QPTR

The Pointer Environment

5th Edition

Contents

Introduction

The Pointer Toolkit

The Pointer Toolkit is aimed at applications programmers who wish to produce programs of the new "user-friendly" type. While many writers have produced very successful menu- and pointer-driven programs, there have so far been no agreed standards, resulting in users having to learn a new interface for each program, and each programmer having to re-invent the wheel to implement his own menu and/or pointer system. With the advent of the QJUMP Pointer Environment, all this is in the past. The programmer is relieved of the burden of writing the whole of the user interface, often 90% of the programming effort, and can concentrate on providing a good range of facilities. Users end up with a program which they know how to drive even before they open the box.

The Pointer Environment is a complex piece of software which has been in development for over a year at the time of writing, and is still being improved today. We therefore make no apology for the length of this manual, nor for the amount of effort required to start using the software: if it were an evening's work to learn all about it, it would not be a useful tool. We realise that there are likely to be aspects of the software which programmers would like to see treated in greater detail: anyone experiencing problems in using the software is always welcome to contact us (preferably by letter) and we will do our best to advise.

The software is in several parts. The Pointer Interface extends and modifies the QL's standard screen driver (the CON_/SCR_ device), taking care of the non-destructive windows and the position and appearance of the pointer sprite (arrow, padlock etc.): in addition it provides some extra TRAPs to read the pointer position, save window contents, write graphics objects and so on.

The Window Manager provides a set of utilities for manipulating windows. It works on data set up in memory, defining the size, position, colour and contents of windows. Routines are provided to draw, move and remove a window, re-draw part of a window, and to get user input via a window. If used from machine code then the programmer may provide routines to be called under particular circumstances (e.g. hitting the QUIT item): from SuperBASIC the options are more limited, since SuperBASIC procedures may not be called from within machine code routines. The Pointer Interface must be present to use the Window Manager.

The combination of the Pointer Interface and Window Manager is called the Pointer Environment.

The SuperBASIC Pointer Toolkit gives the SuperBASIC programmer access to the Pointer Environment via a set of special procedures and functions. While not quite as flexible as machine code, particularly when using the Window Manager, it provides a suitable base from which to explore the system before attempting to use it from machine code. Both the Pointer Interface and the Window Manager must be present to use the Pointer Toolkit.

Various applications are provided as examples of machine code and SuperBASIC programs using the Pointer Environment: the SuperBASIC programs require the Pointer Toolkit, the machine code ones do not. The SuperBASIC sprite editor EDSPR uses only the extension routines that call the Pointer Interface: the painting program PAINT also uses the Window Manager routines. There is a DEMO program which was written in SuperBASIC and then re-written in machine code: both versions do the same things, but achieve them in slightly different ways.

For the machine code programmer there are some INCLUDE files of the keys needed to use the Pointer Environment from assembler programs: a set of macros is also provided to assist with setting up window definitions. These are suitable use with the GST Macro Assembler and Linker: other assemblers and linkers may need modified versions.

Where to start

You should read the next section, describing the Pointer Environment and some of the concepts it uses. Once you understand this you are well on the way to being able to write your own programs. The next stage is to examine the **DEMO** program, either the SuperBASIC **_BAS** version or the **_ASM** and **_BIN** assembler version, depending on how strong you feel! The demo doesn't do anything very useful, but it does show you how to set up a simple menu with all the facilities described.

After this, you're on your own. SuperBASIC programmers will find a description of the new routines in the **Keywords** section, with a quick reference index at the end. Assembler programmers have a description of the new TRAPs in the **Pointer Interface** section, and the manager vectors in the **Window Manager** section of the **Programmer's Interface** chapter. Of interest to all will be the **Concepts** chapter, and the **Data Structures** section of the **Programmer's Interface** chapter, although the latter is essential reading only for assembler programmers.

Compiled SuperBASIC

You may wish to compile SuperBASIC programs using the Pointer Toolkit to take advantage of the increased speed and multitasking which are made possible by compiled SuperBASIC programs. There are some problems in doing this, whether you are using Digital Precision's Super/Turbocharge compilers or Liberation Software's Q_Liberator.

Supercharge and Turbo do not permit machine code extensions to return changed parameter values, and so the extensions to read the pointer position, **RPTR**, and to set one line of a sprite, **SPLIN**, will not work. Furthermore, array parameters are not permitted, so neither **SPSET** nor the majority of the Window Manager extensions will work.

Q_Liberator restricts the amount of stack that a machine code extension may use to a smaller amount than that provided by the interpreter: while both allowances are more than stated in the QL Technical Guide, the large amount of stack used by the Window Manager causes problems with Q_Liberated programs compiled using versions up to and including v3.12. Versions 3.21 onwards have an increased stack allowance which fixes this problem, and a utility program, called STKINC, is provided to overcome this problem in older versions of Q_Liberator - see the Utilities chapter for details.

Bug "fixes"

Some toolkits and extensions "fix bugs" in SuperBASIC by replacing ROM routines with their own: where these cause more trouble than they cure the old routine may be restored using the FIXPF utility, described in the Utilities chapter.

History, Geography, Philosophy & Economics

Why the world is the way it is

As you will have noticed, all QJUMP software comes split into a number of separate components, which need to be assembled correctly to "install" the new facilities on your QL. Why have we made life so difficult for you?

In the beginning (always a good start, that), the QL was designed to be an expandable multi-tasking machine, allowing you to use software from many suppliers simultanously to achieve an environment that you can work with comfortably. If you feel that your word processor program is too large or too slow, you can change to another one without changing your spreadsheet or database, which must surely be an improvement over the pre-packaged "integrated programs" available for the current series of IBM PCs and clones. The situation is very like buying hi-fi. You can go for the music centre or tower system, with everything in one box and known to be compatible, or you can take a little more trouble and buy separate components from different manufacturers: the latter solution may result in a bird's nest of wire and a pile of different styled boxes, but the performance will probably be closer to what you were after.

Given the above design philosophy, software for the QL falls into two categories. "Resident extensions" expand the facilities available to the system, by adding new devices or SuperBASIC procedures: RAM disks and SuperToolkit II are examples of resident extensions. "Transient programs" provide services to the user, allowing you to edit text or pictures, play games or what have you: Quill is a typical transient program. As implied by the name, resident extensions are designet to be loaded at the start of a session, and remain resident until the QL is restarted. They should be loaded into the "resident procedure area": space for the extensions may be reserved in this by a call to SuperBASIC's RESPR function, and cannot be freed once allocated. Transient programs are started by the user as required, and disappear from memory when terminated, leaving it free for other transient programs. Space for transient programs is allocated in the "transient program area" by SuperBASIC's EXEC procedure or QPAC II's EXEC etc. menus., and automatically reclaimed by the operating system when the program is terminated.

A limitation imposed by the operating system in the QL is that while there are programs in the transient program area, additional space may not be allocated in the resident procedure area. If you try to allocate more space, using RESPR or LRESPR commands, you will get a "not complete" error message. Ideally you will know what extensions may be required during a session, and arrange for them all to be loaded before starting any programs. In an emergency you can remove all transient programs so that another extension can be loaded, but this is not very convenient! The reason for the limitation is that transient programs "live" just below resident extensions in the memory, both "grow" downwards, and transient programs cannot be moved to make space for new extensions.

The reason for QJUMP's software being split into separate components thus becomes clear. Some components can be written in such a way that they extend the facilities available via operating system, for instance by adding new devices or extending old ones. The Pointer Interface extends the Screen Device Driver, the SPELL device is a completely new one. These extended facilities can then be used, not only by the other components of the software package as supplied, but also by other software writers in their own code. The benefits of this approach are manifold. Firstly, any "dirty" code that is required can be buried out of sight in the extensions, so applications that use them can be totally clean: if any problems arise from the dirty code then only the extensions need be changed. Secondly, the extensions will often provide much of the "difficult" code: writing a menu-driven spelling-checking word processor is much simpler if you don't have to consider how to implement pull-down menus or the best method of complessing a word list. Thirdly, applications can be smaller, leaving more space for further applications or user data, and making them easier to debug. This is particularly valuable with the Pointer Environment, which occupies about 25k. If it were included in indivdual programs, then they would be approximately that much bigger, and you would not get the benefit of non-destructive windows in other programs.

So the typical QJUMP software package consists of a set of "public" extensions, which are loaded in by your BOOT program, plus the application itself, which may be EXECuted as required. The applications themselves tend to be quite small, because they share the extensions with others.

Where it is useful to run more than one copy of an application at once, a further trick may be addeed: a separate job may be started for each copy, but the same code can be shared by both jobs, thus economising on the total space required. This will only work if the application has been written properly, so that is does not modify its own code or embedded data. In this case the code is said to be "re-entrant". This approach is used by the "hotkey" facility provided by the QRAM package, and is improved by the HOTKEY System II, which comes with QTYP II, QPAC II, QD or the QL-Emulator for the ATARI ST. Each time when a given hotkey is pressed a new copy of an application is started as if executed from microdrive or disk, but without the same speed or memory penalty.

Some sample BOOT files

The QL's BOOT facility is intended to be used to set up the QL with all the resident extensions required for a session, which may come from many different sources. The BOOT file is also used in much commercial software to give users instant access to their new software - many users never progress beyond this point, but re-boot their QLs every time they wish to change programs!

Modifying your existing BOOT program to cope with new software can vary from the very easy to the impossible. Very easy BOOT files would consist of EXEC devN_filename, in which case no changes are necessary to your own BOOT. Difficult conversions are where the software's original BOOT file indulges in copyright messages, pretty borders, playing tunes or other methods of obscuring the useful bits of code. Impossible BOOT files are those which include POKEs, or start an application with a CALL statement - these can sometimes be used, but require the attention of an expert machine code hacker to convert them to a sanitary form.

To modify your BOOT program, you will have to determine which resident extensions are needed to run the software. This may be apparent from the manual, or can be found by examining the software's own BOOT file: any code loaded by statements of the form:

```
  base=RESPR(size):LBYTES devN_filename,base:CALL base
```

may be assumed to be a resident extension. The statements can be copied into your own BOOT file at the appropriate point, and the file itself copied to your normal BOOT disc. The above form may be scattered over a number of lines, or obscured by reserving just one area with the RESPR call and LBYTESing several files into it, but the principle remains the same.

In the following examples, the file sizes given are not necessarily accurate: you should use the QRAM Files menu or SuperToolkit II to find the actual size required. It is assumed that the boot medium is in "flp1_": this can of course be changed to any device of your choice. All the examples use the "ptr_gen" version of the Pointer Interface, which works with the QJUMP Internal Mouse Interface, the QL-Emulator for the ATARI ST or the Sandy SuperMouse interface, as well as the keyboard. It supersedes previous versions of the Pointer Interface such as "ptr_kbd", "ptr_imi" and that invoked by the Sandy SuperMouse POINTER command.

1)A simple BOOT file to load and enable QRAM

```
100 base=RESPR(12388):LBYTES flp1_ptr_gen,base:CALL base
110 base=RESPR(7762):LBYTES flp1_wman,base:CALL base
120 base=RESPR(25882):LBYTES flp1_hotkey,base:CALL base
130 HOTKEY
```

The HOTKEY statement in line 130 starts a transient program called HOTKEY, which is responsible for acting on the "ALT /" keystroke and starting QRAM. Once this program is present, it is impossible to reserve space for any more resident extensions without removing the HOTKEY program, so the HOTKEY statement will always occur after all the RESPR statements in the BOOT file.

2) Including SuperToolkit II with QRAM

```
100 base=RESPR(16384):LBYTES flp1_tk2_rext,base:CALL base
110 base=RESPR(12388):LBYTES flp1_ptr_gen,base:CALL base
120 base=RESPR(7762):LBYTES flp1_wman,base:CALL base
130 base=RESPR(25882):LBYTES flp1_hotkey,base:CALL base
140 HOTKEY
```

or

```
100 TK2_EXT
110 base=RESPR(12388):LBYTES flp1_ptr_gen,base:CALL base
120 base=RESPR(7762):LBYTES flp1_wman,base:CALL base
130 base=RESPR(25882):LBYTES flp1_hotkey,base:CALL base
140 HOTKEY
```

Line 100 initialises SuperToolkit II, in the first case from a file "tk2_rext" produced using the configurable version of the toolkit, in the second case from the ROM on a suitably-equipped disc interface.

3) A BOOT file for QRAM and QTYP together

```
100 base=RESPR(5424):LBYTES flp1_qtyp_spell,base:CALL base
110 base=RESPR(12388):LBYTES flp1_ptr_gen,base:CALL base
120 base=RESPR(7762):LBYTES flp1_wman,base:CALL base
130 base=RESPR(29538):LBYTES flp1_hotkey,base:CALL base
140 HOTKEY
```

As for the SuperToolkit II example, the SPELL extensions are loaded in the normal way: the QTYP program itself is assumed to be included in the "flp1_hotkey" file with QRAM.

4) SuperToolkit II, QMON, QRAM, QTYP, QPTR, and RAM disc

```
100 base=RESPR(16384):LBYTES flp1_tk2_rext,base:CALL base
110 base=RESPR(11242):LBYTES flp1_qmon_bin,base:CALL base
110 base=RESPR(5424):LBYTES flp1_qtyp_spell,base:CALL base
120 base=RESPR(12388):LBYTES flp1_ptr_gen,base:CALL base
130 base=RESPR(7762):LBYTES flp1_wman,base:CALL base
140 base=RESPR(29538):LBYTES flp1_hotkey,base:CALL base
150 base=RESPR(9234):LBYTES flp1_qptr,base:CALL base
160 base=RESPR(5108):LBYTES flp1_ramprt,base:CALL base
170 HOTKEY
200 OUTLN #0;512,256,0,0
210 IF RMODE=8 THEN
220   WINDOW #0;448,40,32,216
230 ELSE
240   WINDOW #0;512,50,0,206
250 END IF
260 AT #0;1,0
```

This loads all QJUMP products. Apart from having to load "wman" after "ptr_gen", the order of files is unimportant. As usual, the call to HOTKEY must come last. Lines 200 onward are needed if the Pointer Toolkit is to function correctly.

5) QRAM and Jochen Merz's QD

```
100 base=RESPR(12388):LBYTES flp1_ptr_gen,base:CALL base
110 base=RESPR(7762):LBYTES flp1_wman,base:CALL base
120 base=RESPR(25882):LBYTES flp1_hotkey,base:CALL base
130 base=RESPR(14386):LBYTES flp1_menu_rext,base:CALL base
140 HOTKEY
```

QD Version 2 or 3 from Jochen Merz Software requires the Menu Extension if it is to run, so the "menu_rext" file is loaded in the BOOT file. A copy of this Editor may then be started at any time by EXECuting it from SuperBASIC, thus:

```
EXEC flp1_QD
```

It may also be started from QRAM's or QPAC II's Files menu, of course.

6) QRAM and Q_Liberator runtime system and extensions

```
100 base=RESPR(10016):LBYTES flp1_qlib_run,base:CALL base
110 base=RESPR(1928):LBYTES flp1_qlib_bin,base:CALL base
120 base=RESPR(1476):LBYTES flp1_qlib_ext,base:CALL base
140 base=RESPR(12388):LBYTES flp1_ptr_gen,base:CALL base
150 base=RESPR(7762):LBYTES flp1_wman,base:CALL base
160 base=RESPR(25882):LBYTES flp1_hotkey,base:CALL base
170 HOTKEY
```

This example loads the extensions used to run the Q_Liberator compiler, which may then be run as detailed in the manual. As the runtime system is also loaded, any Q_Liberated programs which do not include it may also be EXECuted.

QRAM is supplied with a utility called BOOT_MAKE, which may be used to speed loading of resident extensions by putting them all into one long file, which loads faster than many shorter files. As a side-effect, there may be a slight reduction in the amount of memory required.

BOOT_MAKE produces two files, a SuperBASIC file normally called "flp1_boot", and the resident extensions file which is of the same name but with the extension "_rext". Extension files may be copied from any number of source media into the "_rext" file, changing the source medium as required: as the destination medium is always being written to, it must stay in the drive until BOOT_MAKE has finished. The dialogue to produce an BOOT file equivalent to that described in example 5 above might be as follows:

```
Boot filename> flp1_boot
Command (ESC to finish)>
Extension file (ESC to finish)> flp2_xtras
Extension file (ESC to finish)> flp2_ptr_gen
Extension file (ESC to finish)> flp2_wman
Extension file (ESC to finish)> flp2_hotkey
Extension file (ESC to finish)>
Command (ESC to finish)> hotkey
Command (ESC to finish)>
```

The resulting BOOT file would be

```
100 base=RESPR(52106):LBYTES flp1_boot_rext,base:CALL base
110 hotkey
```

Pointer Environment

The Pointer Environment for the QL is a comprehensive display- handling interface which improves on the QL's simple window system. It differs from the QL's standard interface in two respects. Firstly, the interface allows overlapping non-destructive windows. Secondly, a window (and by association a job) may be selected for attention directly, using a pointer, as well as indirectly, using the "CTRL C" key on the keyboard.

These differences are intended to be as invisible as possible to existing software: in particular, a considerable amount of time has been spent ensuring that the commonly-used Psion packages will run happily. The major implication of the differences is that significantly more memory is required when using the Pointer Environment.

The Pointer Environment is implemented as two levels. The normal entry is to the Window Manager level, which handles windows and menus. The Pointer Interface level is used by the Window Manager and provides extra Trap #3 entries as used for standard IO operations.

Pointer

All pointer input from the user is directed to a point on the display. The pointer may be visible or invisible, and it may be moved by the cursor keys, joystick or pointing device or else its position may be set directly, either by the Window Manager as a result of a single keystroke, or by an application program.

An object shown on the display may be "hit" by moving the pointer to the object and pressing SPACE, the fire button on a joystick or the left button on a mouse. Within a menu, a keystroke may cause a "hit" as well as setting the pointer position. This allows a menu to be treated either as a single key command system, or else as a point and hit menu system. A "hit" on an item will usually select or de-select that item, but only rarely causes other action to be taken.

ENTER or the right mouse button is known as "do": this differs from a "hit" in that it usually selects the current item and results in an action being performed. The exact interpretation of the difference is ultimately left to the programmer.

Note that an application may only get pointer input from a "managed" window. It is thus very important that any window intended for pointer input should have had its outline set, to signal to the Pointer Interface that it is managed: see the SuperBASIC Keywords section, the Concepts chapter, and the Assembler Programmer's Interface section for details.

Windows

In the context of the Pointer Environment, a window is more than just a portion of the display. An application using the display has just one primary window. Sub-windows may be enclosed within this window, allowing multi-window operation of application programs. An application may open secondary windows within its primary window, but it may not use the area of the display outside its primary window. A secondary window may have sub-windows itself, each enclosed within the secondary window area. Such secondary windows are frequently used to provide pull-down menus. Depending on the complexity of the application, it may be useful to pull down further windows from within a pull-down menu: these "daughter" pull-down windows are limited to be within their parent primary, **not** their parent pull-down, otherwise pull-down menus would have to get progressively smaller!

The distinction between a sub-window and a secondary window is that a sub-window is merely a division of a window: it does not have its own channel. A secondary window, however, is a genuine IO channel with its own independent existence. The Window Manager utilities assume that when one or more secondary windows have been pulled down, all IO operations by that job will be carried out within the most recently pulled-down secondary until it is thrown away.

The size and position of a window (primary or secondary) may be changed by the job that owns it at any time: it is up to the programmer to provide this facility, where appropriate, to enable the user to adjust the display to execute as many jobs as he wishes at any one time.

Where primary windows overlap, the window below is locked until the window above is moved or removed, or the window below is brought to the top of the pile. It is possible to move a window to the top of the pile by "hitting" it. While a window is locked it may not be modified, so applications which rely on continuous modification of their windows (e.g. the ubiquitous clock programs) will not work as intended. It is possible to unlock windows, so that they become destructive.

Menus

The Window Manager includes facilities for handling menus. A menu is a collection of items which may be "hit". Menu items may be of several types: text, blobs, patterns and sprites. Menu items may also have several uses. "Hitting" an item may cause an action, it may select the item for some future action or it may cause a further pull down menu window to be invoked.

The primary window, and any other window pulled down, is treated as a menu. There are a number of standard menu items which will appear in many windows: these have standard "hit" keystrokes which should be used to keep software consistent between different packages.

