 if Hour not changed

ReadASCIITime Reads elapsed time since 00:00:00, January 1, 1904, converts the elapsed time to ASCII time output, and places the output in a specified buffer.

input	*BufferAddress*	WORD

The ASCII time is in *HH:MM:SS mm/dd/yy* format, where:

HH	Hour
MM	Minute
SS	Second
mm	Month
dd	Day
yy	Year

The ASCII string has the high bit cleared.

Text Routines

The routines specified below talk to any card that supports Pascal entry points.

WriteChar

Combines a character with the global AND mask and global OR mask, and then writes the character to the Pascal device specified by the global slot number.

Input	*Character*	WORD

WriteLine

Combines a character string with the global AND mask and global OR mask, and then writes the string to the Pascal device specified by the global slot number. A carriage return and line feed are concatenated to the string.

Input	*StringPtr*	LONG

The first byte of the character string specifies the length of the string.

WriteString

Combines a character string with the global AND mask and global OR mask, and then writes the string to the Pascal device specified by the global slot number.

Input	*StringPtr*	LONG

The first byte of the character string specifies the length of the string.

WriteText

Combines text from a specified location (pointer + offset) with the global AND mask and global OR mask, and then writes the string to the Pascal device specified by the global slot number.

Input	*TextPtr*	LONG
Input	*Offset*	WORD
Input	*Count*	WORD

TextPtr + *Offset* specifies the memory location of the start of the string; *Count* specifies the length of the string.

WriteCString

Combines a character string terminating with the value $00 with the global AND mask and global OR mask, and then writes the string to the Pascal device specified by the global slot number.

Input	*CStringPtr*	LONG

ReadChar

Reads a character from the Pascal device specified by the global slot number, combines the character with the global AND mask and global OR mask, and returns that combination as a result.

Input	*ResultSpace*	WORD
Output	*Character*	WORD

ReadBlock

Reads a block of characters from the Pascal device specified by the global slot number, combines the characters with the global AND mask and global OR mask, and writes the block to a specified memory location.

Input	*ResultSpace*	WORD
Input	*Pointer*	LONG
Input	*Offset*	WORD
Input	*MaxBlockSize*	WORD
Output	*CharsReceived*	WORD

Pointer + *Offset* specifies the starting memory location to write to.

ReadLine

Reads a character string terminating in an EOL character from the Pascal device specified by the global slot number, combines the characters with the global AND mask and global OR mask, and writes the string to a specified memory location.

Input	*ResultSpace*	WORD
Input	*BufferPointer*	LONG
Input	*MaxBlockSize*	WORD
Input	*EOLCharacter*	WORD
Output	*CharsReceived*	WORD

BufferPointer specifies the buffer to write to.

InitPDev

Initializes the Pascal device.

Input	*ResultSpace*	WORD

ControlPDev

Initializes the Pascal device.

Input	*ControlCode*	WORD

StatusPDev

Makes the status call to the Pascal device.

Input	*RequestCode*	WORD

SetInGlobals

Sets the global parameters for the input device.

Input	*AndMask*	WORD
Input	*OrMask*	WORD
Input	*SlotNumber*	WORD

SetOutGlobals

Sets the global parameters for the output device.

Input	*AndMask*	WORD
Input	*OrMask*	WORD
Input	*SlotNumber*	WORD

GetInGlobals

Returns the global parameters for the input device.

Input	*ResultSpace*	WORD
Input	*ResultSpace*	WORD
Input	*ResultSpace*	WORD
Output	*AndMask*	WORD
Output	*OrMask*	WORD
Output	*SlotNumber*	WORD

GetOutGlobals

Returns the global parameters for the output device.

Input	*ResultSpace*	WORD
Input	*ResultSpace*	WORD
Input	*ResultSpace*	WORD
Output	*AndMask*	WORD
Output	*OrMask*	WORD
Output	*SlotNumber*	WORD

Vector Initialization Routines

These routines allow the application to set or get the current vector for the interrupt handlers.

SetVector

Sets the vector address for the interrupt manager or handler specified by the vector reference number.

Input	*VectorRefNumber*	WORD
Input	*Address*	LONG

VectorRefNumbers are given below, under **GetVector**.

GetVector

Returns the vector address for the interrupt manager or handler specified by the vector reference number.