CANCEL should always be present to enable a window to be removed without doing any (further) operation. This item should be "hit" by the keystroke ESC.
HELP should usually be present to provide assistance to the user. This item should be "hit" by the keystroke F1.
DO may sometimes be present to do any actions set up within the window. This item should be "hit" by the keystroke ENTER.
MOVE should usually be present to allow the window to be moved. This item should be "hit" by the keystroke CTRL F4.
SIZE will be present if it is possible to change the size of a window. This item should be "hit" by the keystroke CTRL F3.
WAKE will be present if it is possible to update the contents of a menu. This item should be "hit" by the keystroke CTRL F2.
SLEEP allows you to put the current menu to sleep, which means, set it to a button. This item should be "hit" by the keystroke CTRL F1.

A window is usually divided into sub-windows. There are information sub-windows, which are used for titles, general information etc.. There are menu sub-windows, which are used for collections of similar items under the control of the Window Manager level. And there are application sub-windows which are only used by the application code. An application sub-window has a similar structure to a menu sub-window, but omits part of the standard definition.

It is not necessary for menu items to be within a menu sub-window, they can be put anywhere within the window. This type of item is termed a loose menu item.

Sub-Windows

The function of the menu and application sub-windows is defined by the application itself (hence the name). Frequently they will be used to display large amounts of information, facilities being provided to scroll, pan or fold this information if there is not enough room for all the items or information within the sub-window.

The menu items for scrolling, panning and folding a sub-window are part of the definition of a sub-window, and should appear whenever the sub-window is too small to display all the information.

There may be a "scroll bar" to the right of a scrollable sub-window. This scroll bar is a map showing the portion of the sub-window contents which is actually visible within the vertical range of the sub-window contents. "Hitting" the scroll bar will scroll the sub-window to the hit position. Within the sub-window there may be arrow bars to allow the sub-window to be scrolled a row or a page at a time.

Similarly there may be a "pan bar" below a pannable sub-window. Panning and scrolling may also be invoked by ALT arrow and SHIFT ALT arrow keystrokes.

Folding a sub-window is accomplished by splitting the sub-window and independently scrolling or panning part of the sub-window. In order to keep track of which parts of a folded sub-window are visible, there may be an index row above the sub-window or an index column to the left of the sub-window (or both). Splitting or joining the parts of the sub-window is accomplished by a "do" keystroke on the scroll or pan bar to the right of or below the sub-window.

Objects, Items etc.

An object is something represented on the display. An object may be text, a sprite, a pattern or a blob. Text is just readable characters. A sprite is a picture of something, on a transparent background: a sprite is the only type of object which may be used as both a pointer and a menu item. A pattern is a (repeating) pattern of colours, but has no limits and so no shape. A blob defines a shape, but has no colour or pattern. Combining a blob with a pattern produces a visible object.

An item is part of a menu. An item may consist of more than one object. All the objects comprising an item are linked together, and so "hitting" one object within an item selects all the objects. To simplify the code and to make execution as fast as possible, all the objects within one item should be contiguous within the object list.

There are three main states for a menu item: unavailable (cannot be selected), available and selected. In addition, an available or selected item may be the current item (the item that the pointer points to) or not. The current item is indicated by a border around it, and the three main states are indicated by various colour attributes, blobs or patterns.

Window Definition

When a window is pulled down, or redrawn, the window definition provides all the information required to draw the window, its border, the menu items in the window, the sub-windows and their borders and the menu items within the sub-windows. After a window is pulled down, the menu definition provides all the information to process hits. Unfortunately, because a window may be moved and have its size and shape altered, much of the information will tend to be variable. The basic window definition is treated as invariant, as this will usually be either in ROM or in program RAM. On setting up a window, a variable RAM based "working definition" will be created. The table overleaf shows the structure of a window definition: it is described in more detail in the Data Structures section of the Assembler chapter.

Window definition
- window size
- window origin
- window attributes
- window pointer sprite
- window help pointer
- loose menu item attributes

- loose menu object list
  - object hit area
  - object justification rules
  - object type (text, sprite, pattern, blob)
  - object selection keystroke
  - object pointer
  - item number
  - action routine pointer

- information sub-window list
  - information sub-window size
  - information sub-window origin
  - information sub-window attributes
  - information object list
    - object size
    - object origin
    - object type (text, sprite, pattern, blob)
    - object attributes
    - object pointer

- application sub-window list
  - menu / application sub-window size
  - menu / application sub-window origin
  - menu / application sub-window attributes
  - pointer sprite pointer
  - setup routine pointer
  - draw routine pointer
  - hit routine pointer
  - control routine pointer
  - maximum number of control sections
  - sub-window selection keystroke

  - sub-window control definitions
    - control block pointer
    - index size/spacing
    - index item attributes
    - control item attributes

- menu item attributes
    - number of columns and rows
    - offsets to start of columns/rows
    - object spacing lists
        - object spacing
        - object hit area
    - row list
        - start object pointer
        - end object pointer
    - object lists
        - object justification rules
        - object type (text, sprite, pattern, blob)
        - selection keystroke
        - object pointer
        - item number
        - action routine

Event Vector

The event vector is a record of all the events which have occurred since a call was made. There are several levels to the complete Pointer Environment. On entry to each level, its events in the vector are cleared: on return through a level, the events which have occurred within that level are added to the vector.

The vector is a long word, each major level has 8 bits reserved for its own events

Pointer level	bit 0	keyclick
	bit 1	key down
	bit 2	key up
	bit 3	pointer moved
	bit 4	pointer out of window
	bit 5	pointer in window
	bit 6	
	bit 7	
Sub-window level	bit 8	sub-window split
	bit 9	sub-window join
	bit 10	sub-window pan
	bit 11	sub-window scroll
	bit 12	
	bit 13	
	bit 14	
	bit 15	
Window level	bit 16	do
	bit 17	cancel
	bit 18	help
	bit 19	move window
	bit 20	change size
	bit 21	sleep
	bit 22	wake
	bit 23	

What you get

The following two files are used to add the Pointer Toolkit facilities to the QL when you start it. You will probably wish to merge the BOOT file with your existing BOOT to include other extensions.

BOOT	
BOOT_REXT	contains PTR_GEN, WMAN, QPTR and STK2

Qram owners wishing to re-create their BOOT_REXT to include the Pointer Toolkit and upgraded Pointer Interface and Window Manager should include these files **in this order**. The PTR_GEN version of the Pointer Interface supports the QJUMP Internal Mouse Interface, the Thor and Atari ST keyboard and mouse interfaces, and the Sandy SuperQBoard with mouse interface. If for some reason you have both the SuperQBoard and QIMI then the QIMI is used. SuperQBoard owners should omit the POINTER command from their BOOT file, as PTR_GEN replaces and upgrades the built-in version of the Pointer Interface. If you have SuperToolkit II then you can omit STK2.

PTR_GEN	Pointer Interface, general version
WMAN	Window Manager
QPTR	SuperBASIC Pointer Toolkit
STK2	cut-down version of SuperToolkit II

The following three files are SuperBASIC demonstrations of the Pointer Toolkit.

DEMO_BAS	SuperBASIC version of the demo
PAINT_BAS	painting program, uses the Window Manager
PAINT	compiled version of the above
EDSPR_BAS	sprite designing program, does not use the Window Manager

The following files contain the assembler sources for a machine-code version of the above DEMO_BAS program, suitable for assembling and linking using the GST Macro Assembler. The last four are: two files of keys required, the linker command file to link with, and a ready-assembled and linked version of the program.

DEMO_ACTION_ASM	action and hit routines
DEMO_DRAW_ASM	window drawing routine
DEMO_INIT_ASM	initialisation and termination
DEMO_MLYOT_ASM	menu layout
DEMO_MMAIN_ASM	main menu definition
DEMO_SETUP_ASM	menu setup routine
DEMO_SPRITE_ASM	sprites used in the demo
DEMO_TEXT_ASM	text used in the demo
DEMO_WMAN_ASM	action routines that call the Window Manager
DEMO_KEYS	keys for the above files
DEMO_SMS	SMS2 keys used in the above files
DEMO_LINK	linker command file
DEMO_BIN	assembled version of the demo

The following files may be INCLUDEd in your own assembler files to define suitable symbols for the manipulation of the data structures in the Pointer Environment.

WMAN_KEYS	keys for vectors etc.
WMAN_WDEF	window definition structure
WMAN_WSTATUS	window status area structure
WMAN_WWORK	working definition structure
WMAN_MENU_MAC	menu generating macros
WMAN_TEXT_MAC	text string generating macros
QDOS_IO	keys used to access the Pointer Interface
QDOS_PT	external keys for the Pointer Interface
PTR_KEYS	internal keys for the Pointer Interface
KEYS_COLOUR	some useful colours
KEYS_K	symbolic names for keystrokes

Two utility programs are provided to modify screen images and compiled SuperBASIC programs. There is also a procedure to restore the ROM definitions of SuperBASIC procedures and functions. These are documented in the Utilities chapter.

CVSCR	convert screen utility
STKINC	stack increase utility
FIXPF	SuperBASIC "ROM restore" utility

Versions of the Pointer Interface and Window Manager as shipped with Qram v1.07 are included - they will only be of interest to writers of commercial software who wish their products to be compatible with older versions of the Pointer Environment.

OLD_PTR_KBD	old version of Pointer Interface (v1.05)
OLD_WMAN	old version of Window Manager (v1.03)

CONFIG - the standard configuration program - is explained in the last part of this manual. Two files are provided to allow you to implement own configuration blocks in your assembly programs.

CONFIG	the CONFIG program itself
CONFIG_MAC	macros for setting up config blocks
CONFIG_DEMO_ASM	a demonstration of the use of the macros

The Demonstration Programs

Four demonstrations are included with the Pointer Toolkit. The SuperBASIC ones will all run on a QL as set up by the BOOT file supplied. When you get to the stage of reconstructing your own BOOT file to add QPTR to it, you should note that the demos use SuperToolkit II routines, as included in the STK2 file. In addition, it is **vital** that SuperBASIC is flagged as "managed" - lines 110 to 160 of the BOOT file supplied contain the magic to do this, and may usefully be copied into your own BOOT file.

Two of the demonstrations are of no practical use, but serve to compare and contrast the way in which the facilities of the Pointer Environment are used from SuperBASIC and machine code. These are the files starting with the DEMO_ prefix.

The SuperBASIC program EDSPR demonstrates that it is possible to write pointer-driven programs without using the Window Manager parts of the Pointer Toolkit: you should also find it of use when designing sprites for use in machine code programs.

The SuperBASIC program PAINT demonstrates one or two areas of the Window Manager interface not used in the DEMO_ files, such as partial window operations and the graphics object drawing operations.

Both EDSPR and PAINT have been successfully compiled and run, using the Q_Liberator compiler: a compiled version of PAINT is supplied. If you re-compile PAINT, you may need to process the result with the STKINC utility to run it, as it uses the Window Manager. EDSPR may be compiled and run as is, because it does not use the Window Manager. See the **Utilities** chapter for more details.

The DEMO_ programs

The DEMO_ programs come in two versions: the version ending in _BAS is SuperBASIC, and may be LOADed and RUN in the normal way: the version ending in _BIN is machine code, and may be EXECuted from the SuperBASIC command line or the FILES menu of Qram.

Programs using the Window Manager go through a number of similar stages in their execution. They start by using the pointer information TRAP IOP.PINF to find the Window Manager vector. This may fail due to the absence of either the Pointer Interface or the Window Manager, it which case the program will probably have to give up. SuperBASIC programs find the Window Manager vector every time a Pointer Toolkit routine which requires it is used.

The next stage is to combine the **static** definition of the initial window with any **dynamic** information that may be required. The static definition is normally contained within the program itself, either in SuperBASIC DATA statements or in a Window Definition generated by the assembler using the macros provided or DC.x directives. The dynamic information may be generated before, during or after the conversion of the static definition to a "working definition", or any convenient combination of the three. For instance, the assembler version of the demo has a zero pointer to the "You have used the BEEP..." information in its static definition, and generates the complete string and resets the pointer in the working definition once the working definition has been mostly set up by the WM.SETUP routine.

Once a working definition has been generated, the window may be positioned and drawn - this is one operation in SuperBASIC, and two in machine code. User-defined code may be supplied to draw some non-standard parts of the window, for instance the musical staff in the demo program.

Now that the window is visible, input may be invited and acted upon. In machine code, the Window Manager can be made to do some of the hard work of deciding what the input consisted of and calling an appropriate action routine. In SuperBASIC this selection of an action routine has to be done by the SuperBASIC program itself.

The SuperBASIC version splits into three major units. Lines 1000 to 9999 contain the "action" part of the program, which sets up the data structures and changes them in response to user input. Lines 10000 to 19999 contain the "initialisation" part of the program, and also the data used to describe the window layout. Lines 20000 onwards contain "setup" routines usable in any SuperBASIC programs to set up window definitions.

The window you see is defined by the contents of the DATA statements in lines 12000 to 19999. It has four "loose menu items", defined in lines 12620 to 12720. It has two "information sub-windows", defined in lines 12840 to 12960: these contain two and one "information items" respectively, defined in lines 12730 to 12830. There are two "application sub-windows": the one defined in lines 13550 to 13590 has a short definition, implying that anything happening in that window needs to be dealt with by SuperBASIC. The second application sub-window is also a menu sub-window: the items it contains are defined in lines 12970 to 13140, their "spacing lists" in the X and Y directions in lines 13150 to 13320, and the "row list" splitting the linear item list into rows in 13330 to 13420. The "control definition" is set up in lines 13430 to 13500: this gives the two independently-scrollable sections. Three sprites are defined in lines 12200 to 12610: the first two are used as pointers, the last in the "move window" loose menu item. One set of standard colours and window attributes are used for all items and windows: these are defined in lines 12110 to 12190 and 12040 to 12100 respectively.

The definitions mentioned above are initialised by the setup functions and procedures at the end of the program. These expect DATA statements of the appropriate form, which are READ into arrays and the data structures set up by calling the corresponding MK_xxx function which is provided by the Pointer Toolkit. The result of this is passed back and may be used in subsequent DATA expressions: for instance, the main application window table, defined in lines 13520 to 13670, is then referred to in line 13740 by a DATA statement defining the contents of the window. The variable used here is main_awt: similarly the other variables main_sprite, main_lot and main_iwt have been defined earlier and are now referred to when setting up the main definition. The necessity to do this results in the "bottom-up" sequence of window definition in SuperBASIC, as opposed to the "top-down" sequence possible in assembly code, and which is probably more readable.

Once set up, the "action" part of the program then uses the Pointer Toolkit procedure DR_PULD to draw the window, and waits for user input by using the RD_PTR procedure. The result of the input is then acted upon. If the input occurred in the first application sub-window, then a note of the appropriate pitch and duration is played: clearly, any action could be taken here, depending on the application, so such sub-windows are very flexible but require more effort on the part of the programmer. The second sub-window, being a menu sub-window, is taken care of entirely by the Window Manager. Finally a hit on a loose menu item produces a returned sub-window number (swnum%) of -1, and radically different effects depending on which item is hit. Quit is quite simple, and just stops the program after discarding the window contents with a call to DR_UNST: ALL copies its resulting state to all items of the menu sub-window, and re-draws that sub-window: BEEP makes a simple beep, and changes and re-draws an information sub-window: and the move window item uses the supplied routine to move the window, and then resets its own state to available. The SELect ON construction here is peculiar to the SuperBASIC interface to the Window Manager. In the machine code version each item has its own "action routine" which is called as a result of the Window Manager having done its own equivalent of the SELect ON.

The machine code version in DEMO_BIN is made up of all the _ASM files, assembled and linked together as specified by the _LINK file. MENU_ASM and SPRITE_ASM define the data structures, INIT_ASM and SETUP_ASM convert them into a "working definition", DRAW_ASM provides a routine for drawing the staves in the first application sub-window, and ACTION_ASM provides all the routines used to act on user input. The principal difference in operation between this demonstration and the one written in SuperBASIC is that all actions are called directly from the Window Manager: the only action resulting from the initial call to WM.RPTR returning is after Quit has been hit to kill the job off.

The status area for the window is set up in the job's data area, which is pointed to by A6. A small amount of space is left below this to keep information which does not belong in the window's status area, such as the Window Manager vector. Note the use of dummy COMMON blocks to allocate the correct amount of space for the status area, the menu status block, the section control block and the variable information item. This method of making the Linker do all the hard work does take extra time when re-assembling and linking the program, but saves more by removing the need to check every file manually when a small change is made.

The EDSPR program

This simple program may be used to design sprites, blobs and patterns for use in other programs. It produces output that can be assembled directly to produce sprite definitions, or edited to produce blobs or patterns. You will also need to edit the output for use in SuperBASIC programs. To convert a sprite to a blob, you should remove the pattern and set the relative pointer to it to zero. Sprites to be used as patterns must be a multiple of 16 pixels wide, but require no modification. To generate a graphics object that is valid in more than one mode, separate definitions for each mode should be linked together by altering the relative pointer from its default zero value.

You are provided with a 5x5 initial grid, with each block representing one pixel of the sprite to be designed. The grid may be expanded and contracted in both directions by using the ADD and DELete ROW and COLumn items found in the Functions menu: the pointer sprite will change to show which function is currently active. Pixels may be set to any colour or transparent (black and white stipple) by selecting the required colour from the palette to the left of the main editing grid. The area above the palette signals the currently selected colour, and also acts as a "test area" so that you can see what the sprite you are designing looks like actual size and on varying backgrounds.

The Functions menu also allows you to set the origin of the sprite and to change display modes. After using either of these options, or selecting SET PIXel mode, or changing the colour to be used, the program is in SET PIXel mode and the pointer is the default arrow.

The Files menu gives you the options of saving or loading sprites designed with EDSPR: the filename is made up of the program default plus the given name plus the _ASM extension. The file format is suitable for assembling with the GST Macro Assembler, and also includes a human-readable copy of the definition: this is what is used when loading a sprite design.

The PAINT program

This program demonstrates pull-down windows, menus of sprites, patterns and blobs, and the various graphics object-drawing routines. It was developed progressively as a test-bed for the Pointer Toolkit, and is thus of fairly modular construction but of only moderate readability! To document it fully would double the size of this manual, so we suggest that you make a listing, and experiment with the program.

The area that you can work on defaults to a size of 640x640 pixels: you can move about this area as required, using the MOVE option from the Tools menu. If you convert an existing 512x256 screen image using the CVSCR utility supplied, and load this, you will not be able to move as far.

The Files menu allows you to save or load all the picture, or just the paste buffer: if you hit the filename then you can enter a different name to be used for the save or load operation. The selected operation will take place when you hit the OK item or do a "do" keystroke.

While drawing, a "hit" will usually start drawing whatever object has been selected in the Tools menu. Further "hits" will draw a line or flip between changing an ellipse's aspect ratio and its size/ inclination. A "do" will draw the object at its currently shown position, and an ESCape will abandon the current object. While in "doodle" mode, a "hit" will drop a blob or sprite, and a "do" will draw a line of blobs (but not sprites) from the last blob dropped to the current pointer position.

The spray option allows densities of between 5% and 95% when spraying patterns: note that with a combination of a small brush (blob) and a low density you may find that no pixels are sprayed.

Cut and paste work on rectangular areas **smaller** than the drawing area. If you wish to import an existing screen into the PAINT program, some work is necessary, as a whole 512x256 screen is too big to paste into the drawing area. The recommended method is as follows:
a) convert the screen image using the CVSCR utility
b) within PAINT, LOAD the converted image (ALL the picture, not the paste BUFFER)
c) use CUT and SAVE BUFFER to carve out the chunks you want from the screen image
d) re-start PAINT, or load a bigger picture to get back to a large picture area
e) use LOAD BUFFER and PASTE to put the chunks of the screen image where you want them

The Brush menu allows you to select various sizes and shapes of brush, which are combined with the selected paint when spraying or doodling. There are also two sprites (a flower and an apple) which are used directly, and not combined with the current paint. You may either hit the required brush and then the OK item, or "do" the required brush to select it.

The Paint menu provides access to various patterns with which to draw, and is used in a similar way to the Brush menu. The patterns at the top of the menu are all the possible checkerboard combinations of the colours available in the current mode, and may be used to draw objects of any sort. Lower down you will find various special patterns which can only be used when in the doodling and spraying modes: these become unavailable if the line, ellipse or block modes are selected. The first four or eight of these special patterns are stipples of the basic colours with "transparent" ink, which allow you to blacken, whiten, redden etc. parts of your drawing. There are also red gingham and brickwork patterns, two sizes of latticework with transparent holes in, and a green and transparent grass pattern.

The "Buffer" paint converts the contents of the paste buffer into paint, which may be used for doodling or spraying. The area saved in the paste buffer must be at least 16 pixels wide, this being the minimum allowable width for a pattern. When you select this option, the Paint menu is thrown away and you must position the pattern to line it up with the existing picture as required - this is similar to the "paste" option in the Tools menu. In this case, however, the buffer is only pasted in temporarily, and it is truncated in the horizontal direction, so that the width is a multiple of 16 pixels.

Concepts

This chapter is intended as a reference guide to the new concepts introduced by the Pointer Environment, as well as some old ones that have acquired a new significance within the Pointer Environment. Any terms used in the description of a concept that themselves have a description in this section are shown in `Courier` thus.

`Action routine`

Any item, be it a `loose menu item` or member of a `menu sub-window`, may be provided with an action routine. This will be called from within the `Window Manager` whenever a "hit" or "do" keystroke is made **and** the item is the current item **and** the item is not unavailable.

Within the Pointer Toolkit only pre-defined action routines are used, as it is not possible to call SuperBASIC routines from machine code.

`Application object list`

The `objects` in a `menu sub-window` are grouped into one or more application object lists (in SuperBASIC, one list only). The list is arranged into rows by the sub-window's `row list`.

An application object list defined from SuperBASIC also contains, at the start, the set of `item attributes` which are to be used with the objects defined in the list.

`Application spacing list`

The `objects` in a `menu sub-window` are arranged in a regular array of rows and columns: however, these rows and columns need not all be of the same height or width. A pair of spacing lists is required, one for the rows and one for the columns: there must be as many entries in the row spacing list as there are rows, and similarly for the columns. An entry in a spacing list defines (a) the size of the object itself, and (b) the spacing between the start of this object and the next: this should obviously be greater than the size of the object! If a row, say, consists of a number of objects of various heights, then the corresponding entry in the row spacing list should allow just enough space for the highest object.

`Application sub-window`

An application sub-window is an area of an application's window used for a particular purpose, for instance the drawing area in a drawing program or a file list in a file copying utility. Since the uses of such an area are very variable, the `Window Manager` requires the application program to provide routines to draw, read the pointer in, and modify such a sub-window.

A special case of an application sub-window is a `menu sub-window`, which can use some special routines provided by the Window Manager.

`Application sub-window list`

The application sub-window definitions used in any window will all take up different amounts of memory, depending on their complexity. It is therefore impossible to arrange them into a list in the same way as, say, `loose menu items`, which are all the same size. An application sub-window list of regular-sized entries is therefore used, which consists of a set of pointers to the sub-window definitions, followed by a pointer with a "silly" value (zero, in fact) which marks the end of the list.

`Blob`

A blob is a set of data somewhere in memory defining the shape of a graphics item, say a circle. Given a set of suitably defined `patterns`, one could use such a blob to draw red, green, white, brickwork, gingham etc. circles.

`Bottom window`

The bottom window is special, in that it is the window that will become `top` of the `pile` when "CTRL C" is pressed.

`Control definition`

A `menu sub-window` which is (or may be) divided into one or more `sections` requires a control definition to tell the `Window Manager` where each section starts in the sub-window, which is the first visible row or column in the section, and how many visible rows or columns there are in the section. This control definition will be modified by the sub-window's `control routine` as the user scrolls, pans, splits or joins the sections.

`Control routine`

When the pointer is within an `application sub-window` the action to be taken when a `pan/scroll bar` or `index item` is "hit" depends on the application itself. Therefore an application must supply a control routine for each sub-window which can be called by the `Window Manager` when either of those items is "hit". In the case of a `menu sub-window`, the Window Manager provides a standard control routine **WM.PANSC** which will prove useful in the majority of cases.

When using the Pointer Toolkit, only pre-defined control routines may be used as it is not possible to call SuperBASIC routines from machine code. If a menu sub-window is defined then the standard **WM.PANSC** routine is used, otherwise the **RD_PTR** call which entered the Window Manager returns.

Draw routine

All `application sub-windows` may be supplied with a draw routine, which is called by the `Window Manager` at the appropriate point when drawing the contents of a window for the first time. In the case of a `menu sub-window` this draw routine will frequently be a call to the Window Manager's own menu-drawing routine **WM.MDRAW**. Note that whether a draw routine is supplied or not, the Window Manager will always draw the sub-window's border and will clear it to the background colour, unless the "do not clear" flag is set. If a menu sub-window has `index items` and/or `sections` then a separate routine, **WM.INDEX**, must be called to draw the index items and/or pan/scroll bars etc..

When using the Pointer Toolkit, only pre-defined draw routines may be used as it is not possible to call SuperBASIC routines from the code. If the sub-window is a menu sub-window then the **WM.MDRAW** routine is used, otherwise no draw routine is used. If the sub-window has sections or index items these will also be drawn.

Hit area

A window's hit area covers the same area as the `outline`, but excluding the shadow. If a special pointer is defined for use within a window, it will appear only when the pointer is within the hit area of that window, and the window is `unlocked`.

Hit routine

When the pointer is within an `application sub-window` the action to be taken when the pointer is moved or a key is pressed depends on the application itself. Therefore an application must supply a hit routine for each sub-window which can be called by the `Window Manager` when either of the above events takes place. In the case of a `menu sub-window`, the Window Manager provides a standard hit routine **WM.MHIT** which will prove useful in the majority of cases.

When using the Pointer Toolkit, only pre-defined hit routines may be used as it is not possible to call SuperBASIC routines from machine code. If a menu sub-window is defined then the standard **WM.MHIT** routine is used, otherwise the **RD_PTR** call which entered the Window Manager returns.

Index items

A `menu sub-window` may have index items at the top and/or left-hand edge to show what is in a given column or row: for instance a spreadsheet might use the index items to show the row numbers and column letters. An index item list is of the same form as an `application object list`.

Information object list

An information object list defines the size, position, type and so on of each object that appears in an `information sub-window`. As with a `loose item list`, it is terminated with a special value: unlike loose objects, however, information items are fairly static and do not require `item numbers` or `action routines`.

Information sub-window list

The information that appears in a window may usefully be grouped into a number of information sub-windows, each with its own `window attributes` and `information object list`. These sub-windows are defined in a list of regularly spaced entries, terminated by a special value, called an information sub-window list.

Initial position

When a window is positioned by the `Window Manager`, the pointer will always appear at the position specified by the window origin in the `window definition`. When the call is made to the Window Manager to position the window, the application may specify how the pointer is to be moved to achieve this: an initial pointer position of (-1,-1) requests that the pointer be moved as little as possible, and a positive pair of co-ordinates requests that the pointer be moved as near as possible to that `absolute` position. The existing or given position may have to be modified if the window would fall outside the screen or its primary with the pointer at this position: this modification will be as small as possible.

Item

An item consists of one or more `objects`, all of which are in the same window or `menu sub-window`, and have the same `item number`. A "hit" on any one of the objects comprising a given item will cause all the objects in that item to be re-drawn with the new `status`.

Item attributes

An `item`, whether it is a `loose menu item` or contained in a `menu sub-window`, may have one of three `statuses`. When the item's status changes it will be re-drawn using a different set of item attributes, depending on its new status. For each of the three possible statuses, there are four attributes that may change: the background colour, on which the object is drawn: the text colour, used if there is any text in the item: the `blob` shape, used if part of the item is a `pattern`: and the pattern, used if part of the item is a blob. Thus selecting a pattern from a menu might change its blob from a circle to a tick, and change its background from white to green.

Item number

In each `loose` or `application object list`, the `objects` are given item numbers. These item numbers associate one or more objects with each flag in the `status block`, so that a "hit" on one object may affect the appearance of more than one object, but will only directly change the status of one item.

Note that the Pointer Toolkit restricts you to one object per item, as item numbers are assigned automatically by the various **MK_** routines.

Locked window

A window is locked while there is another `primary` window which (a) is above it in the `pile`, and (b) overlaps it. Most attempts to output to or input from a locked window will wait until the call `times out` or the window becomes unlocked: the exception is a pointer read (**RPTR**) with both bits 4 and 5 (in and out of window) set, which always returns immediately.

Loose menu item

It is frequently useful to have, within a window, a set of menu items that are permanently visible without having to pull down a `sub-menu` or `pan/scroll` a `menu sub-window`. Such items are often positioned in an irregular manner, as opposed to the regular row and column array of a menu sub-window. This need is catered for in the `Window Manager` by having a set of "loose" menu items which each have their own position and size, as well as the usual type, `action routine` etc.

Loose item list

All the `loose menu items` in a window are defined in one loose item list, containing data on their size, position, type and so on. The end of the list is marked by an entry of a special value which cannot occur anywhere else - experience shows that omitting this is a frequent cause of "mysterious" problems!

A loose item list defined from SuperBASIC also includes the set of `item attributes` to be used with the objects defined in the list.

Managed window

A window is said to be managed if its `outline` has been set by a call to **OUTLN**. Only if a window and its `primary` are managed will you be able to use it for pointer input or make use of `sub-windows`: there are also differences when `size checking` on an **OUTLN** or **WINDOW** call, and **CLOS**ing the window.

The BOOT program as supplied on the QPTR master medium sets SuperBASIC's outline: lines 110 to 160 must be copied to your own BOOT program if the Pointer Toolkit is to work correctly.

Menu sub-window

A menu sub-window is a special case of an `application sub-window`, consisting of `objects` arranged in a regular array of rows and columns. Similar or related objects will frequently be grouped together, for instance filenames in one column, file lengths in the next. Depending on the application single or multiple objects may be `selected`, and `pan/scroll bars` may be required to allow the user to view all the objects in the menu. The objects are defined in one or more `application object lists`, grouped into rows by the `row list`, with spacings between objects defined by `spacing lists`.

Outline

All windows, primary or secondary, have an outline. The primary window's outline is either set by an explicit call to **OUTLN**, or is maintained by the Pointer Interface to be just big enough to enclose the primary and all its secondaries: the first case is that of a managed window, the second is said to be unmanaged.

If the outline of a primary has been set, making it managed, you will get an "out of range" error if you try to set any of its secondaries outside it, either with **WINDOW** or with **OUTLN**. If you reduce the primary's outline with a further call to **OUTLN**, any secondaries whose area would then fall outside the new outline are reset so that their outline, hit and active areas are all the same as the primary's new hit area (i.e. as big as possible). Since their size has (probably) changed, any save area they may have is discarded.

Pan/Scroll bars

A menu sub-window may not be big enough to show all the objects in the menu: in this case the sub-window will usually provide pan and/or scroll bars to allow the user to move sideways or up and down through the objects respectively.

Pattern

A pattern is a set of data somewhere in memory that defines the colours with which a graphics item may be drawn: for instance, a brickwork pattern would consist of red blocks with white lines between them. Using suitable blobs, one could draw brickwork-coloured squares, triangles, circles, crescents and so on.

Pick

A window is said to be picked to the top of the pile if an action by the user or a program causes it to be transferred to the top. This transfer consists of a number of internal re-arrangements which you aren't very interested in (honest!), saving any primary that's about to be overlapped, restoring the contents of the picked window to the screen, and unlocking it. You can pick a window either from a program, using **PICK**, or by pointing to a visible bit of it with the pointer and hitting a key or mouse button, or typing "CTRL C". The last of these always picks the bottom window, the former two pick a specified window.

Pile

The set of primary windows present at any time may be thought of as resembling a pile of overlapping sheets of paper on a desk (the screen). There is a slight difference, in that two windows that do not overlap are always at different levels in the pile, even if they appear to be at the same level. A typical pile, viewed from the side (not possible!) might look like this:

```
-----           <- top window
        ---     <- unlocked, but not top
 -------        <- locked
-----------     <- bottom window, also locked
```

`Pointer`

If the mouse (if any) is moved or a read pointer call is made, a pointer of some sort will appear on the screen: this may take various forms depending on the state of the window to which it points.

`Pointer Environment`

The combination of the `Pointer Interface` and the `Window Manager` forms the complete Pointer Environment with both high and low level access for the programmer.

`Pointer Interface`

The Pointer Interface provides an extended and modified console driver, and forms the lower level of the `Pointer Environment`. For the programmer it provides some extra **TRAP #3**s (D0=$6C to $7F) to allow applications to read the pointer and so on

`Primary window`

Any job running in the QL may have a number of windows open at any one time: one of these, usually the first one used for I/O `not` the first one opened) is designated the job's primary window. This window's `outline` defines the area restored when the job is `picked` to the top of the `pile`. If the outline of a primary is explicitly set by **OUTLN** then the window becomes `managed`, and `size checking` is performed in a slightly different way. If the outline is not explicitly set, then the primary is `unmanaged`, and the outline can be "stretched" by opening new `secondaries` or moving existing ones.

`Scan order`

While the pointer is visible the Pointer Interface keeps track of which window contains it by scanning the `pile`. It is worth knowing how this is done, so that you know why the pointer is that boring little arrow and not the super-duper sprite you just designed! More seriously, if the sprite isn't what you expect then it's probably because the window you're using to read the pointer is `unmanaged`, or because its `primary` is unmanaged. Overleaf is a description of how the Pointer Interface decides which window contains the pointer, and thus which sprite to display.

```
FOR all primaries in current display mode, from top down
    IF pointer in this primary
        IF primary is managed
            FOR all its secondaries, in reverse order of use
                IF this secondary is managed
                    IF in this secondary
                        SET channel ID to secondary
                        SET no sub-window
                        SET secondary's pointer sprite
                        FOR all sub-windows of secondary
                            IF in sub-window
                                SET pointer sprite
                                SET sub-window number
                                EXIT sub-window
                            END IF
                        END FOR sub-window
                        EXIT to CHECK_POINTER_SPRITE
                    END IF
                END IF
            END FOR secondaries
            SET channel ID to primary
            SET no sub-window
            SET primary's pointer sprite
            FOR all sub-windows of primary
                IF in sub-window
                    SET pointer sprite
                    SET sub-window number
                    EXIT sub-window
                END IF
            END FOR sub-window
            EXIT to CHECK_POINTER_SPRITE
        ELSE
            FOR primary and all second., in reverse order of use
                IF in active area
                    SET channel ID
                    SET default sprite
                    SET no sub-window
                    EXIT to CHECK_POINTER_SPRITE
                END IF
            END FOR all windows
            SET no channel ID (-1)
            SET no sprite
            SET no sub-window
            EXIT to CHECK_POINTER_SPRITE
        END IF
    END IF
END FOR primaries
FOR all primaries in other mode
    IF in primary
        SET channel ID
        EXIT to CHECK_POINTER_SPRITE
    END IF
END FOR primaries
SET in no window
```

```
CHECK_POINTER_SPRITE:
    IF whole screen locked
        SET pointer sprite to "locked"
    ELSE
        IF window size/move/query
            SET pointer sprite to "size/move/query"
        ELSE
            IF channel in other mode
                SET pointer sprite to "other mode"
            ELSE
                IF channel busy or doing keyboard read
                    SET "busy" or "keyboard"
                END IF
            END IF
        END IF
    END IF
    FOR all versions of the pointer sprite
        IF this version is OK in this mode
            EXIT to SET_POINTER_RECORD
        END IF
    END FOR versions
    SET pointer sprite to "arrow"

SET_POINTER_RECORD:
    fill in pointer, channel ID, relative co-ordinates,
        sub-window number, window definition
    clear event vector and keystroke/keypress
```

`Secondary window`

A job may have more than one window open at once: the first used of these will be designated the `primary` window, all the rest will be secondaries. When a secondary's `outline` is set, that area of the screen is saved, so that when the outline is set again it may be restored (and the new area saved).

`Sections`

When a `menu sub-window` is too small to show all its `objects` at once, it may be found convenient to split the sub-window into one or more sections which can be `pan/scrolled` through the data: for instance, one would require two sections to look at the top and bottom of a spreadsheet simultaneously. The actions of panning, scrolling, splitting and joining the sections of a sub-window are taken care of by that sub-window's `control routine`.

`Setup`

The process of converting from a `window definition` to a `working definition` is the setup stage. In the machine code case it is accomplished by the `Window Manager` routine **WM.SETUP**. The SuperBASIC routines **DR_PPOS** and **DR_PULD** do a similar job on the definition set up by the **MK_WDEF** routine, and also call the appropriate positioning and window drawing routines.

Setup routine

When Window Manager sets up an application sub-window the data structures to be generated depend on the application itself. Therefore an application may supply a setup routine for each sub-window which can be called by the Window Manager during the setup stage. In the case of a menu sub-window, the Window Manager provides a standard setup routine **WM.SMENU** which will prove useful in the majority of cases.

When using the Pointer Toolkit, only pre-defined setup routines may be used as it is not possible to call SuperBASIC routines from machine code. If a menu sub-window is defined then the standard **WM.SMENU** routine is used, otherwise no setup routine is used.

Size checking

When a **WINDOW** or **OUTLN** call is made, the size required must be checked. If the window to be re-sized is unmanaged, then the check requires that the new size will fit on the screen: this is also the case when an **OUTLN** call is made for the primary window of a job. If the window to be resized is a managed secondary window, then it must fall within the hit area of its primary.

Sprite

A sprite, as used by the Pointer Interface, is a set of data somewhere in memory which defines both the shape and colour of a graphics object. Such an object may be (a) drawn within a window, or (b) used as a pointer: the familiar arrow, padlock, K and no-entry pointers are all sprites. This is somewhat different from the games programmer's definition of sprites, which move around of their own accord colliding with one another in a most unsettling manner.

Status

Any loose menu item or item in a menu sub-window has an associated status: this may be unavailable, available, or selected. This status is shown visually by changing the colours or shapes of the objects which comprise the item, and is recorded in a status block for use by the application. The colours and shapes used for each status are defined by the item attributes, each window having one set for its loose menu items (if any), and one set for the items in each menu sub-window.

Status block

A window will have a status block for its loose menu items, and one for each of its menu sub-windows. Each item has a one-byte flag, which will take different values depending on the item's status, at a position in the block corresponding to the item number. In addition, the flag may have its bottom bit set to indicate to the Window Manager that its status has changed and that the object should be re-drawn. Action routines are usually called with a pointer to a status block and an item number, so that the status of the item whose action routine has been called may be checked or modified.

`Sub-menu`

A sub-menu is very similar to an ordinary menu, but is contained in a `secondary window` that has been pulled down within its `primary`. Depending on the application a sub-menu might appear at a fixed point or close to the pointer. Usually sub-menus contain a set of associated options for which there isn't room in the main menu, or which would make it too cluttered. An example is the SORT sub-menu in QRAM.

`Sub-window`

Any `managed` window may have a list of sub-windows attached to it. When a **RPTR** call has been made, the Pointer Interface will `scan` through the `pile` of windows and set the pointer sprite to that defined for the sub-window containing the pointer (if any). If the pointer read returns then the co-ordinates of the pointer will be relative to the sub-window, making a programmer's life easier, we hope! The position of a sub-window is defined relative to its window, so it does not need to be reset if the window is re-defined.

A sub-window is only of relevance when doing a pointer read, to change the pointer sprite seen and the sub-window number and position returned: you cannot print to or clear sub-windows. If you wish to modify the area corresponding to a sub-window, you have to set a real window channel to that area - the `Window Manager` provides a routine to do this.

The `Window Manager` uses a sub-window for each application sub-window to determine whether the pointer is in an application sub-window or the main body of the window.

`Timing out`

It is possible to specify how long the QL should keep trying to do an I/O call for before giving up and returning a "not complete" error message - this is called timing out. All the Pointer Toolkit routines keep trying indefinitely, and thus never time out, but you may find that some other programs (or programming languages) use finite timeouts, and therefore fail to do some I/O sequences correctly if they try to do them while their windows are `locked`.

`Top window`

The top window in the `pile` is special in that it is always `unlocked` since nothing can overlap it, and it is the only window allowed to use the keyboard for input.

`Unlocked window`

A `primary` window is said to be unlocked if there is no primary above it in the `pile` which overlaps it. While a window is unlocked all attempts to output to it will succeed: attempts to do keyboard input from it will succeed if it is the `top` window. If a window is not unlocked then output will appear either when the window becomes unlocked, or not at all if the output call `times out` before the window becomes unlocked.

In addition, an `unlockable` window is always unlocked, regardless of any overlapping windows.

Unlockable window

A window may be made unlockable, in which case all output to it will appear instantly, regardless of whether there is an overlapping window or not: this is done by a special version of the **PICK** routine. This is what life was like before the Pointer Environment, jolly messy!

Unmanaged window

A window is said to be managed if no **OUTLN** call has been made to set its `outline`: in this case it is assumed that the job using the window is unaware of the existence of the Pointer Interface, and thus the effect of some I/O calls is slightly changed. For instance, any `sub-windows` are ignored during a pointer read. There are also some differences between unmanaged and `managed` windows when they are **CLOSE**d.