Input	*ResultSpace*	WORD
Input	*VectorRefNumber*	WORD
Input	*Address*	LONG

VectorRefNumbers are as follows:

$00	Tool Locator #1
$01	Tool Locator #2
$02	User's Tool Locator #1
$03	User's Tool Locator #2
$04	Interrupt Manager
$05	COP Manager
$06	System Death Handler
$07	AppleTalk Interrupt Handler
$08	Serial Com. Controller Interrupt Handler
$09	Scan Line Interrupt Handler
$0A	Sound Interrupt Handler
$0B	Vertical Blanking Interrupt Handler
$0C	Mouse Interrupt Handler
$0D	Quarter Second Interrupt Handler
$0E	Keyboard Interrupt Handler
$0F	FDB Response Byte Interrupt Handler
$10	FDB SRQ Interrupt Handler
$11	Desk Accessory Manager
$12	Flush Buffer Handler
$13	Key Micro Interrupt Handler
$14	One Second Interrupt Handler
$15	EXT VGC Interrupt Handler
$16	Other Unspecified Interrupt Handler
$17	Cursor Update Handler
$18-FF	Invalid

HeartBeat Interrupt Queue

These tools allow a vector to be installed or removed from the HeartBeat Interrupt service queue.

SetHeartBeat

Installs the task specified by the pointer into the HeartBeat Interrupt service queue.

Input	*Pointer*	LONG

You must precede the task with a long word pointer which the tool uses to link to the next HeartBeat interrupt service task; a word parameter for a count which is used by the handler to keep track of how many VBL occurences remain before service is rendered; and a word parameter containing a signature value used to verify the presence of the task header. The task should end by executing an RTL back to the HeartBeat Interrupt handler. When this call is made, the tool assumes that the heartbeat interrupt handler will be used, and installs the HeartBeat interrupt handler into the VBL interrupt vector. An example is as follows:

```
HEARTBEAT   EQU    *
            DS     4,0          ;Link to next task
TASKCNT     DW     $nnnn        ;# VBL's until service
            DW     $A55A        ;Signature word
START       EQU    *            ;task starts here
            LDA    #nnnn        ;task must reset count
            STA    TASKCNT
             :
             :
            RTL                 ;back to handler
```

The count word and link long word are initialized by the tool. The count word is decremented by the HeartBeat interrupt handler, and is reset by the task. When a task is installed in the HeartBeat chain, the four bytes reserved for the link will be loaded with a $0000. The four bytes reserved for the link in the procedure just previous to the procedure currently being installed will be loaded with the address of the procedure currently being installed. Count specifies how many heartbeats remain before service is rendered to a procedure.

You can install ROM-based heartbeat tasks, but to do so you must permanently allocate twelve bytes of RAM to the task. The task header must be loaded into RAM, followed by a JMP instruction and the address of the ROM-based task as shown below:

```
HEARTBEAT   EQU   *
            DS    4,0         ;Link to next task
TASKCNT     DW    $nnnn       ;# VBL's until service
            DW    $A55A       ;Signature word
START       EQU   *           ;task starts here
            JMP   >ROMTASK    ;jump to rom based task
```

The ROM-based task still has the responsibility of resetting the task counter.

```
ROMTASK     EQU   *
            LDA   #nnnn       ;task must reset count
            STA   TASKCNT
             :
             :
            RTL               ;back to handler
```

DelHeartBeat

Deletes the task specified by the link address from the HeartBeat Interrupt service queue.

Input	*Pointer*	LONG

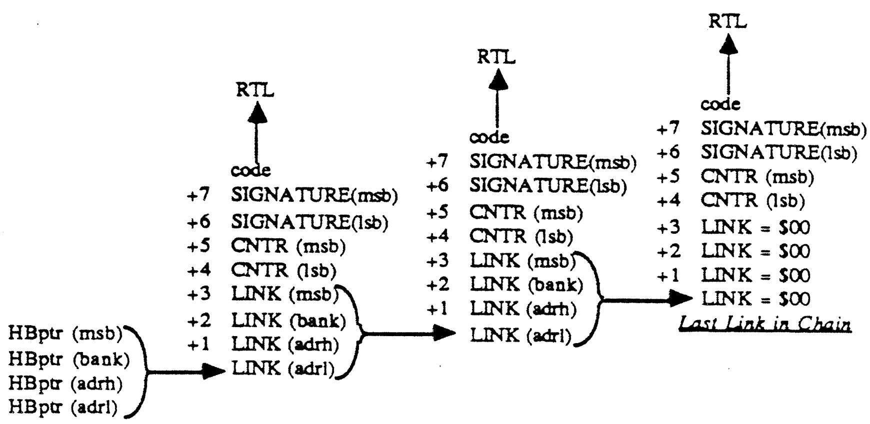
Errors that may occur when making tool calls to set or delete heartbeat tasks are as follows:

Error Code	Descriptions
$0003	Task already installed in heartbeat queue.
$0004	No signature in task header.
$0005	Queue has been damaged.
$0006	Task was not found in queue.