Unset

Once a `primary` or `pull-down` window has been set up and drawn, the definition will remain until the application removes it. The `Window Manager` provides a routine to do this which does all the operations required to make it safe to modify or remove the window's `working definition`. This routine is **WM.UNSET**.

The SuperBASIC unset routine not only calls the **WM.UNSET** vector, but converts all the absolute pointers in the data structures back into their relative forms.

Window definition

A window definition is an embryonic form of a full `working definition`, which is converted into the latter by a `setup` routine, frequently with the addition of some extra data: for instance, a file-copying program might generate its own `application object list` from the directory of a disc.

It may be convenient for applications written in different languages to have different window definition formats, and to provide their own setup routines.

Window Manager

The Window Manager is a set of utility routines which assist with the maintenance of windows, and which forms the higher level of the `Pointer Environment`. A number of routines are provided which translate and interpret data structures either set up by or contained within a program. Translation involves conversion of a `window definition` of the form recognised by the Window Manager to a `working definition`. Interpretation frequently takes the form of drawing or re-drawing part of a window.

Since the Window Manager is able to call various application-supplied routines, quite complicated effects can be achieved without the programmer having to write all the "boring bits".

`Working definition`

Whereas a `window definition` may take many forms, a working definition must always be of the same form. The first action of any application will usually be to translate the window definition into a working definition using its `setup` routines: subsequently the `Window Manager` will be able to work on the data structure produced, as it will now be in a standard form.

A typical window

1) A sprite type loose menu item, centred in the space allocated to it. This is the "move window" item, which should be present in most applications. It is "hit" by the standard key "CTRL F4" and specially treated within the Window Manager by generating a "move window" event.

2) Two text type loose menu items: these are also centred. The View item is specific to the application, and is "hit" by the V key. The HELP item should be present in most applications, and is therefore "hit" by a standard help key, F1, and specially treated within the Window Manager by generating a "help" event.

3) Two information objects, both of them text. The medium name and statistics object is in a window of its own, so that it can be re-drawn when necessary.

4) A menu sub-window. The objects in this are centred vertically, but left-justified horizontally. Both objects in a row, the filename and the file statistics, have the same item number, and thus share the same state; in this example, all files are available. Sub-windows like this do not have a separate channel of their own.

5) The current item in the primary window, which is also selected.

6) The current item in the pull-diwn window: this has not been selected, so it still shows in the available colours. Because this is a pull-down window, it has its own status area, so there is no confusion between this current item and the previous one.

7) The pointer: while this remains within the border showing that the DO item is current, a "hit" will select that item. As the pointer is moved, the Window Manager removes and replaces this border around whichever menu item the pointer is within.

8) A pull-down window. In contrast to the sub-window, this does have its own channel, which is opened when the window is pulled down and closed when it is discarded. This is an example of a secondary window, and thus lies entirely within its primary.

9) Scroll arrows: whenthe number of files is too large for the menu sub-window, the application increases the number of control sections from none to one, and calls the Window Manager routine provided to draw these bars. The Window Manager also provides the routine to scroll through the list of files.

10) Scroll bar: this allows easy scrolling through the whole range of files.

SuperBASIC

Keywords

The Keywords added by the Pointer Toolkit are split into two groups. The first deals with those routines which use only the Pointer Interface, the second with the routines that also require the Window Manager.

Pointer Interface routines

Optional parameters are included in square brackets, thus `[option]`, or curly brackets `{xpos,ypos}`.

Where this is of the form `[#ch,]` it shows that a channel number may be specified. If in any case it is not, the channel number defaults to #1 as usual.

Where an option occurs in square brackets that parameter may be specified or not as desired; where it occurs in curly brackets it may be specified zero, one or more times. For some optional parameters a table of the default values is given, with the effect the default value will have. If the default value is given as "none", then the procedure or function will do something different if the parameter is given, and there is no value that you can give this parameter that will have the same effect as omitting it. For instance, the **RPIXL** function just reads the colour of a pixel if no scan direction is given, but always scans if a scan direction is given, and no value of the scan direction parameter means "do not scan".

Separators are significant only where specified: otherwise you may choose any of the five possibilities (, ; ! \ TO), depending on which you find the most readable.

`HOT_STUFF str1$[,str2$]`

Option	Default	Meaning
str2$	""	stuff only str1$

This procedure puts a string into the HOTKEY buffer: `str1$` is put in the buffer first, immediately followed by `str2$` if present. The string in the HOTKEY buffer may be retrieved by typing "ALT SPACE" in any job, which will act as if the characters of the string had been typed instead of the "ALT SPACE".

This facility is available only if the HOTKEY job (supplied with QRAM) is active.

`LBLOB [#ch,][TO]{xpos,ypos{ TO xpos,ypos},}blob,pattern`

This procedure draws one or more lines of blobs. Apart from the optional channel number and the required `blob` and `pattern`, the parameters consist of co-ordinates preceded by TO or a comma: those preceded by a comma set the start point for drawing, those with a TO draw a line of blobs to the given end point and reset the start point to that end point. The start point is also set by the `WBLOB` procedure, and is kept in SuperBASIC's channel table between calls, so successive `LBLOB TO ...` calls will work as expected.

Co-ordinates are in pixels, blobs which would fall wholly or partly outside the window are not drawn.

`MKPAT addr,buffer`

Converts a screen save `buffer`, as created with the `PSAVE` function, into a pattern. The contents of the buffer are copied to the address given in `addr`, and there must be enough memory there for that copy of the buffer plus a graphics object header (18 bytes). The amount of memory required may be determined by a call to the **SPRSP** function, giving a `width` parameter the same as the x-size of the buffer, and a `height` parameter of `half` the buffer height.

The width will be truncated to the nearest 16 pixels, so the saved image in the buffer must be at least 16 pixels wide.

`MS_HOT [#ch,]hot$`

Set the string stuffed into the current keyboard queue when both mouse buttons are pressed simultaneously. The string `hot$` may be 0, 1 or 2 non-null characters to clear or set 1 or 2 characters to be stuffed. Because these characters appear in the keyboard queue before any further processing is done, they may be translated by the ALTKEY or HOTKEY processes to produce longer strings or start HOTKEY jobs.

You are advised to use this procedure only in BOOT files or utilities which invite the user to supply a mouse hotkey, e.g. system control panels.

`MS_SPD [#ch,]accel[,wakeup]`

Option	Default	Meaning
wakeup	none	don't change wakeup speed

This procedure modifies the response of the keyboard and mouse pointer movement. The `accel` parameter sets the acceleration of the mouse, making the pointer move quickly or sluggishly: it also affects the gradual speed increase when the pointer is driven from the keyboard.

The `wakeup` parameter applies only to the mouse, and sets the minimum speed that has to be reached before the (currently invisible) pointer appears: a high value will mean that an accidental nudge of the mouse while you are typing wll be less likely to cause the pointer to appear.

Both parameters are limited to a range of 0 to 9.

You are advised to use this procedure only in BOOT files or utilities which invite the user to change the mouse response, e.g. system control panels.

`OUTLN [#ch,]xsize,ysize,xorg,yorg[,xshad,yshad][,move]`

Option	Default	Meaning
xshad	0	no x shadow
yshad	0	no y shadow
move	0	discard window contents

The **OUTLN** procedure sets the "outline" of a window, and signals to the Pointer Interface that the window is "managed" - see the CONCEPTS section for explanations of these terms. Only managed windows with managed "primaries" may be used for pointer input: SuperBASIC's primary window is usually #0.

The three optional parameters default to zero, but you can specify the move key, the shadow widths, or both if you wish. The shadow will appear to the right or the bottom if `xshad` or `yshad` are positive. The `move` key will discard the current window contents if it is zero, or move them to the new position if it is set to 1 - you must keep the x and y sizes the same for this to work! If you set the outline of a secondary window, then the area underneath it will be saved, and restored when the outline is set again: this allows you to implement pull-down windows without having to do the saves and restores yourself.

`result =PICK( [#ch,]job-ID | key)`

This function picks the primary window belonging to a given job to the top of the "pile" on the screen, in the same way that the user can pick windows with "CTRL C" or by pointing and hitting with the pointer. The `job-ID` may be specified as two numbers, <job number>, <tag>, or as one composite number, <tag>*65536+<job number>: this is consistent with SuperToolkit II. Alternatively a `key` may be specified. If this is -1 then whichever job is at the bottom of the pile will be picked to the top: if it is -2, then the window specified will be marked "unlockable".

If the job specified doesn't have a window, or doesn't exist, then the result will be -2, the QDOS error code for "invalid job" - otherwise it will be zero, signalling success.

This function should be used with discrimination, unless you find it particularly amusing to have windows popping up at random.

Example:

```
1000 IF PICK(job_id)<0 THEN PRINT "Can't pick ";job_name$
```

`PREST [#ch,]buffer,bufxo,bufyo,xsize,ysize,winxo,winyo,keep`

This procedure restores a block, `xsize` by `ysize` pixels, from a `buffer` into a window. If `keep` is set to 1 then the buffer is kept, if 0 then it is discarded. The buffer may also be discarded by using the SuperToolkit II procedure `RECHP`.

```
result=PSAVE([#ch,]buffer,bufxo,bufyo,xsize,ysize,winxo,winyo
                [,bufxs,bufys])
```

Option	Default	Meaning
bufxs/ys	none	buffer is set up, address is valid

This function saves a block from a window into a `buffer` in memory: the block size and origin in the window are given in `xsize`, `ysize`, `winxo` and `winyo`, and the origin in the buffer of the block to be overwritten is given in `bufxo` and `bufyo`. A new buffer is set up by specifying a buffer size in terms of pixels, in `bufxs` and `bufys` - in this case the `result` returned is the address of the buffer. This function, and its complementary procedure `PREST`, allow the generation of graphics data over an area bigger than the screen of the QL. Note that when the buffer is set up, it is cleared to black, and that the only way of modifying it is with `PSAVE`.

Example:

```
100 REMark Save the top left 100x100 pixels of channel 1
110 REMark into the top left of a new 512x768 buffer.
120 :
130 buffer=PSAVE(0;0,0;100,100,0,0;512,768)
140 :
150 REMark Now draw a big circle, and save that 100
160 REMark pixels across the buffer.
170 :
180 FILL 1:CIRCLE 50,50,30
190 d=PSAVE(buffer;100,0;250,200,0,0)
200 :
210 REMark Now restore some of what we saved before,
220 REMark and some of the circle, at the bottom
230 REMark right of the window.
240 :
250 PREST buffer;50,50;100,100,150,100;1
```

`result=RMODE`

This function reads the current display mode, returning 4 for 4-colour mode and 8 for 8-colour. This function can and should be used to avoid doing MODE calls to set the display mode to the one the QL is in already!

`result=RPIXL([#ch,]xstart,ystart[,direction[,colour[,same]]])`

Option	Default	Meaning
direction	none	no scan
colour	-1	start pixel is reference colour
same	0	scan to different colour pixel

The simple form of this function returns the colour (0-7) of the pixel at `xstart,ystart`.

If a `direction` is given, the function scans horizontally or vertically from the start point (0=up, 1=down, 2=left, 3=right) until a pixel of a different colour is found, and returns the co-ordinate of that pixel. Since the scan is horizontal or vertical the other co-ordinate remains constant.

If a `colour` is given then the scan looks for a pixel of a different colour to that given: if no colour is given, or the given colour is specified as -1, then the colour of the start pixel is used.

If the `same` flag is given, a value of 1 scans for a pixel of the same colour as the reference: a value of 0 scans for a different colour.

If the scan reaches the edge of the window without finding a pixel of the required colour then the co-ordinate returned is -1.

`RPTR [#ch,]xabs%,yabs%,term%,swnum%,xrel%,yrel%,bt$`

Read the pointer position in the given window, which must be "managed" - see the description of **OUTLN** and the Concepts chapter for more details. The procedure will return under various circumstances, depending on the value of `term%`:

Bit set	returns if...
0	...a keyboard key or mouse button is pressed.
1	...a keyboard key or mouse button is, or continues to be, pressed. Normal auto-repeat speeds apply.
2	...a keyboard key or mouse button is released.
3	...the pointer is moved from the given absolute co-ordinates
4	...the pointer is, or moves, out of the window
5	...the pointer is in, or moves into, the window

Bit 6 is reserved - do not set it! Bit 7 selects a special mode, in which all other jobs' windows are locked, and a special sprite appears depending on the values of bits 0 and 1:

Bit set	sprite shown
1	"window change size"
0	"window move", unless bit 1 is set
neither	"empty window"

Bits 2 to 6 should all be clear when bit 7 is set. The co-ordinates returned are always absolute, rather than relative to the origin of the window used to make the call.

Apart from the above "window request" mode, the co-ordinates returned in `xrel%` and `yrel%` will be relative to the origin of a window or "sub-window". If the pointer was in a sub-window then the value of `swnum%` will be 0 or greater, otherwise it will be -1. See the description of **SWDEF** to find out about sub-windows.

If a "return on move" is requested then `xabs%` and `yabs%` are used as the reference point - when the pointer is moved from this position then the call will return. These variables are normally set up at the start of the program, and subsequently updated only via the **RPTR** call.

The value of `bt$` is a single character string. If a button or key press happened, the character will correspond to the key `except` for the following "event keystrokes":

Key	CHR$	Event
None	0	no key pressed
SPACE/left mouse	1	hit
ENTER/right mouse	2	do
ESC	3	cancel
F1	4	help
CTRL F4	5	move window
CTRL F3	6	change size

The values of `xabs%`, `yabs%`, `term%` and `swnum%` should be set before calling this procedure, as they are used to determine when the call will return. On return all the parameters will be set to the appropriate values. `Note that if you call the procedure with the wrong type of variable (float instead of integer, for instance) then you'll get some very odd results - use only integers for the first six parameters, and a string for the last.`

As this routine returns values through the parameter list, it is not compatible with the Super/Turbocharge compilers.

Examples:

```
1000 xa%=0 : ya%=0 : kystk=1 : swnum%=-1
1010 OUTLN 256,202,256,0;1 : BORDER 1,255
1020 REPeat l
1030   rt%=kystk : REMark Return when a key is hit
1040   RPTR xa%,ya%,rt%,swnum%,x%,y%,bt$
1050   PRINT #2;x%,y%,CODE(bt$)
1060 END REPeat l
1000 REMark Set up current absolute position
1010 REMark and sub-window number:
1020 REMark OUTWN+INWIN returns instantly
1030 :
1040 OUTLN 256,202,256,0;1 : BORDER 1,255
1050 outwn=16:inwin=32:rt%=outwn+inwin
1060 xa%=0:ya%=0:swnum%=-1
1070 RPTR xa%,ya%,rt%,swnum%,x%,y%,bt$
```

`result=SPRSP(width,height)`

This function calculates the memory space required to store the definition of a sprite of the given `width` and `height`, both in 4-colour mode pixels. This is particularly useful for loading multiple sprites into one piece of memory by calculating the space for each and then allocating it all at once: this reduces overheads and heap fragmentation.

`SPHDR addr,xsize,ysize,xorg,yorg,md[,next]`
`SPHDR addr,next`

This procedure sets up a sprite header to be filled by the **SPLIN** procedure: there must be enough room at the address given in `addr` for a sprite of the required size.

The sprite may be linked to the `next` one in a list, either as an option on the long form of the procedure, or using the short form. Such linked sprites may be defined for use in different modes, as specified by `md`. When used as a pointer or drawn using **WBLOB** or **WSPRT**, the list will be searched for a definition suitable for use in the current mode.

Example:

```
1000 REMark Set up a pointer for #1, shape depending
1010 REMark on mode.
1100 :
1110 REMark First the pointer that appears
1120 REMark in mode 4
1130 :
1140 spr4=ALCHP(SPRSP(9,9))
1150 SPHDR spr4;9,9,5,5;4
1160 linum%=0
1170 SPLIN spr4,linum%,' ww '
1180 SPLIN spr4,linum%,' waw '
1190 SPLIN spr4,linum%,' waaw '
1200 SPLIN spr4,linum%,' wawaw '
1210 SPLIN spr4,linum%,' wawwawww'
1220 SPLIN spr4,linum%,'waaaaaaaw'
1230 SPLIN spr4,linum%,'wwwwwawww'
1240 SPLIN spr4,linum%,' waw '
1250 SPLIN spr4,linum%,' www '
1300 :
1310 REMark Now set up a sprite to appear in mode 8
1320 REMark and link it to the mode 4 sprite.
1330 :
1340 spr8=ALCHP(SPRSP(20,10))
1350 SPHDR spr8;20,10,10,5;8;spr4
1360 linum%=0
1370 SPLIN spr8,linum%,' wwwwww '
1380 SPLIN spr8,linum%,' wwaaaaww '
1390 SPLIN spr8,linum%,' wawwwwaw '
1400 SPLIN spr8,linum%,' wawwwwaw '
1410 SPLIN spr8,linum%,' wwaaaaww '
1420 SPLIN spr8,linum%,'wwawwwwaww'
1430 SPLIN spr8,linum%,'waww wwaw'
1440 SPLIN spr8,linum%,'wawwwwwwaw'
1450 SPLIN spr8,linum%,'wwaaaaaaww'
1460 SPLIN spr8,linum%,' wwwwwwww '
1500 :
1510 REMark Attach it to #1
1520 :
1530 OUTLN 256,202,256,0;1 : BORDER 1,255
1540 SWDEF : SWDEF -1;252,200,0,0;spr8
1600 :
1610 REMark Read the pointer: the sprite you see
1620 REMark depends on the display mode
1630 :
1640 ax%=0:ay%=0:swnum%=0:rt=1
1650 REPeat l
1660   rt%=rt
1670   RPTR ax%,ay%,rt%,swnum%,xr%,yr%,bt$
1680 END REPeat l
```

`SPLIN addr,linum%,patt$`

Fill in one line of pixels in a sprite. The header must have been set up previously using the **SPHDR** procedure. The line to set is given by `linum%`, with line 0 being the top: if the line number is too big you will get an "out of range" error. The pixel colours are specified in `patt$`, as for **SPSET**. If the line number parameter is a variable then it will be incremented after this call, so successive calls to **SPLIN** will set successive lines of a sprite: this feature will not work with the Super/Turbocharge compilers.

`SPRAY xorg,yorg,blob,pattern,pixels`

This procedure works in a similar way to **WBLOB**, but instead of writing the whole blob it writes only a few pixels from it: the number of pixels written is given by the `pixels` parameter. These are chosen "at random" from the blob to give a spray effect. Somewhere between 5% and 20% of the total number of pixels in the blob usually gives a good result. If you spray several times with the same parameters the blob will gradually fill in, but there is no guarantee that it will ever do so completely, even if the `pixels` parameter is the same as the total number of pixels in the blob.

`SPSET addr,xorg,yorg,md,shape$(ysize,xsize)`

This procedure sets up the data for a sprite, in a suitable form for a particular QL mode as specified in `md`. The size is given by the dimensions of the string array `shape$` defining the sprite: for convenience you may pass an array slice. The sprite's origin must also be given in `xorg,yorg`.

The colour of each pixel of the sprite is specified by a character in the string array, the top left pixel being specified by `shape$(0,1)`, the top right by `shape$(0,xsize)`, the bottom right by `shape$(ysize-1,xsize)` and so on. Note that the rows run from 0 to n-1, as in other arrays, but the columns from 1 to n as for strings.

The colour characters permitted are "`aurmgcyw `", standing for pixels that are blAck, blUe, Red, Magenta, Green, Cyan, Yellow, White and transparent (space).

Example:

```
100 DIM shape$(10,10):RESTORE 180
110 READ xsize,ysize,xorg,yorg,md
120 FOR i=0 TO ysize-1:READ shape$(i)
130 addr=ALCHP(SPRSP(xsize,ysize))
140 SPSET addr,xorg,yorg,md,shape$(0 TO ysize-1,1 TO xsize)
150 REMark Concentric rings with a hole in the centre
160 DATA 7,7,3,3,4
170 DATA '  www  '
180 DATA ' wgggw '
190 DATA 'wgrrrgw'
200 DATA 'wgr rgw'
210 DATA 'wgrrrgw'
220 DATA ' wgggw '
230 DATA '  www  '
```

`SWDEF [#ch,][swnum[,xsize,ysize,xorg,yorg[,sprite]]]`

Option	Default	Meaning
swnum	none	clear all sub-window definitions
xsize..yorg	none	clear given sub-window definition
sprite	none	use default sprite

This procedure sets or clears a sub-window definition. If no parameter is given then the sub-window list for the window is removed entirely: if just the sub-window number `swnum` is given, then that sub-window definition is removed: and if a definition is given, then that sub-window is (re-)defined. Optionally the address of a sprite definition, `sprite`, may be appended, in which case the pointer will change to that sprite when it is within the sub-window.

The origin given is relative to the "hit area" of the window, which must be "managed". The sub-window definition for the main part of the window may be set by specifying a sub-window number of -1: the origin in this case is absolute. Removing the sub-window definition of the main part of the window will reset the sprite to the default, and the area to the hit area.

Note that if you wish to use N sub-windows, you must specify all sub-windows from 0 through N-1, and in addition the window's primary must be managed (must have had its outline set with **OUTLN**). Sub-windows are checked starting at sub-window 0, up to the first unset one, and then the main part. To avoid fragmenting the heap more than is necessary, you are advised to define the highest numbered sub-window first.

Example:

```
100 REMark Remove all current definitions, and put
110 REMark one sub-window across the top of #1, and one
120 REMark down the side with a special "hand" sprite.
130 :
140 SWDEF
150 SWDEF 1;250,20,0,0
160 SWDEF 0;40,100,0,21;hand
```

`WBLOB [#ch,]x,y,blob,pattern`

This procedure writes the `blob` into the given channel, using the `pattern`, at the given co-ordinates `x,y`. These co-ordinates are also used to update the default start point for the `LBLOB` procedure. The blob specifies the shape of what appears, the pattern the colour, so you would need one blob and three patterns to draw red, yellow and blue flowers. In this version the blob is not drawn if it overlaps the edge of the window, or falls outside it. The blob and pattern are pointers to items of the appropriate sort - probably loaded into the heap with an **ALCHP** followed by an **LBYTES**, or set up from SuperBASIC by calls to **SPSET**, **SPHDR** or **SPLIN**. In early versions of the Pointer Interface no check is made on the blob and pattern, and the blob drawing routine can be crashed quite easily by duff data: you have been warned!

Note that any sprite may be used as a blob, and any sprite whose width is a multiple of 16 may be used as a pattern.

`WSPRT [#ch,]x,y,sprite`

This procedure is very similar to **WBLOB**, except that the `sprite` data structure defines both shape and colour information, so you would need three complete sprite definitions to draw red, yellow and blue flowers - but they could all be different shapes. The same comments apply with regard to drawing outside the window and using valid sprite definitions.

A feature of versions 1.13 onward of the Pointer Interface is that the built-in sprite definitions may be written if a small integer is specified rather than an address:

Value of sprite	Sprite drawn
0	Pointer arrow
1	Lock
2	Window request
3	4 or 8
4	Keyboard
5	No Entry
6	Window Move
7	Window Resize

`WREST [#ch]`

This procedure restores the saved area of the given window. The save area is lost. This procedure should be used only when the window size has not changed.

Window Manager routines

The following SuperBASIC routines form an interface to the Window Manager. They are in four groups, definition routines, drawing routines, access routines and change routine.

The majority of these routines make use of arrays to pass long parameter lists to them with the minimum of typing: unfortunately routines which use array parameters are not compatible with the Super/Turbocharge compilers, and you will be unable to compile programs which use them with these compilers.

The amount of stack used by the Window Manager on some calls is greater than that permitted for machine code SuperBASIC procedures or functions: this has not caused us any problems with the interpreter, but has resulted in crashes with program compiled with Q_Liberator, versions up to 3.12. Versions from 3.21 onwards allow more stack, and do not suffer from this problem. If you have Q_Liberator v.312 or earlier then compiled programs may be used if processed with the STKINC utility: see the Utilities chapter for more details.

Definition routines

These set up parts of a window working definition, given parts of the window definition in one or more arrays. Each is a function which returns the address of the data structure set up: these addresses are then used as parameters in further calls to the Window Manager routines.

```
lilst=MK_LIL(attr(3,3),size%(n,1),org%(n,1),jus%(n,1),sk$,
                type%(n),strg$(p,m),pspr(q),pblb(r),ppat(s))
```

Make a loose item list, complete with attributes.

There are n+1 items in the list. Each item has its own size, origin and justification in the appropriate arrays, the x-attribute being in `arr%(i,0)` and the y in `arr%(i,1)`. The justification specifies whether the object is to be left/top justified (positive values), right/bottom justified (negative values) or centred (zero). Non-zero values give the distance in pixels from the appropriate edge of the area defined by the size and origin of the item.

The `type%` array specifies not only the type of each item in the bottom byte of each word, but also the action to be taken on "hitting" each item: if the top byte is zero, then no further action is taken, if non_zero then the RD_PTR call returns: if +1, the item's status is reset to available before returning, if -1 no change is made to the status. To set the top byte to +1 or -1, add +256 or -256 to the item type. The value of the bottom byte may be 0, 2, 4 or 6 for string, sprite, blob or pattern items: up to p+1 elements of `type%` may have a bottom byte of 0, q+1 of 2, and so on. When an element specifies that an object should be of a given type, then the next object is taken from the appropriate array. Thus if `type%` contains the values 0, 2, 2, 4, 2 and 6 the objects will come from `strg$(0)`, `pspr(0)`, `pspr (1)`, `pblb(0)`, `pspr(2)` and `ppat(0)`.

If an item is null (a zero length string or zero pointer) then it is assumed that the item is absent: such items may be reset later with the CH_ITEM procedure.

```
iolst=MK_IOL(size%(n,1),org%(n,1),imod(n),type%(n),
             strg$(p,m),pspr(q),pblb(r),ppat(s))
```

Make an information object list. `size%, org%, type%` and the object arrays are the same as for a loose item list. There are no justification or select key arrays, and the top byte of `type%` is ignored. Objects are taken in turn from the `strg$, pspr, pblb` and `ppat` arrays, depending on the contents of `type%`, as for the MK_LIL function.

If an information object is a piece of text, or a blob or pattern, additional information is required to draw it: in the case of text, you need to specify how big it is and what colour: a blob needs to be drawn using a pattern: and a pattern needs to drawn using a blob. The `imod` array specifies this additional information: if item N is a blob or pattwern then `imod(N)` contains a pointer to a pattern or blob to combine with it. If item N is text then the colour and size are combined using the magic formula

<ink>*65536+<csize_x>*256+<csize_y>

So a lagre red piece of text would have an attribute of 2*65536+3*256+1, or 131841.

```
aolst=MK_AOLST(iattr(3,3),jus%(n,1),sk$,type%(n),
          strg$(p,m),pspr(q),pblb(r),ppat(s))
```

Make an application sub-window object list. Very similar to a loose manu item list, except that there are no size or origin attributes. If the bottom byte of `type(0)` is odd then the list is assumed to be of index items, and the item number is set to $FFFF and the action routine to 0. In this case the attributes specified are those to be used for the index items (see below).

```
cdef=MK_CDEF(maxsed%,arrc%,barc%,secc%)
```

Make a control definition list: this specifies the maximum number of sections into which the sub-window can be split, and the colours for the arrows (`arrc%`), bars (`barc%`) and bar sections (`secc%`). After this area is reserved enough space for a section control block with up to `maxsec%` sections.

```
aslst=MK_ASL(size%(n,1)[,isiz%,ispc%])
```

Make an application sub-window spacing list. `size%(i,0)` gives the hit size, `size%(i,1)` the spacing. The sizes and spacings for the index bars may also be set. Two spacing lists are required for each sub-window, one for each axis.

```
rwlst=MK_RWL(aolst,se%(n,1))
```

Make an application sub-window row list. There are n nows, the i'th starting with item `se%(i,0)` and ending just before item `se%(i,1)`. The object list is at `aolst`, as returned by a call to the MK_AOL function.

```
apw(n)=MK_APPW(wdef%(3),wattr%(3),ptr,sk$,
             [x_cdef,y_cdef,
              x_off%,y_off%,
              x_aslst,y_aslst,
              x_aolst,y_aolst,
              rwlst])
```

Make an application sub-window definition. If a menu sub-window is required, all parameters must be given, although the pointers to the control definitions and index list definitions (`x_cdef, y_cdef, x_aolst,` and `y_aolst`) may be zero: the spacing list and row-list pointers (`x_aslst, y_aslst` and `rwlst`) are required. The pointer and select key (`ptr` and `sk$`) may be zero and the null string if these are not required. The number of items in a spacing list, index item list and row/column must be consistent.

As a special case a sub-window may be defined with only the first four parameters, in which case a special hit routine is used which results in a RD_PTR call returning every time the pointer is moved or a key is hit in that sub-window.

```
iwlst=MK_IWL(wdef%(n,3),wattr%(n,3),iolst(n))
```

Make an information sub-window list. Each information sub-window has a soze and position in `wdef%(i)`, attributes given by `wattr%(i)`: the pointer to the object list in `iolst(i)` should be the result of a call to the MK_IOL function.

```
awlst=MK_AWL(apw(n))
```

Make an application sub-window list. The array of pointers, to sub-window definitions generated by the MK_APPW function, is copied and terminated with a long word of zero.

```
wdef=MK_WDEF(wdef%(3),wattr%(3),ptr,lilst,iwlst,awlst)
```

Make a complete window definition. Any of the last four pointers may be zero. If non-zero, `ptr` should point to a sprite definition to be used as the pointer in the window, while `lilst`, `iwlst` and `awlst` are the results of calls to the MK_LIL, MK_IWL and MK_AWL functions.

The window position specified in the `wdef%` array parameter is NOT the absolute position at which the window will be drawn, but the initial position of the pointer within the window when it is drawn.

Drawing routines

These procedures set up and draw a window from definitions generated by the definition functions above, and allow an application to re-draw part of a window. Routines are also provided to position a given window channel "over" part of a window, so that embellishments may be added and so forth. This is particularly useful in the case of pull-down windows, whose channels are inaccessible to the SuperBASIC program.

The `wdef` parameters required by all these routines is the result of a call to the MK_WDEF function.

```
DR_PPOS [#ch,]wdef,xpos%,ypos%[,lflag%(n)]
        {,aflag%(p,q)[,ctx%(maxsec%,2)][,cty%(maxsec%,2)]}
```

Position a primary window, or ...

```
DR_PULD wdef,xpos%,ypos%[,lflag%(n)]
        {,aflag%(p,q)[,ctx%(maxsec%,2)][,cty%(maxsec%,2)]}
```

... pull down a window. After a window has been positioned or pulled down then it is drawn. A flag array is passed for the loose items (`lflag%`) and a flag array (`aflag%`) and zero, one or two control definition arrays (`ctx%` and `cty%`) for each menu sub-window, and the items drawn with the given statuses. The channel for a pull-down window is opened, a primary window's channel must already be open.

When the window appears, the pointer will always be set to the initial pointer position within the window as specified when the window definition was set up. If the positioning parameters `xpos%` and `ypos%` are set to -1, then the pointer will be moved as little as possible (often no distance) to accomplish this. If, however, `xpos%` and `ypos%` are set to some other value, then the pointer will be set as close to that absolute position as possible before the window is pulled down.

A window is always positioned so that its X origin is a multiple of two: this ensures that any stipples used in the window remain "in phase" at all times.

```
DR_LDRW wdef,lflag%(n)
```

The flag array `lflag%(n)` is copied into the loose items status block, and the loose items are then re-drawn. If no change bit is set in any flag, then all items are re-drawn, otherwise only changed items are re-drawn.

```
DR_ADRW wdef,aswnum%,aflag%(p,q)
        [,ctx%(maxsec%,2)][,cty%(maxsec%,2)]
```

The flag array `aflag%` is copied into the status block of the application sub-window referred to by the `aswnum%` parameter, the control definition arrays `ctx%` and `cty%` (if any) copied into the control block, and the menu sub-window is re-drawn, using the same rules as for loose menu items. If element (0,1) of a control definition is non-zero, then the whole sub-window is re-drawn, regardless of the item status changes.

`DR_IDRW wdef,infwm`

This procedure re-draws any of the first 32 information sub-windows in the window given by `wdef`. The `infwm` is interpreted as a bit map of the windows to be re-drawn, with a clear bit corresponding to a window to be re-drawn. Thus a value of -2=$FFFFFFFE will re-draw information sub-window 0 only, -6=$FFFFFFFA will re-draw windows 0 and 2, and so on.

`DR_AWDF [#ch,]wdef,swnum%`

Set a channel to cover the same screen area as the given application sub-window.

`DR_IWDF [#ch,]wdef,iwnum%`

Set a channel to cover the same screen area as the given information sub-window.

`DR_LWDF [#ch,]wdef,item%`

Set a channel to cover the same screen area as the given loose item.

`DR_UNST wdef`

Unset a window definition. A window that was pulled down is removed and its channel closed.

Access routines

```
RD_PTR wdef,item%,swnum%,event%,xrel%,yrel%
       [,lflag%]{,aflag%[,ctx%][,cty%]}
```

Read the pointer via the Window Manager: the call returns when a window event occurs, or a return item is "hit". In addition to the returned parameters, the item statuses are copied back into the appropriate arrays. The item number and sub-window number of the last item hit are returned in `item%` and `swnum%`, and the event causing the return in `event%`: this may be 128 for a hit on an item causing an automatic return, or one of the following values, caused by an "event generating" keystroke:

Event name	Keystroke	`event%` value
Do	ENTER	1
Cancel	ESC	2
Help	F1	4
Move	CTRL F4	8
Resize	CTRL F3	16
Sleep	CTRL F1	32
Wake	CTRL F2	64

The flag and control arrays are copied into the relevant status areas on entry. If any of the statuses have changed (signalled by odd flag values), the changed items only are re-drawn: if a control definition has changed, then the whole of that menu is re-drawn. This frequently avoids the need for explicit re-draw calls.

The returned pointer co-ordinates `xrel%` and `yrel%` are relative to the top left corner of the sub-window.

If the pointer is in an application sub-window which is not a menu sub-window, then the call will return whenever a key is pressed or the pointer is moved. Since such a sub-window has no items in it, the keystroke and keypress are returned in the high and low byte of `item%`. Note that moving the pointer via the cursor keys produces keystrokes, whereas moving it with a mouse does not.

Change routines

`CH_ITEM wdef,swnum%,item%,type%,selkey$,value`

Change the given item in the given sub-window to the new value, type and select key, given in `value, type%` and `selkey$.` The type of the value may be string or floating point, depending on the type of the item. Special values are:

`swnum%`	-1 for loose item, -n for information item in information window n-2 (n>1): thus -2 to alter information window 0, -3 to alter window 1 etc...
`type%`	-1 for no change
`selkey$`	"" for no change (ignored in information window) chr$(0) for no select key

`CH_PTR wdef,swnum%,newptr`

Change the pointer sprite for a sub-window. If the sub-window number given in `swnum%` is -1 then the main window's sprite is re-defined. If the address of the pointer sprite, given in `newptr`, is zero then the default sprite is used. This is the same as the main window's sprite for a sub-window, and is the arrow sprite for the main window.

`CH_WIN wdef[,xdsiz%,ydsiz%]`

Change a window's size or position. If only the `wdef` parameter is given then the window's position is changed, otherwise the size change required is returned in `xdsiz%` and `ydsiz%`. Since the window's layout will probably change fairly drastically when the size changes, it is up to the programmer to decide the effect of the result returned. Note that changing the position of a primary window does not change the positions of its secondaries: any sub-windows of the moved window do move with it, as their positions are defined relative to it.

As for the initial positioning of a window, the X origin will always be a multiple of four, and the Y origin a multiple of two, to keep stipples "in phase".

Array parameters

Some forms of array parameters are used in many of the above routines: their dimension and contents are defined below.

Array name	Contents
wattr%(3)	Window Attributes
Element	Data
0	shadow depth
1	border width
2	border colour
3	paper colour

iattr(3,3)	Item Attributes
Element	Data
0,0	current item border width
0,1	current item border colour
0,2/3	spare,0
1,0	unavailable item background colour
1,1	unavailable item ink colour
1,2	unavailable item pointer to blob
1,3	unavailable item pointer to pattern
2,0 TO 3	available item
3,0 TO 3	selected item

Note that only the current/unavailable attributes are used for index items, but that the available and selected attributes must still be set. If a separate attribute array is used for index items, rows 2 and 3 may be left as 0.

wdef%(3)	(Sub-)window size/position definition
Element	Data
0	window x size
1	window y size
2	window x origin (Initial pointer position, when
3	window y origin used in main window def.)

The flag arrays determine the status of each item in a window: if an item's status is changed by the program, a re-draw may be requested by adding 1 to the required status. The re-draw will take place either when specifically requested by a call to one of the re-draw routines, or automatically on a call to RD_PTR.

lflag%(n) and	Loose item flag array and
aflag%(n,m)	menu item flag array
Flag value	Item status
0	available
16	unavailable
128	selected

cta%(maxsc%,2)	Control definition array
Element	Data
0,0	current number of control sections
0,1	<>0 if the control definition is changed
i,0	start pixel position
i,1	start column/row
i,2	number of columns/rows

Index of keywords

The keywords are summarised in alphabetical order, together with an indication of what action they perform. Those marked PTR require the Pointer Interface, WMAN need the Window Manager in addition: unmarked ones are independent of either. Those marked P are procedures, F are functions: an A signifies that the routine uses array parameters, and an R that it returns results through its parameter list. Having either of the latter properties makes a program using the routine uncompilable with the Super/Turbocharge compilers.

CH_ITEM	WMAN	P	change a menu item
CH_PTR	WMAN	P	change a menu or sub-window's pointer sprite
CH_WIN	WMAN	PR	change a window's position or size
DR_ADRW	WMAN	P A	re-draw an application sub-window
DR_AWDF	WMAN	P	put window over application sub-window
DR_IDRW	WMAN	P A	re-draw an information sub-window
DR_IWDF	WMAN	P	put window over information sub-window
DR_LDRW	WMAN	P A	re-draw loose menu item(s)
DR_LWDF	WMAN	P	put window over loose item
DR_PPOS	WMAN	P A	position and draw a primary window
DR_PULD	WMAN	P A	position and draw a pull-down window
DR_UNST	WMAN	P	unset and remove a window
HOT_STUFF		P	put string(s) into the hotkey buffer
LBLOB	PTR	P	draw line(s) of blobs
MKPAT		P	turn a part-window save area into a pattern
MK_AOL		F A	make an application sub-window object list
MK_APPW		F A	make an application sub-window definition
MK_ASL		F A	make an application sub-window spacing list
MK_AWL		F A	make a list of application sub-windows
MK_CDEF		F	make a control definition
MK_IOL		F A	make an information object list
MK_IWL		F A	make an information window list
MK_LIL		F A	make a loose item list
MK_RWL		F A	make an application sub-window row list
MK_WDEF		F A	make a window definition
MS_HOT	PTR	P	set mouse-hotkey string
MS_SPD	PTR	P	set mouse speed parameters
OUTLN	PTR	P	set a window's outline and shadow
PICK	PTR	F	pick/unlock a job
PREST	PTR	P	part window restore from buffer
PSAVE	PTR	F	part window save to buffer
RD_PTR	WMAN	PRA	read pointer via window manager
RMODE		F	read current display mode
RPIXL	PTR	F	read/scan for pixel colour
RPTR	PTR	PR	read pointer directly
SPHDR		P	set up sprite header
SPLIN		PR	set up one line of sprite
SPRAY	PTR	P	spray pixels
SPRSP		F	calculate space required for a sprite
SPSET	PTR	P A	set up sprite definition from array
SPTR	PTR	P	set pointer to new position
SWDEF	PTR	P	(re)set sub-window definition/pointer sprite
WBLOB	PTR	P	write a blob
WSPRT	PTR	P	write a sprite

Assembler

Programmer's Interface

Pointer Interface

The base level of the Pointer Interface is accessed through extended IOSS trap #3 operations. These traps are used in the same way as ordinary QDOS IO calls, but there are some distinctive characteristics.

Where an x,y coordinate pair is required, this is passed as a long word with the x coordinate in the upper word, and the y coordinate in the lower word.

In place of the single window area used by normal console output calls (set by SD.WDEF) the Pointer Interface recognises four different window areas. The largest is the window outline: this is the total area occupied by a window. The second largest is the window hit area: this is the window outline less the window's shadow. These two areas are set by the pointer trap IOP.OUTL. The outline (of a secondary window) is used by the save and restore traps (IOP.WSAV and IOP.WRST). The outline and hit areas of the primary windows are use by the buried layers of the Pointer Interface to determine which windows are locked by other windows which are on top.