ClrHeartBeat

Clears the HeartBeat Interrupt service queue.

HeartBeat Queue



System Death Manager

This tool call vectors through the system death vector. At system power-up time, a default System Death Manager's vector will be installed. The default System Death Manager will display either a default or application-specific error message and an error code. If the pointer to the system death message is set to a value of $00000000, then the default system death message will be displayed.

SysDeathMGR Causes system death.

Input	*ErrorCode*	WORD
Input	*PointerMsg*	LONG

System Death error codes are as follows :

Error Code	Description
$0004	Divide by zero.
$0015	Segment loader error.
$0017-$0024	Can't load 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-$0053	Memory Manager error.
$0100	Can't mount system startup volume.
$0200	Heartbeat Task Queue damaged.

Get Address

This tool call returns an address of a byte, word, or long parameter referenced by the firmware.

GetAddr Returns the address of a byte, word, or long parameter.

Input	*ResultSpace*	LONG
Input	*RefNumber*	WORD
Output	*PtrToIntStatus*	LONG

RefNumbers are defined below:

Ref. #	Length	Parameter
0	Byte	IRQ Interrupt Flag (IRQ.IntFlag)
1	Byte	IRQ Data Flag (IRQ.DataReg)
2	Byte	IRQ Serial Port 1 Flag (IRQ.Serial1)
3	Byte	IRQ Serial Port 2 Flag (IRQ.Serial2)
4	Byte	IRQ Apple Talk Flag (IRQ.APTLKHI)
5	LongWord	HeartBeat Tick Counter (TickCnt)

The two bytes of interrupt status are defined as follows:

IRQ.IntFlag	D7	1 = mouse button currently down
	D6	1 = mouse button was down on last read
	D5	Status of AN3
	D4	1 = 1/4 second interrupted
	D3	1 = VBL interrupted
	D2	1 = Mega II mouse switch interrupted
	D1	1 = Mega II 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	If all bits clear then no FDB data valid, else the bits indicate the number of valid bytes received minus 1. (2-8 bytes total)

Mouse Tools

These tools interface with the mouse firmware. They can be used to set mouse mode, inquire about mouse status, read the clamp and position values, and set the clamp values.

ReadMouse

Returns mouse position, status, and mode.

Input	*ResultSpace*	WORD
Input	*ResultSpace*	WORD
Input	*ResultSpace*	WORD
Output	*Xposition*	WORD
Output	*Yposition*	WORD
Output	*Status&Mode*	WORD

InitMouse

Initializes mouse clamp values to $000 minimum and $3FF maximum, and clears mouse mode and status.

Input	*MouseLook*	WORD

MouseLook values are as follows:

0 = Search for mouse
1-7 = Specify mouse slot

SetMouse

Sets the mode value for the mouse.

Input	*ModeValue*	WORD

Modevalue is as follows:

$00	Turn mouse off.
$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, VBLIRQ active.
$09	Set transparent mode, VBLIRQ active.
$0B	Set movement interrupt mode, VBLIRQ active.
$0D	Set button interrupt mode, VBLIRQ active.
$0F	Set button or movement interrupt mode, VBLIRQ active.

HomeMouse Positions mouse at minimum clamp position.

ClearMouse Sets both X and Y axis position to $000.

ClampMouse Sets clamp values to new values, and then sets mouse position to the minimum clamp values.

Input	*XaxisMinClamp*	WORD
Input	*XaxisMaxClamp*	WORD
Input	*YaxisMinClamp*	WORD
Input	*YaxisMaxClamp*	WORD

GetMouseClamp Returns the current mouse clamp values.

Input	*ResultSpace*	WORD
Input	*ResultSpace*	WORD
Input	*ResultSpace*	WORD
Input	*ResultSpace*	WORD
Input	*YaxisMinClamp*	WORD
Input	*YaxisMaxClamp*	WORD
Input	*XaxisMinClamp*	WORD
Input	*XaxisMaxClamp*	WORD

PostMouse Positions mouse at the coordinates specified in the input parameters.

Input	*Xposition*	WORD
Input	*Yposition*	WORD

ServMouse Returns the mouse interrupt status.

Input	*ResultSpace*	WORD
Output	*IntStatus*	WORD

ID Management

This tool is used to insert, delete, or inquire status regarding an identification reference. The ID is used to tag segments as belonging to a specific application or desk accessory.