Within the hit area there is the window area set by SD.WDEF. This is the area within which all output will be put: this area will often be fairly dynamic.

Also within the hit area there are all the sub-windows. The sub-window area definitions are in a list which is set by the pointer trap IOP.SWDF. This sub-window list holds not only definitions of the sub-window areas, but, for each area, a pointer to the sprite to be used as a pointer when the pointer is in that area. The only pointer trap which uses the sub-window definitions is IOP.RPTR (read pointer). If the pointer is within a sub-window of the window, then the pointer coordinates in the pointer record are set relative to that sub-window.

As the sub-window definition list is held outside the IO sub-system, it is important that the list be detached from the window channel before the memory holding the list is returned to QDOS. This will not be a problem if the window channel is closed first or both are returned by the job being removed from the machine.

Before using any of the Pointer Interface calls, it is as well to check whether the Pointer Interface is installed, and locate the Window Manager routines.

The Pointer Interface provides facilities for pointer control, pointer access and window control as well as some additional IO calls to access the area under the pointer. Some IO calls to windows which overlap the area occupied by the pointer will cause the pointer to be removed from the screen before the call is executed. When this occurs the pointer will be restored about a fifth of a second after the last standard IO call to the screen. The pointer will, however, appear as soon as a pointer position is requested. Where possible, the screen operations will be carried out without blanking the pointer.

You will find a set of symbols defined in QDOS_IO for use with these TRAPs.

Additional IO calls

Name	D0	Function
IOP.FLIM	$6c	Find window limits
IOP.SVPW	$6d	Partial window save
IOP.RSPW	$6e	Partial window restore
IOP.SLNK	$6f	Set linkage block
IOP.PINF	$70	Information enquiry
IOP.RPTR	$71	Read pointer
IOP.RPXL	$72	Read pixel at x,y
IOP.WBLB	$73	Write blob at x,y
IOP.LBLB	$74	Write line of blobs
IOP.WSPT	$76	Write sprite at x,y
IOP.SPRY	$77	Spray pixels in blob
IOP.OUTL	$7a	Set window outline
IOP.SPTR	$7b	Set pointer position
IOP.PICK	$7c	Pick window
IOP.SWDF	$7d	Set window definition pointer
IOP.WSAV	$7e	Save window area
IOP.WRST	$7f	Restore window area

IOP.FLIM

Trap #3 D0=$6C

Find window limits

Call parameters		Return parameters	
D1		D1	preserved
D2	0	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to result area	A1	preserved
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
IPAR	D2 <> 0

This call finds the limits of where a window's outline may be set by a call to IOP.OUTL - setting the outline outside this will give an "out of range" error, setting it within this area will not, unless the window's primary is moved after the call to IOP.FLIM. A1 points to a four-word area of memory into which the limits are returned in the usual X-size, Y-size, X-origin, Y-origin format. These are absolute co-ordinates. A primary is limited to the whole screen area, a secondary to its primary's outline.

Trap #3 D0=$6D

IOP.SVPW

Save part window

Call parameters		Return parameters	
D1	x,y start of block in area	D1	address of save area
D2	0 or x,y size of save area	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	size/start of window block	A1	preserved
A2	address of save area (D2=0)	A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
ORNG	block is not in window or save area
IMEM	no room to set up save area (D2=0 only)

This routine saves part of the contents of a window into a save area in memory. The size and position of the block to be saved are passed in a 4-word definition block pointed to by A1 (c.f. IOP.FLIM). The pixel position in the save area to which the block should be saved is passed in D1. If D2<>0 then a new save area is set up, whose size in pixels is given in D2: otherwise the area pointed to by A2 is used. The routine allows the use of bit images larger than the 512x256 limit imposed by the QL's hardware.

Trap #3 D0=$6E

IOP.RSPW

Restore part window

Call parameters		Return parameters	
D1	x,y start of block in area	D1	preserved
D2	<>0 to keep save area	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	size/start of window block	A1	preserved
A2	address of save area	A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
ORNG	block is not in window or save area

This routine restores part of a save area into a block in a window. Optionally the save area may be returned to the common heap. This routine complements the IOP.SVPW routine.

IOP.SLNK

Trap #3 D0=$6F

Set Bytes in Linkage Block

Call parameters		Return parameters	
D1.w	position in linkage to set	D1	preserved
D2.w	number of bytes to set	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to data to set	A1	address of linkage block
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open

IOP.PINF

Trap #3 D0=$70

Get Pointer Information

Call parameters		Return parameters	
D1		D1.l	pointer version (n.nn)
D2		D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1		A1	window manager vector
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
IPAR	no pointer interface installed

The version number is a four byte ASCII string e.g. '1.15'. The Window Manager vector contains the entry points for the upper level routines. For example, to call the routine at vector address $08 the following code may be used:

```
MOVEQ     #$70,D0          find entry point vector
MOVEQ     #-1,D3
MOVE.L    CHAN_ID(A5),A0   set our own channel ID
TRAP      #3
TST.L     D0               is there an interface?
BNE       OOPS             ... no
MOVE.L    A1,D0            is there a Window Manager?
BEQ       OOPS             ... no
JSR       $08(A1)          call vectored routine $0
```

Trap #3 D0=$71

IOP.RPTR

Read pointer

Call parameters		Return parameters	
D1.l	x,y pointer coordinates	D1	x,y pointer coordinates
D2.b	termination vector	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to pointer record	A1	preserved
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN channel not open

The coordinates passed (in D1) to the trap are used to check whether the pointer has moved since the last call. Both the call and return parameters are in screen, **not window**, coordinates.

The termination vector is used to determine which events will cause a "complete" return from the call, and it corresponds to the least significant byte of the event vector:

bit 0	key or button stroke in window / window resize
bit 1	key or button pressed (subject to auto repeat)
bit 2	key or button up in window
bit 3	pointer moved from given coordinates in window
bit 4	pointer out of window
bit 5	pointer in window
bit 6	reserved
bit 7	window request

If both bit 4 and bit 5 are set, then the pointer call will always return immediately, even if the window is locked!

Bits 7 is used to request a pointer "hit" regardless of whether the pointer is inside or outside the window. This call must be made with infinite timeout. While such a request is pending in the top window, all windows are locked and only the top window will get the "hit". The pointer sprite will be set according to the status of bits 0 and 1. If bit 7 is set then all bits other than bits 0 and 1 should be zero. If bit 0 is set then the move window sprite will be used; if bit 1 is set then the window change size sprite is used; otherwise the empty window sprite will be used.

The pointer record is 24 bytes long:

00	long	ID of window enclosing the pointer
04	word	sub-window enclosing pointer (or -1)
06	word	x pixel coordinate of pointer within (sub-)window
08	word	y pixel coordinate of pointer within (sub-)window
0a	byte	0=no keystroke <>0 key or button code
0b	byte	0=no key down <>0 space or button depressed
0c	long	event vector all zero except LS Byte
10	4 words	(sub-)window definition (size, origin)

To determine the window that a pointer is in, the Pointer Interface scans the pile of primary windows looking for the first window whose hit area the pointer is in. If that window has a window definition list and the pointer is outside the main window definition (i.e. it is pointing to the border) then the pointer is considered to be outside all windows. If the window does not have a definition list and the pointer is outside the current window area (set by SD.WDEF), then the pointer is also considered to be outside all windows.

If the pointer is not in a window, the conventional ID -1 is returned instead of an actual ID (note that as a negative "tag" is possible, the second word of the ID should be checked to find out if the channel number is negative). In this case, the pointer coordinates will be relative to the display origin.

If the pointer is within a sub-window of the window (as defined by a IOP.SWDF call) then the x,y coordinates in the pointer record will be relative to the origin of sub-window. Otherwise, the sub-window number will be -1 and the x,y coordinates will be relative to the main window. If there is no window definition list, then the x,y coordinates in the pointer record will be relative to the origin of the current window definition. In either case, the definition of the window or sub-window is put into the end of the pointer record.

For a button on a pointer device the code is the button number. For a keypress on the keyboard, the code is the extended ASCII code of the character.

IOP.RPXL

Trap #3 D0=$72

Read pixel colour

Call parameters		Return parameters	
D1.l	x,y coordinate	D1.l	new position | colour
D2.l	scan key | scan colour	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1		A1	preserved
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
ORNG	x, y is not in window

key bit	meaning
31	set => scan required
19	set => scan until same colour: else scan to different
18/17	00=scan up, 01=scan down, 10=scan left, 11=scan right
16	set => compare with given colour, else with start colour

The x,y coordinates are relative to the current window area set by SD.WDEF. If no scan is required (D2..31=0) then the colour of the specified pixel is returned in D1.w. If a scan is required then it may proceed from the given start pixel co-ordinates in one of four possible directions, terminating when a pixel of the same/a different colour to the given colour/colour of the pixel at the start position is found. If the scan reaches the edge of the window before a pixel of the required colour is found then the co-ordinate returned in the high word of D1 is set to -1. Since the scan is in either the x or the y direction, the y or x co-ordinate of the termination pixel is the same as that of the start pixel.

IOP.WBLB

Trap #3 D0=$73

Write a blob

Call parameters		Return parameters	
D1.l	x,y coordinate	D1	preserved
D2	0	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to blob definition	A1	preserved
A2	pointer to pattern definition	A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
ORNG	x, y is not in window
IPAR	bad data structure

IOP.LBLB

Trap #3 D0=$74

Write a line of blobs

Call parameters		Return parameters	
D1.l	x,y start coordinate	D1.l	x,y end coordinate
D2.l	x,y end coordinate	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to blob definition	A1	preserved
A2	pointer to pattern definition	A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
IPAR	bad data structure

The write blob call writes a blob of the pattern into the window, and the line of blobs a line from the start to (but not including) the end coordinates, which are relative to the current window area set by SD.WDEF. A blob which falls wholly or partially out of the window causes an error in IOP.WBLB, and is ignored in IOP.LBLB.

This version checks the form of the blob and pattern against the current screen mode, and searches along each chain until it finds a definition with the appropriate form. If it encounters the end of the chain or an odd pointer before this, a "bad parameter" error will be returned.

Trap #3 D0=$76

IOP.WSPT

Write a sprite

Call parameters		Return parameters	
D1.l	x,y coordinate	D1	preserved
D2		D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to sprite definition	A1	preserved
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
ORNG	x, y is not in window
IPAR	bad data structure

The write sprite call writes a sprite into the window. This version of the Pointer Interface cannot handle sprites which partially overlap the edge of the window.

The x,y coordinates are relative to the current window area set by SD.WDEF.

This version checks the form of the sprite against the current screen mode, and searches along the chain until it finds a definition with the appropriate form. If it encounters the end of the chain or an odd pointer before this, a "bad parameter" error will be returned.

The internal sprites may be used by passing a small number in A1, rather than a pointer:

Name	Number	Sprite
SP.ARROW	$00	arrow
SP.LOCK	$01	padlock
SP.NULL	$02	empty window
SP.MODE	$03	wrong mode (4 or 8)
SP.KEY	$04	keyboard entry
SP.BUSY	$05	no entry sign
SP.WMOVE	$06	window move
SP.WSIZE	$07	window change size

IOP.SPRY

Trap #3 D0=$77

Spray pixels in blob

Call parameters		Return parameters	
D1.l	x,y coordinate	D1	x,y coordinate
D2	number of pixels to spray	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to blob	A1	preserved
A2	pointer to pattern	A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
ORNG	x, y is not in window

This call sprays the number of pixels required into a window: the colour of each is determined by the pattern, and each falls on a non-transparent part of the blob. If the number of pixels required exceeds the number of pixels in the blob the call will terminate with no error, and may duplicate the effect of a call to IOP.WBLB: but there is no guarantee that one or more calls to IOP.SPRY with the same blob in the same position will eventually fill in the entire blob.

IOP.OUTL

Trap #3 D0=$7A

Set Window Outline

Call parameters		Return parameters	
D1.l	x,y shadow widths	D1	???
D2	1 to keep contents, else 0	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to window definition block	A1	preserved
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
ORNG	window not within screen

This call defines a window's outline, its hit area and shadow. A1 points to a normal window definition block (4 words: x,y sizes, x,y origin) which defines the window hit area. The shadow widths area added to this to make the window outline, and the shadows are drawn. It is the use of this call which indicates to the Pointer Interface that the window concerned is a genuine managed window. All subsequent SD.WDEF calls to this window will be checked against the window hit area instead of the total display area.

For secondary windows, IOP.OUTL also saves the area beneath the window, avoiding the need for explicit IOP.WSAV and IOP.WRST calls.

If the key in D2 is set to 1 then the contents of the window will be preserved, allowing applications to move a window with one call to IOP.OUTL: note that the size must stay the same for this to work properly!

IOP.SPTR

Trap #3 D0=$7B

Set pointer position

Call parameters		Return parameters	
D1.l	x,y coordinate	D1	x,y coordinate
D2.b	origin key	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1		A1	preserved
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
ORNG	x, y is not in window

This call sets the current pointer position. It should be used with discretion as sudden pointer position changes could prove to be very unpleasant for the user.

The origin key should be zero if the pointer coordinates in D1 are absolute. D1 is always set to absolute coordinates on return. A key of -1 will set the position relative to the current window definition. A key of 1 will set it relative to the hit area.

Trap #3 D0=$7C

IOP.PICK

Pick window

Call parameters		Return parameters	
D1.l	job ID or key	D1	???
D2	0 or k.wake	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1		A1	preserved
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
IJOB	invalid job ID

If a job ID is given, the primary window owned by that job will be "picked" to the top of the pile. If the key is given as -1, then the bottommost job will be picked to the top. If the key is given as -2, the window is marked "unlockable". If D2 is set to k.wake, a wake event is sent after the pick. This call will work even if the channel given is locked: it should be used very sparingly, if at all.

Trap #3 D0=$7D

IOP.SWDF

Set Sub-Window Definition List

Call parameters		Return parameters	
D1		D1	preserved
D2		D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1		A1	preserved
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open

This call is used to set the pointer to the sub window definition list. This is a sub-set of the window working definition. A1 points to a long word pointer to a table of pointers to sub-window definitions. This pointer may be zero. It is followed by a sub-window record for the main part of the window. The pointers to sub-window definitions are long words, the list is terminated by a zero long word. Each pointer points to a sub-window record.

A sub-window record specifies the area and, if desired, a pointer to a sprite to be used as pointer when the pointer is in that sub-window. The structure of a sub-window record is as follows:

sw_xsize	$00	word	(sub-)window x size (width) in pixels
sw_ysize	$02	word	(sub-)window y size (height) in pixels
sw_xorg	$04	word	x origin of (sub-)window
sw_yorg	$06	word	y origin of (sub-)window
sw_wattr	$08		(sub-)window attributes in 4 words - spare, border width, border colour, paper colour
sw_psprt	$10	long	pointer to pointer sprite for this (sub-)window

IOP.WSAV

Trap #3 D0=$7E

Window Area Save

Call parameters		Return parameters	
D1.l	length of save area (or 0)	D1	preserved
D2		D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	address of save area (D1>0)	A1	preserved
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open
IMEM	out of memory

IOP.WRST

Trap #3 D0=$7F

Window Area Restore

Call parameters		Return parameters	
D1		D1	preserved
D2.b	<>0 to keep save area	D2	preserved
D3.w	timeout	D3	preserved
		D4+	all preserved
A0	window channel ID	A0	preserved
A1	address of save area (or 0)	A1	preserved
A2		A2	preserved
		A3+	all preserved

Error returns:

ICHN	channel not open

These routines save and restore bit images from and to a window's hit area. The memory to be used may be supplied by the application (D1 or A1 non-zero) or allocated internally. The former option is preferable, as the internal save area pointer may already be in use; it is used to implement pull-down windows, for instance.

Window Manager

The window management routines are supplied to do all of the most common operations in handling pull-down movable and resizable windows and menus within these windows. The actions of the window management routines are controlled by data structures supplied by the application.

Symbols for the vectors are defined in the `WMAN_KEYS` file, which may be INCLUDEd in any program which makes use of these routines.

In many cases, the window data structures will have pointers to application supplied action routines. This effectively means that the application code calls the window manager routines, which, in turn, call application routines. To simplify the application code, the window manager routines treat certain registers in a uniform way:

When the window manager routines call an application routine, A2 is set to point to the window manager vector, while A5 and A6 are not used or modified by any window manager routines. Thus A5 and A6 can be used by the application routines as pointers to internal data structures.

There are four distinct phases involved in setting up and using a managed window. First the window definition is copied and expanded into the working definition. Next the working definition is used to open an appropriate window. Then the window contents are filled in. Finally, the window is accessed via a call to read the pointer.

Before starting to set up a window, the application must have initialised the window status area. This is a work area which is accessed by both the window management routines and the application program. It contains such useful information as the current item, the panning and scrolling state of the application sub-windows and the status of all the items within all the (sub-)windows.

The start of the status area holds pointers to the window definitions. Often the initial state of the rest of the status area will be mostly zero. Where pull-down windows are used, the status area will usually be maintained from one use of the window to the next time the window is set up to be used.

Setup routines

The routine WM.SETUP may be called to transfer a window definition to the window working definition. It is possible for an application to set up its own working definition, but it is easier to use the window manager routine.

The window definition is a fixed skeleton of the appearance of the window, as in practice the window contents are liable to change. This variability is catered for in two ways. Firstly, the application must supply its own routine to transfer the definition of each application sub-window: for standard format menus, the application sub-window setup routine will just be a call to WM.SMENU. Secondly, after the working definition has been set up, it may be modified by the application. In particular, if there is a menu within the window which has a variable object list, then the object lists should be set up by the application code after the main part of the working definition has been set up by WM.SETUP.

Depending on the size of window required, one of a number of layouts will be selected from the list provided in the window definition. The WM.FSIZE routine may be used to determine which will be selected: the result of this might, for instance, be used to allocate the correct amount of memory for the working definition.

In the next phase the window is initialised. For the primary window, the routine WM.PRPOS will position and set up a primary window. For secondary windows, the routine WM.PULLD should be called to pull down a window within the primary window area. These routines will try to position the window so that the pointer will point to the current item in the window without being moved. If this is not possible, then the pointer itself will be moved. WM.PRPOS and WM.PULLD both set the window border and clear the window. After the window has been initialised, fancy borders or other adornments may be added by the application.

The window should now be filled in. Most of the operations to fill in the window will be performed by the routine WM.WDRAW. However, the application sub-windows are initialised but not filled in. This is left to the application code. If the sub-window is a standard format menu, then the menu drawing routine WM.MDRAW may be called to fill in the sub-window.

In the final phase, the routine WM.RPTR may be called to read the pointer. This routine will return with the event vector in D2. This will indicate what actions (if any) are required to be done. Any "hits" on loose menu items or items within a menu sub-window will have been processed within the window management level by the hit and action routines supplied by the application.

If a "hit" on a loose menu item, or a sub-window menu item, requires the window to be changed (moved, squashed, stretched, thrown away etc.), then the action routine should set the appropriate bit in the event vector and return to the application code. This ensures that the application will always have control over its own windows.

Vector $54

WM.FSIZE

Find size of layout

Call parameters		Return parameters	
D1	x,y size (or 0)	D1	actual x,y size
D2		D2.w	layout number
		D3+	all preserved
A0		A0	preserved
A1		A1	preserved
A2		A2	preserved
A3	pointer to window defn	A3	preserved
A4		A4+	all preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Not set

If this routine is required it will usually be called before WM.SETUP to determine which of the possible layouts WM.SETUP will select from the repeated part of the window definition. If the required size is given as 0 then the default size will be used. The actual size that the window will be is returned in D1: this will be the same as that passed if the layout selected is scaleable, otherwise it will be smaller in one or both dimensions. It will be larger if the size requested was smaller than the smallest possible layout.

The layout number is returned in D2: this will be zero for the first layout, 1 for the second and so on. This may be used to allocate the correct amount of memory for the working definition (the following code assumes you have set the size required and pointer to the window definition):

```
    JSR      WM.FSIZE(A2)      find out which layout
    ADD.W    D2,D2
    ADD.W    D2,D2             turn into offset
    MOVE.L   WWTAB(PC,D2.W),D1 find space in table
    JSR      MEMGET(PC)        and allocate it
    ...
WWTAB
    DC.L     WWA.MENU          space for layout 0...
    DC.L     WWB.MENU          ...and layout 1
```

Vector $04

WM.SETUP

Setup a managed window

Call parameters		Return parameters	
D1.l	x,y size (or 0, or -1)	D1.l	x,y size
		D2+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to status area	A1	preserved
A2		A2	preserved
A3	pointer to window defn	A3	preserved
A4	pointer to working defn	A4+	all preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Always returns OK

The managed window setup routine WM.SETUP is called to transfer information from the window definition to the window working definition. It is the responsibility of the applications code to provide an area of memory large enough to accommodate the window working definition. This may seem unfair, but only the application will be able to determine the maximum space required in this area.