GetNewID Returns a value ID number and type.

Input	*Type*	WORD
Output	*IDnum&Type*	WORD

DeleteID Removes a specified ID from the current ID list.

Input	*IDnum&Type*	WORD

StatusID Returns with Carry set if ID not active, Carry clear if ID is active.

Input	*IDnum&Type*	WORD

Interrupt Control

This tool allows certain interrupt sources to be enabled or disabled.

IntSource Enables or disables the interrupts source specified by the source reference number.

Input	*SrcRefNumber*	WORD

SrcRefNumbers are shown below:

Ref. #	Source
$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	Enable Keyboard Buffering
$0009	Disable Keyboard Buffering
$000A	Enable FDB Data Interrupts
$000B	Disable FDB Data Interrupts

Firmware Entry Points

FWentry

Allows some Apple II entry points to be called from full native mode.

Input	*ResultSpace*	BYTE
Input	*ResultSpace*	WORD
Input	*ResultSpace*	WORD
Input	*ResultSpace*	WORD
Input	*AregisterToFirm*	WORD
Input	*XregisterToFirm*	WORD
Input	*YregisterToFirm*	WORD
Input	*EntryRefNumber*	WORD
Output	*AregisterFromFirm*	WORD
Output	*XregisterFromFirm*	WORD
Output	*YregisterFromFirm*	WORD
Output	*ProcessorStatus*	BYTE

Note that all inputs in word format will be truncated to a byte value prior to dispatching to the firmware entry point.

Reference #	Entry point	Address
0	Bell1	$FBDD
1	Wait	$FCA8
2	Count	$FDED

Tick Counter

GetTick

Returns the current value of the tick counter.

Input	*ResultSpace*	LONG
Output	*TickCount*	LONG

Basic Entry Points

The following functions allow the basic entry points to be called from full native mode. The functions use the global parameters defined earlier in this tool set.

BasicInit

Initializes the basic device, as defined by the output slot in the Global parameters.

Input	*InitCharacter*	WORD
Output	*TickCount*	LONG

InitCharacter must have the character in the low byte of the word.

BasicIn

Returns data from the basic device.

Input	*ResultSpace*	WORD
Output	*Data*	WORD

The *Data* is returned in the low byte of the word.

BasicOut

Outputs a data byte to the basic device.

Input	*Data*	WORD

HEX to ASCII

HexIt

Converts a word integer into four ASCII bytes.

Input	*ResultSpace*	LONG
Input	*Integer*	WORD
Output	*ASCIIint*	LONG

PackBytes and UnPackBytes

PackBytes and Unpackbytes provide for the packing and unpacking of any data. The functions are usually used for graphic images.

PackBytes

Packs bytes into packed format.

Input	*StartPtr*	LONG
Input	*SizePtr*	LONG
Input	*BufferPtr*	LONG
Input	*BufferSize*	LONG
Output	*NumPackBytes*	WORD

StartPtr is equal to the start of the area to be packed. *SizePtr* is equal to a WORD containing the size of the area. *BufferPtr* is equal to the start of the output buffer area.

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. Therefore, packing data and writing it to a file could be accomplished by using code similar to the Pascal code segment that follows:

.
.
.

```
FUNCTION packbytes (    VAR picptr     : POINTER;
                        VAR picsize    : POINTER;
                            bufptr     : POINTER;
                            bufsize    : INTEGER)
    : 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 Unpacks bytes from packed format.

Input	*BufferPtr*	LONG
Input	*BufferSize*	WORD
Input	*UnpackAreaPtr*	LONG
Input	*SixePtr*	LONG
Output	*NumUnpackBytes*	WORD

BufferPtr points to the buffer containing the packed data. *BufferSize* contains the size of the buffer containing the packed data. *UnpackAreaPtr* is a pointer to the area where the unpacked data will be placed. *SixePtr* contains the size of the area to contain the unpacked data. *NumUnpackBytes* is the number of bytes unpacked.

Upon completion, the pointer to the unpacked data is positioned one past the last unpacked byte and the size of the area is reduced by the amount unpacked. Therefore, the following Pascal code segment could be used to unpack data from a file:

.

```
FUNCTION unpackbytes (        bufptr     : POINTER;
                              bufsize    : INTEGER;
                         VAR  picptr     : POINTER;
                         VAR  picsize    : POINTER)
   : INTEGER;   EXTERNAL;
                 .
                 .
                 .
mark := 0;             {i.e. start of a 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 - all different
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 repeats of next 1 byte taken as 4 bytes (as in '10' case)