If the window size is given as 0, then the default window size will be used. If the window size is given as -1, then the window size and position in the working definition will not be changed. This is to allow re-use of a window (see WM.UNSET and WM.WRSET).

The window size is used to determine the window layout and scaling factors. If no definition can be found that is small enough to accommodate the given window size, than the size of the window in the last definition in the list will be used.

Where possible, WM.SETUP will set up complete structures. If there are empty pointers or structures in the window definition, these will be transferred to the working definition as empty pointers or structures. When it comes to transferring the definitions of application sub-windows to the working data structure, the basic sub-window definition is transferred, and then an application supplied routine is called to setup the rest of the sub-window working definition.

To simplify calls back into the window manager routines, A2 will be set to point to the window manager vector, while A5 and A6 remain unused since the call to WM.SETUP.

In the case of a standard menu, the application supplied routine will just be a branch to the standard menu setup routine

```
    JMP       WM.SMENU(A2)      setup standard menu
```

```
set pointer to window status area in working definition
set pointer to window definition in window status area
set no current item in window status area
set window mode in status area
set channel ID in working definition
set pointer to pointer record
find definition to suit size
set x,y scaling factors
set window attributes block
set pointer to pointer sprite
set loose menu item attributes block
set help pointer
set pointer to information sub-window list
for all information sub-windows
    set true size and origin
    set window attributes
    set pointer to information object list
set number of information sub-windows
for all information sub-windows
    set end of list
    for all information objects
        set object size and position
        set object type and attributes
        set object pointer
set number of information objects
set end of list
set pointer to loose menu item list
for all loose menu items
    set object size and position
    set object justification rule
    set object type and selection keystroke
    set pointer to object and item number
    set pointer to action routine
set number of loose menu items
set end of loose menu item list
set application sub-window list address
set sub-window sprite list address to same
for all application sub-windows
    set application sub-window pointer list (implicit end=0)
set number of application sub-windows
for all application sub-windows
    set true size and origin
    set window attributes
    set pointer to pointer sprite
    set pointers to sub-window draw and hit routines
    set pointer to sub-window control routine
    set selection keystroke
    for x and y
        set maximum number of sections
        if non-zero
            set pointers to part-window control blocks
            copy all control attributes
        else
            preset control section of menu definition to 0
    call application sub-window setup routine
```

The call parameters to the application sub-window setup routine are the same as the parameters to the standard menu setup routine. The registers A3 and A4 are used as running pointers to the window definition, and the working definition respectively. On calling the application sub-window setup routine A3 points after the application sub-window basic definition, or after the sub-window control definition (if present). A4 points to the next unset location in the window working definition. On exit from the application sub-menu setup, A4 should be updated to point to the next unset location in the window working definition. A3 need not be updated or preserved.

The window scaling parameters D1 and D2 are the amount by which the window size exceeds the minimum in the x and y directions. These are words.

Application Sub-Window Setup Routine

Call parameters		Return parameters	
D1.w	x scaling	D1	preserved
D2.w	y-scaling	D2	preserved
		D3+	all preserved
A0		A0	???
A1	pointer to status area	A1	???
A2	window manager vector	A2	???
A3	pointer to sub-window defn	A3	???
A4	pointer to working defn	A4	updated
A5	not used by any routine	A5	used as required
A6	not used by any routine	A6	used as required

Error returns:

D0 and the status register must be set

A1 contains the pointer to the status area which was passed to WM.SETUP. To simplify calls back into the window manager routines, A2 is set to point to the window manager vector, while A5 and A6 remain unused since the call to WM.SETUP. All of A0 to A3 may be treated as volatile.

Vector $08

WM.SMENU

Setup standard sub-window menu

Call parameters		Return parameters	
D1.w	x scaling	D1	preserved
D2.w	y-scaling	D2	preserved
		D3+	all preserved
A0		A0	preserved
A1	pointer to status area	A1	preserved
A2		A2	preserved
A3	pointer to sub-window menu defn	A3	updated to after menu def
A4	running pointer to working defn	A4	updated to next unset location
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Always returns OK

Vector $08 WM.SMENU Set Up a Standard Menu Sub-Window

```
set pointer to menu status block
set item attributes
set number of rows and columns
set pointers to spacing lists
    copy spacing lists
set pointers to index object lists
    set index object lists
set pointer to row list
    set row pointers
        set object lists
```

Window Manager Set Window Routines

The primary window position routine WM.PRPOS is called to position the primary window for an application. The position of the window is determined by the current pointer position in conjunction with the "origin" of the window (specified in the working definition) or the position of the current menu item (specified in the window status area). This ensures that the pointer will move as little as possible when the window is opened, while keeping the window within the limits of the display. A window is always positioned such that its X origin is a multiple of four, and its Y origin is a multiple of two: this ensures that any stipples used in the window are always "in phase".

The routine WM.PULLD is the equivalent call for a secondary window. This has the same effect as the primary open call, but the window pulled down is limited to be within the primary window area.

The routine WM.UNSET is called to unset the sub-window definition pointer in the screen driver so that a working definition may be removed or replaced.

The routine WM.WRSET is called to reset a primary or pull down window so that the same window may be used with a new working definition. N.B. see WM.UNSET

Vector $0C	Primary Window Positioning	WM.PRPOS
Vector $10	Pull Down Window Open	**WM.PULLD**
Vector $14	Window Unset	**WM.UNSET**
Vector $18	Window Reset	**WM.WRSET**

Call parameters		Return parameters	
D1	window "origin" or -1.l	D1+	all preserved
A0		A0	channel ID of window
A1-A3		A1-A3	preserved
A4	pointer to working defn	A4+	all preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors

If an "origin" position is given, this (in absolute screen coordinates) is used, in place of the current pointer position, to position the window.

Vector $10   WM.PULLD   Pull Down a Window

```
open console and fill in its channel ID
set "pulled down" flag
... then WM.PRPOS
```

Vector $0C   WM.PRPOS   Position a primary window

```
get window channel ID from working definition
find current pointer position and save it
calculate window origin
set window outline and shadow (saves pull down window area)
adjust pointer position
adjust window definition block to exclude border
... then WM.WRSET
```

Vector $18   WM.WRSET

```
draw border and clear window
set sub-window definition pointer
```

Vector $14   WM.UNSET

```
unset sub-window definition pointer
if window was pulled down
        restore area covered up
        restore old pointer position
```

Window Manager Drawing Routines

When the working definition has been set up and the window opened, the general purpose routine WM.WDRAW is called to draw the entire window contents. The information windows are set up and the information objects are drawn. Then the loose menu items are drawn. Finally each application sub-window is set up, bordered and cleared and the application sub-window draw routine is called to fill in the contents and the index bars.

WM.WDRAW

Vector $1C

Draw window contents

Call parameters		Return parameters	
		D1+	all preserved
A0		A0	channel ID of window
A1-A3		A1-A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors

Vector $1C   WM.WDRAW   Draw Window Contents

```
for all information sub-windows
    set sub-window size, position and border
    set sub-window background
    clear sub-window
    for each object
        draw in position

for all menu items
    draw in position

for all application sub-windows
    set sub-window size, position and border
    set sub-window background
    clear sub-window
    call application sub-window draw routine
```

The application sub-window draw routine is called to draw the contents and, if required, the indices for the sub-window. When it is called, the window definition (SD.WDEF) will have been set to the sub-window outline. The application routine is passed the pointer to the start of the working definition in A4, and the pointer to the sub-window definition in A3. The sub-window definition in the window status area will be set and D7 holds the origin of the window, not the sub-window. The pointer to the window status area can be found in the working definition which is pointed to by A4.

Application Sub-Window Draw Routine

Call parameters		Return parameters	
		D1+	all preserved
D7.l	x,y origin of window	D7	preserved
A0	window channel ID	A0	preserved
A1		A1	???
A2	window manager vector	A2	???
A3	pointer to sub-window defn	A3	???
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

D0 and the status register must be set

To simplify calls back into the window manager routines, A2 is set to point to the window manager vector, while A5 and A6 remain unused since the call to WM.WDRAW.

Part Window Drawing Routines

There are four window management routines to help drawing or redrawing parts of windows. These routines may be called from the application sub-window drawing routines (called from WM.WDRAW) or from the action or control routines (called from WM.RPTR and WM.MHIT).

These are the standard menu drawing routine, WM.MDRAW, the index drawing routine, WM.INDEX, the sub-window definition routine, WM.SWDEF, and the loose menu item drawing routine, WM.LDRAW.

WM.MDRAW

Vector $20

Standard Menu Drawing

Call parameters		Return parameters	
		D1-D2	all preserved
D3.b	0 all, -1 selective	D3+	all preserved
A0	window channel ID	A0	channel ID of window
A1		A1	preserved
A2		A2	preserved
A3	pointer to sub-window defn	A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors

If D3 is set to -1 for the call to WM.MDRAW, then only those items whose status has the change bit set (WSI..CHG) will be drawn. Note that the status flags are not modified by this routine; this is because an item may consist of more than on object, or an object may be visible in more than one section, so the status flags need to be preserved throughout the routine. The application will therefore need to clear any change bits that are set after this routine has been called.

Vector $20   WM.MDRAW   Draw Standard Menu in Sub-Window

```
set sub-window definition
    for all row sections
        for all rows visible within section
            for all column sections
                for all columns visible within section
                    if draw all or WSI..CHG set in status
                        draw object in colours appropriate to status
```

Vector $24

WM.INDEX

Standard Sub-Window Index

Call parameters		Return parameters	
		D1+	all preserved
A0	window channel ID	A0	channel ID of window
A1		A1	preserved
A2		A2	preserved
A3	pointer to sub-window defn	A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors

Vector $24   WM.INDEX   Draw Sub-Window Indices

```
set main window definition
if column index
    for all column sections
        for all columns visible in section
            draw column index object
if row index
    for all row sections
        for all rows visible in section
            draw row index object

if pannable
    for all column sections
        draw pan bar
if scrollable
    for all row sections
        draw scroll bar

set sub-window definition
if pannable
    for all column sections
        for all row sections
            draw pan arrows
if scrollable
    for all row sections
        for all column sections
            draw scroll arrows
```

WM.UPBAR

Vector $70

Update pan/scroll bars

Call parameters		Return parameters	
D0	x,y section to update	D0	preserved
		D1+	all preserved
A0	window channel ID	A0	channel ID of window
A1		A1	preserved
A2		A2	preserved
A3	pointer to sub-window defn	A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors

This routine allows re-drawing of a given section scroll or pan-bar. If you set D0 to -1, nothing is updated. The first call to draw bars and arrows should be WM.INDEX, any further update of the bar positions should be done with WM.UPBAR. This saves a lot of time as only the part which (possibly) has been modified is re-drawn. There is also no need to re-draw the arrows (if they exists) after a scroll or pan operation.

WM.SWDEF

Vector $28

Set Sub-Window Definition

Call parameters		Return parameters	
		D1+	all preserved
A0	window channel ID	A0	channel ID of window
A1		A1	preserved
A2		A2	preserved
A3	pointer to sub-window defn	A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors

This routine may be used to reset the definition of any application or information sub-window.

Vector $28   WM.SWDEF   Set Sub-Window Definition

```
find sub-window definition
make absolute screen coordinates
set window definition with zero border width
```

Vector $2C

WM.LDRAW

Loose Menu Item Drawing

Call parameters		Return parameters	
		D1-D2	all preserved
D3.b	0 all, -1 selective	D3+	all preserved
A0	window channel ID	A0	channel ID of window
A1		A1	preserved
A2		A2	preserved
A3		A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors

If D3 is set to -1 for the call to WM.LDRAW, then only those items whose status has the change bit set (WSI..CHG) will be drawn. This routine is normally used when a change in status of one loose item affects the status of others, or when a loose item's object has been changed. Note that the status flags are not modified by this routine; this is because an item may consist of more than on object, or an object may be visible in more than one section, so the status flags need to be preserved throughout the routine. The application will therefore need to clear any change bits that are set after this routine has been called.

Vector $2C   WM.LDRAW   Draw Loose Menu Items

```
set main-window definition
for all loose menu items
    if draw all or WSI..CHG set in status
        draw object in colours appropriate to status
```

WM.IDRAW

Vector $3C

Draw information sub-windows

Call parameters		Return parameters	
		D1-D2	all preserved
D3	bits clear to redraw window	D3+	all preserved
A0		A0	channel ID of window
A1		A1	preserved
A2		A2	preserved
A3	pointer to sub-window defn	A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors

This routine allows an application to re-draw any of the first 32 information sub-windows: if bit N of D3 is clear then information sub-window N will be cleared and re-drawn. This routine will normally only be used when the information objects in a window have been changed.

```
for information sub-window 0..31
    if bit N clear in D3
        set sub-window definition
        draw sub-window border
        clear sub-window
        for all objects in sub-window
            draw object
```

There is a set of four vectors used to set the window to an area used by an information sub-window, loose menu item, application sub-window or section of application sub-window. In each case D1 specifies the number of the entity (not to be confused with a menu item number) and D2 specifies the colour(s). If D2 is a negative long word, then only the window area will be set, otherwise these routines will set the ink, paper and strip colours and the "over" state to 0 as well as setting the area.

WM.SWINF

Vector $58

Set window to info window

Call parameters		Return parameters	
D1.w	info window number	D1	preserved
D2.l	ink colour / no reset	D2	preserved
		D3+	all preserved
A0		A0	channel ID of window
A1		A1	pointer to window in work def
A2		A2	preserved
A3		A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors
ORNG Info window number out of range

WM.SWLIT

Vector $5C

Set window to loose item

Call parameters		Return parameters	
D1.w	loose item number	D1	preserved
D2.l	item status / no reset	D2	preserved
		D3+	all preserved
A0		A0	channel ID of window
A1		A1	pointer to item in work def
A2		A2	preserved
A3		A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors
ORNG Item number out of range

Vector $60

WM.SWAPP

Set window to application sub-window

Call parameters		Return parameters	
D1.w	application window number	D1	preserved
D2.l	ink colour / no reset	D2	preserved
		D3+	all preserved
A0		A0	channel ID of window
A1		A1	pointer to window in work def
A2		A2	preserved
A3		A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors
ORNG Application window number out of range

Vector $64

WM.SWSEC

Set window to application sub-window section

Call parameters		Return parameters	
D1.l	x,y section numbers	D1	preserved
D2.l	ink colour / no reset	D2	preserved
		D3+	all preserved
A0		A0	channel ID of window
A1		A1	preserved
A2		A2	preserved
A3	ptr to sub-window definition	A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors
ORNG Application window or section number out of range

Vector $68

WM.DRBDR

Draw border around current item

Call parameters		Return parameters	
		D1+	all preserved
A0	channel ID of window	A0	preserved
A1	window status area	A1	preserved
A2		A2	preserved
A3		A3	preserved
A4		A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors

This routine draws a border using the current item information in the window status area.

To clear the current item, set the most significant bit of WS_CITEM and, if WS_CIACT is clear, call WM.DRBDR, otherwise call the routine pointed to by WS_CIACT and then clear WS_CIACT.

To set a current item, set WS_CITEM, WS_CIBRW, WS_CIPAP (to the highlight colour) and the hit area WS_CIHIT. Then call WM.DRBDR. Finally reset WS_CIPAP to the background colour.

Window Manager Access Routines

Once the window, and all its sub-windows, have been set up, the pointer may be read using the window read pointer vector. This routine repeatedly reads the pointer, waiting for a move or keystroke event, and calls any hit or action routines that may be required. If any bits in the window or sub-window bytes of the event vector become set, then the routine will return. Other window manager access routines are available to handle menus within sub-windows and to provide utility support for application sub-windows

Window Manager Read Pointer

The window manager read pointer routine (WM.RPTR) handles all the pointer movement and keystrokes outside the sub-windows. It also does some occasional operations within sub-windows, and traps some keystrokes before they reach the application sub-window hit routines.

The rules governing the operation of WM.RPTR are rather complex, but are designed to make the interface operate as close to an intuitive model as is reasonable. The operation is complex because the interface has to be capable of handling not only menu selection by keystroke and menu selection by pointing device, but also menu selection by cursor key and arbitrary pointer input.

The three most important keystrokes are SPACE, which corresponds to a click on the left mouse button, ENTER which corresponds to a click on the right mouse button and ESC. SPACE or left click is referred to as "hit", ENTER or right click is "do". For some reason, ESC is known as "cancel".

Current Item

One of the functions of WM.RPTR (and its menu support routine WM.MHIT) is to maintain a current menu item. This item is outlined on the display. As long as the pointer remains within the "hit area" of the item, the item will remain outlined. As soon as the pointer moves out of the hit area, then the outline will be removed. If the current item is "hit", then, if it is available, the status is toggled, and the appropriate action routine called. "do" is similar to "hit" except that if the item is available the status is set to selected before the action routine is called.

Alternatively, items can be selected on a single keystroke. This has the effect of moving the pointer to a new current item, and then causing a "hit". Since the "hit" will cause a call to an action routine, it is possible for the application to automatically convert the "hit" to a "do" (or a "cancel" or any other event).

From the point of view of WM.RPTR, the main window is divided into two distinct areas: that part of the window which falls within an application sub-window, and that part not within any application sub-window. Every window is considered to have at least some menu operations. Some of these, e.g. HELP or DO, may be accessible from any application sub-window.

Keystroke Selection

Most keystrokes on the keyboard are treated as shorthand menu selections. The keystroke is converted to upper case, and it is compared against the selection keystrokes defined for the loose menu items, the selection keystrokes defined for the application sub-windows or, in WM.MHIT, the selection keystrokes defined for the sub-window menu items.

The current version of the Window Manager allows you to underscore the character which is the selection keystroke of a text item. The type of this item is text-position, which means, first character is -1, second -2 and so on.

There are some keystrokes which are defined to cause window events:

ENTER or a double click will cause a "do" event;
ESC will cause a "cancel" event;
F1 will cause HELP event;
CTRL F4 will cause a MOVE window event;
CTRL F3 will cause a change SIZE event;
CTRL F2 will cause a WAKE event;
CTRL F1 will cause a SLEEP event.

The treatment of these keystrokes will depend on both the organisation of the window, and the position of the pointer.

The WM.RPTR routine is a loop reading the pointer record. Whenever there is a move or keystroke to be processed, it checks first of all for the event keystrokes, then other keystrokes, and if there is no keystroke, it checks whether the current item has changed. When appropriate, it calls either a loose menu item action routine, or a application sub-window hit routine. If, at the end of all the processing of a keystroke or move an event has been generated, WM.RPTR will return. Otherwise it will continue to read the pointer record.

If there is a "do" event and there is a current item, then the corresponding item is selected and the appropriate action routine is called.

If there is an event keystroke other than "do" or there is a "do" with no current item, then the loose items are searched for a corresponding selection key. If one is found, the loose menu item status is toggled and the action routine called. If no corresponding selection key is found, then, unless it is a "do" or a "cancel" within an application sub-window, the appropriate bit will be set in the event vector and the routine will return.

If there is a "do" or a "cancel" within an application sub-window and there is no "do" or "cancel" loose menu item, then the application sub-window hit routine will be called.

If there is not an event keystroke, a check is make to see if the pointer has moved outside the current item hit area. If it has, the current item is cleared (set negative) and the border redrawn.

Next, if there is a keystroke, the loose menu item list will be searched for a corresponding selection keystroke. If one is found, the item status will be toggled and then the appropriate action routine will be called.

If the keystroke is not found in the loose menu item list then all (except the current) application sub-windows are searched for a corresponding selection keystroke. If one is found, the pointer is moved to the centre of the application sub-window and the sub-window hit routine is called.

If there is no keystroke, or the keystroke is not the selection keystroke for a loose menu item or an application sub-window, then, if the pointer is within a sub-window, the hit routine is called, or else the loose menu item list is searched to find a new current item.

On return from any loose menu item action routines, D4 is checked. If it is non zero, the corresponding bit of the window event byte is set and WM.RPTR returns after testing D0.

On return from a sub-window hit routine the window byte of the event vector is checked. If any bits are set, WM.RPTR returns after testing D0.

If a loose menu action routine or application sub-window hit routine returns a non-zero condition code, WM.RPTR will return after testing D0. This can be used to force a return without either an event or error.

Vector $30 **WM.RPTR**

Read Pointer

Call parameters		Return parameters	
A0		A0	channel ID of window
A1		A1	preserved
A2		A2	preserved
A3		A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors
Any error returned by action or hit routine

Vector $30 WM.RPTR Read Pointer

```
repeat until window event or error
    read pointer
        if event keystroke
            process it and call appropriate action/hit routine
            next read pointer

    clear current item if pointer moved out of it

    if keystroke
        process it and call appropriate action/hit routine
        next read pointer

    if in application sub-window
        call hit routine
        next read pointer

    if new current item
        set item and border
```

The window manager requires all application sub-windows to have hit routines. In the case of a standard format menu in an application sub-window, this may be just a direct jump to the WM.MHIT routine:

```
JMP      WM.MHIT(a2)      do move or hit in standard menu
```

```
|                                                                          |
|    Application Sub-Window Hit Routine                                    |
|                                                                          |
|    Call parameters                     Return parameters                 |
|                                                                          |
|    D1   x,y pointer position           D1   x,y pointer position        |
|    D2   uppercased key, -1 or 0        D2   ???                         |
|                                        D3.w timeout for next PT.RPTR    |
|    D4   event number of keystroke      D4   ???                         |
|                                        D5+  all preserved               |
|                                                                          |
|    A0   window channel ID              A0   preserved                   |
|    A1   pointer to status area         A1   ???                         |
|    A2   window manager vector          A2   ???                         |
|    A3   pointer to sub-window defn     A3   ???                         |
|    A4   pointer to working defn        A4   preserved                   |
|    A5   not used by any routine                                         |
|    A6   not used by any routine                                         |
|                                                                          |
|    Error returns:                                                        |
|                                                                          |
|         D0 and the status register must be set                          |
|                                                                          |
```

The pointer in D1 is in absolute (`not sub-window`) coordinates. The uppercased keystroke in D2 also has SPACE ($20) converted to "hit" ($01) and ENTER ($0a) converted to "do" ($02). If D2 is -1, then the application sub-window has been "hit" by an external keystroke.

D4 can only be 0, pt..do (16) or pt..cancel (17) when the application sub-window hit routine is called. All other event keystrokes are handled by the routine WM.RPTR.

If a bit is set in the window byte of the event vector by a hit routine, then WM.RPTR will return to the application. Note that WM.RPTR does not set the "do" event if the pointer is within an application sub-window: this is left to the hit routine.

An application sub-window hit routine may, of course, set the "do" event bit at any time.

D3 will normally be returned unchanged. For compatibility, the msw of D3 is ignored by WM.RPTR. For WM.RPTRT, the msw should be cleared if D3 is modified. If, for example, the application sub-window requires to monitor the keypress byte continuously, a short or even zero timeout may be specified. Note that, if a zero timeout is specified, the keystroke (as opposed to keypress) will always be zero.

Vector $34

WM.MHIT

Standard window hit routine

Call parameters		Return parameters	
D1.l	x,y pointer position	D1	preserved
D2	uppercased keystroke or 0	D2	preserved
		D3	-1
D4.b	0 or pt..do	D4+	preserved
A0		A0	channel ID of window
A1		A1	preserved
A2		A2	preserved
A3	ptr to sub-window definition	A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors

Vector $34   WM.MHIT   Standard Menu Hit

```
if no keystroke and no current item
    find new current item
    if found: mark current item
else if "hit" or DO
    find current item
    if found
        mark current item
        if current item available
            if HIT: toggle status
            if DO: set status selected
            redraw current item and call action routine
            if status changed: redraw current item
else
    find matching selection keystroke
    if found
        un-mark current item
        set pointer
        mark current item
        if current item available
            toggle status
            redraw current item and call action routine
            if status changed: redraw current item
```

This routine is intended to be called from application sub-window hit routines to locate the appropriate section of a multiple section window and check for "hit" or "do" on the pan or scroll arrows, or for pan or scroll keystrokes.

Vector $48

WM.MSECT

Find menu section

Call parameters		Return parameters	
		D0.w	0 or pan/scroll item number
D1.l	x,y pointer position (absolute)	D1	preserved
D2	uppercased keystroke	D2	preserved
D3		D3	x,y section number -1 if in pan/scroll arrows
D4.b	event number of keystroke	D4.b	preserved or pt..pan or pt..scrl
A0	channel ID of window	A0	preserved
A1		A1	preserved
A2		A2	preserved
A3	ptr to sub-window definition	A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

>0 of pan or scroll event generated

The item number returned in D0.w is the pan/scroll item and is set only if D4 is set to pt..pan ($A) or pt..scrl ($B). The less significant byte is the section number to which the operation applies, the most significant nibble is %0111. Bits 8 to 11 specify the type of event in greater detail.

Bit 8	set for scroll down or pan right
Bit 9	set for pan left or right
Bit 10	set for extra pan/scroll ("do" on arrows or ALT+SHIFT)
Bit 11	zero

The action routines called from WM.MHIT are optional. As WM.MHIT sets the appropriate byte in the status block, it is not necessary for the application to do anything about a "hit" until a "do" causes WM.RPTR to return to the application. On the other hand, the action routine itself can set the "do" event, or it can act on the "hit" directly.

Note that the action routine is called on a "hit" whether the status is selected or unselected, but not if it is unavailable. The action routine may change the status of the item, or even the objects within the item.

Standard Menu Action Routine

Call parameters		Return parameters	
D1.l	virtual column/row for item	D1	???
D2.w	item number	D2	???
		D3	???
D4.l	0 or pt..do	D4.b	0 or window event to set
		D5+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to menu status block	A1	???
A2	window manager vector	A2	???
A3	pointer to sub-window defn	A3	???
A4	pointer to working defn	A4	preserved
A5	not used by any routine	A5	used as required
A6	not used by any routine	A6	used as required

Error returns:

D0 and the status register must be set

(A1,D2.w) points to the current item's status byte. D4 may be set to force a "do" or any other window event.

If there is no action routine for a particular item, then a "do" keystroke will cause a "do" event.

The application window control routine is called either from the routine WM.RPTR for a "hit" on the pan or scroll bars associated with a window, or from WM.MHIT when there has been a "hit" on the pan or scroll arrows. The item number is the special item number for pan and scroll operations. The least significant byte gives the part menu number to be panned or scrolled. The routine may adjust the window itself or merely adjust the control tables and call the sub-window draw routine. In either case, the event flag should be set to zero. Alternatively the event flag may be left set, and then WM.RPTR will return to the calling routine with the appropriate event set.

If the routine is called as the result of a "hit" on a pan or scroll bar, the most significant word of D3 will hold the position of the hit, while the least significant word of D3 will hold the length of the bar. Otherwise the routine will have been called as a result of a "hit" on the arrow bars, in which case D3 will have the value -1.

Application Window Control Routine

Call parameters		Return parameters	
		D1	???
D2.w	item number	D2	???
D3.l	position of "hit" or -1	D3	???
D4.b	pan or scroll event	D4.b	0 or window event to set
		D5+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to status area	A1	???
A2	window manager vector	A2	???
A3	pointer to sub-window defn	A3	???
A4	pointer to working defn	A4	preserved
A5	not used by any routine	A5	used as required
A6	not used by any routine	A6	used as required

Error returns:

D0 and the status register must be set

The simplest form of control routine is just a call to the window manager panning and scrolling routine WM.PANSC

```
JMP      WM.PANSC(A2) do standard pan scroll
```

The loose menu item action routines are similar to the standard menu action routines (after all, a loose menu item is really part of a standard menu). One difference is that the menu manager requires there to be an action routine for a loose item corresponding to an event.

Loose Menu Item Action Routine

Call parameters		Return parameters	
D1.l	x,y pointer position	D1	???
D2.w	uppercased keystroke	D2	???
		D3	???
D4.b	event number of keystroke	D4.b	0 or window event to set
		D5+	all preserved
A0	window channel ID	A0	preserved
A1	pointer to status area	A1	???
A2	window manager vector	A2	???
A3	pointer to loose menu item	A3	???
A4	pointer to working defn	A4	preserved
A5	not used by any routine	A5	used as required
A6	not used by any routine	A6	used as required

Error returns:

D0 and the status register must be set

The pointer in D1 is in absolute (not window) coordinates. The uppercased keystroke in D2 also has SPACE ($20) converted to "hit" ($01) and ENTER ($0a) converted to "do" ($02) and all other event keystrokes converted to the event number less 14.

If the loose menu item was "hit" by a window event keystroke, then the event number (16 to 23) will be in D4. Otherwise D4 will be zero. The action routines may set the appropriate bit in the event vector as required or may return an event number in D4. However, WM.RPTR will only return to the calling routine if D4 is non-zero or the condition codes are non-zero - the event vector is not checked directly.

In the case of a loose menu item which causes an event, the action routine may derive the event number from the selection keystroke. All such loose menu items may be handled by the same code:

```
MOVEQ    #14,D4           set event number - event code
ADD.B    WWL_SKEY(A3),D4  add event code
MOVEQ    #0,D0            done
RTS
```

Pannable and Scrollable Sub-Windows

The window management routines have two views of pannable and scrollable windows. The first is the automatic pan and scroll operations within the routine WM.RPTR. These operations are caused by events occurring outside the application window. The second view is from the routine WM.MHIT which will cause pan or scroll operations from within a standard menu sub-window.

For either of these views, panning or scrolling will only be available if the appropriate part of the window working definition has been set up.

Any application may, of course, do its own panning or scrolling operations on a sub-window. It would be preferable if these operations were done in the same way as the window manager.

The values WWA_NXSC and WWA_NYSC define the pannablility and scrollability of a sub-window. If WWA_NYSC is 0, then the window is not scrollable, If it is 1, then the window is scrollable, but may not be split. If it is greater than 1, the window may be split into independently scrollable sections.

External Pan and Scroll

If a sub-window is set up to be scrollable, then the right hand border of the window is widened by 8 pixels to accommodate a "scroll bar". This scroll bar is 6 pixels wide and in two colours. The background bar represents the full "height" of the information being shown, superimposed on this is a shorter bar representing that part of the information which is actually visible.

A different section of the information may be viewed by "hitting" the scroll bar. "Hitting" the top of the scroll bar will scroll to the top of the information. "Hitting" the bottom of the scroll bar will scroll to the bottom, while "hitting" the middle will scroll to the middle.

As this bar is in the extended border of the sub-window, it is outside the sub-window and any "hit" in this area will not call the application sub-window hit routine. It will, instead, call the application sub-window control routine.

If the working definition has been set up so that there may be more than one vertical section, then the sub-window may be "split" by a "do" on the scroll bar. The scroll bar will also be split. Each section of the scroll bar represents the position of the visible information in the appropriate section of the sub-window. Conversely, a "do" on the break between two scroll bars will re-join the sections.

If a sub-window is set up to be pannable, then the bottom border is deepened by 5 pixels to accommodate a 4 pixel deep "pan bar". This functions in the same way as the scroll bar.

Internal Pan and Scroll

The standard menu hit routine WM.MHIT traps certain cursor movements as causing pan or scroll operations: these are ALT arrow to pan or scroll by one column or row at a time, and ALT SHIFT arrow to pan or scroll by the width or height of a section.

When a scrollable standard menu is drawn by WM.MDRAW, 4 pixel rows (plus the width of a current item border) are left vacant at the top and bottom of the sub-window. If there any rows above the topmost visible row, a row of up arrows is inserted at the top. If there are any rows below the bottommost visible row, then a row of down arrows is inserted at the bottom.

If a scrollable standard menu is split, then space is left at the split for two rows of arrows (separated by the width of a current item border).

If a row of up arrows is "hit", then the menu will scroll up by one item. If there is a "do" on a row of up arrows, then the menu will scroll up by the height of the section. The down arrows behave in a similar way.

When a pannable standard menu is drawn by WM.MDRAW, 8 pixel columns (plus twice the width of the current border) are left vacant at the left and right of the sub-window. These spaces are used for left and right arrows which have a similar function to the up and down arrows.

Sub-Window Indices

Standard menu sub-windows may have either a column or a row index (or both). These indices are outside the application sub-window and have no function except to convey information to the user. When a sub-window is panned or scrolled, the index will be updated at the same time.

To assist with panning and scrolling standard menu sub-windows, a single routine is provided to pan, scroll, split or join a standard menu.

Vector $38 **WM.PANSC**

Pan / Scroll standard menu

Call parameters		Return parameters	
D2.w	item number	D2	preserved
D3.l	position of "hit" or -1	D3	preserved
D4.b	pan or scroll event	D4.l	0
A0	window channel ID	A0	preserved
A1		A1	preserved
A2		A2	preserved
A3	ptr to sub-window definition	A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

D0 and the status register must be set

Window Move and Change Size

The size dependent layout features of the Window Manager mean that the interpretation of a window change size operation is largely the responsibility of the application. If the Window Manager returns from WM.RPTR with a window move or change size event, then the routine WM.CHWIN may be called directly.

This routine determines the event and the initial pointer position from the window status area and calls the appropriate window query trap. The event bit is cleared at this stage. In the case of a window move, the operation will be completed by WM.CHWIN and 0 is returned in D4.

In the case of a change size operation, WM.CHWIN will determine the distance moved by the pointer and return this as the change of size. If the convention that the window change size icon is in the top left hand corner of the window is being followed, then the move distance should be subtracted from the current window size. The window size event number is returned in D4.

Vector $40 **WM.CHWIN**

Change Window Event Handling

Call parameters		Return parameters	
D1		D1	x,y pointer move
		D2-D3	preserved
D4		D4.l	0 or pt..wsiz
A0		A0	channel ID of window
A1		A1	preserved
A2		A2	preserved
A3		A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors

Utility routines

The following routines are provided to modify the working definition in various useful ways; in particular, they may be used to show status information or get user input that is more complex than can be shown by item statuses or "point and hit" input.

If an information object or loose menu item object requires to be redrawn, then the vectored routines WM.IDRAW and WM.LDRAW can be used. Before redrawing, the objects themselves can be changed using one of the two following routines.

Vector $4C **WM.STLOB**

Set Loose Item Object

Call parameters		Return parameters	
D1	item number	D1	preserved
		D2+	all preserved
A0		A0	preserved
A1	pointer to object	A1	preserved
A2		A2	preserved
A3		A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

ORNG	Item number out of range

BEWARE: the item number is `NOT` the loose menu item number as defined in the loose menu item record, but is the position in the list (starting at zero).

WM.STIOB

Vector $50

Set Information Object

Call parameters		Return parameters	
D1	window number / object number	D1	preserved
		D2+	all preserved
A0		A0	preserved
A1	pointer to object	A1	preserved
A2		A2	preserved
A3		A3	preserved
A4	pointer to working defn	A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

ORNG   Window or object number out of range

The window number (MSW D1) is the position in the list of information sub-windows. The object number (LSW D1) is the position in the list of information objects for that window. Both start from zero.

The object pointed to by A1 in the above routines is not copied to a "safe place"by the routines. It is up to the programmer to ensure that it does not move or get overwritten while it is in use as part of a working definition. In particular, pointing to a string value on the SuperBASIC RI stack or in the variable values area will cause problems.

WM.RNAME
WM.ENAME

Vector $68 Read name
Vector $6C Edit name

Call parameters		Return parameters	
D1		D1.w	terminating character
		D2+	all preserved
A0	channel ID of window	A0	preserved
A1	pointer to name buffer	A1	preserved
A2		A2	preserved
A3		A3	preserved
A4		A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

Any I/O sub system errors
>0 if terminating character not <NL>

These two routines are used to read or edit strings (notionally file or device names). The name buffer is in the form of a standard string: a word with the string length, followed by the characters themselves. The difference between the two vectors is that WM.RNAME puts the cursor at the start of the name, and if the first character is printable, throws the old name away, while WM.ENAME leaves the cursor at the end of the name so that it has to be edited. Additionally, if the first character typed is a space, WM.RNAME will treat this as an ENTER.

The length of the name is limited to the width of the window and the name buffer must be large enough to accommodate this plus one character.

The routines return on reading ENTER, ESC, UP arrow or DOWN arrow. The condition codes will be set to -ve for an IO error, zero for ENTER or positive for other terminator.

This routine converts a small negative error code in D0 into the corresponding string; for instance, D0=-2 converts to "invalid Job". This code works for AH, JM, JS/JSU and all MG versions of the QL ROM - if other versions are used then the catch-all string "unknown error" is returned.

Vector $74

WM.ERSTR

Get string corresponding to error code

Call parameters		Return parameters	
D0	error code	D0	error code
		D1+	all preserved
A0		A0	preserved
A1		A1	pointer to error string
A2		A2	preserved
A3		A3	preserved
A4		A4	preserved
A5	not used by any routine		
A6	not used by any routine		

Error returns:

According to value of D0

Index of TRAPs and vectors

The Pointer Interface TRAPs and Window Manager vectors are listed alphabetically, along with a summary of what each does. Pointer Interface TRAPs start with the prefix `IOP.` and Window Manager vectors with `WM.`

Data Structures

Pointer Interface

Channel Definition block

The Pointer Interface forms the base level of the Pointer Environment and provides all those facilities which are accessed through the IO sub-system (IOSS). These include channel open, close and normal screen IO as well as the pointer IO extensions. The Pointer Environment uses this display driver which coexists with the standard CON and SCR drivers, and extends the CON and SCR drivers to handle overlapping windows. The extended driver requires an extended channel definition block, whose format is discussed here.

The `PTR_KEYS` file contains definitions of the symbols used when manipulating the extended channel definition block. Ordinary applications should not need to use these.

The facility to handle overlapping windows introduces the concept of piles of windows. Windows overlap each other in piles. Any window which is partly obscured by another window is locked and may not be altered. Windows may be moved to the top of the pile by the user, and applications may bury their own windows. Burying a window is actually performed by exhuming the bottom window in the pile. This will not actually bury the window unless the bottom window overlaps the top window. The internal structure used to maintain these piles is a bi-directional linked list of all primary windows. In addition, each primary window has a pointer to an area of memory in which to save its contents when it becomes locked, and a flag to signal whether the window is locked. For the sake of speed, the flag is duplicated in all its secondaries.

One of the major differences between the standard screen handling and Pointer Environment screen handling is the redirection of the keyboard input. Normally the "CTRL C" keystroke is used to redirect the keyboard input. With the Pointer Interface installed, the "CTRL C" keystroke is used to move windows to the top of the pile, redirecting the keyboard input as a side effect. This is achieved by modifying the normal circularly linked list of keyboard queues into a form that allows the detection of the "CTRL C" keystroke by the Pointer Interface. If the keyboard queue is moved to a job which is waiting for character input, then the pointer will be disabled, otherwise the pointer will be enabled. When the pointer is enabled, the cursor keys will move the pointer unless SHIFT, CTRL or ALT is pressed.

An alternative method of moving the window to the top of the pile may be used when the pointer is enabled. This is to move the pointer to part of a new window and "hit" it. If that window is buried, then the window will be picked to the top of the pile and the hit will be ignored. If the window is waiting for character input, then the pointer will be disabled and the hit will be ignored. The keyboard input will then be directed to that window.

To enable programs which have been written for use on a standard QL to function sensibly in the pointer environment, windows are divided into two types: primary and secondary. A primary window represents the total working area for an application. An application may have several secondary windows open, but all of these must be contained within the outline of the primary window. This introduces a new size concept. The standard screen driver in the QL has a window size and position: this is the window working area. The extended screen driver has two other sizes: the outline and the hit area. The outline is the limit enclosing all of an application's windows; Creating any window outside the application's primary window outline will cause the outline to be extended. The outline includes any window borders and shadows. The hit area is the area that the pointer routines will recognise for the purposes of hitting windows and selecting appropriate sprites. The hit area is the outline less any shadow area. The first window used for IO by an application is considered to be the primary window, any other windows owned by the same job are secondary windows. The outline and hit area are maintained in the extended channel definition block, along with a system of pointers linking primary windows to their secondaries, and all secondaries back to their primary.

The pointer routines may also make use of information in window definitions, so there is also a link to a window working definition.

Extended Channel Block

The pointer routines use an extended channel definition block. In order to make this compatible with the internal ROM code, the block is extended below the start of the standard block, but above the 18 byte channel block header.

sd.extnl	$30		screen definition extension length
sd_xhits	-$18	word	x hit size
sd_yhits	-$16	word	y hit size
sd_xhito	-$14	word	x hit origin (screen coordinates)
sd_yhito	-$12	word	y hit origin (screen coordinates)
sd_xouts	-$10	word	x outline size
sd_youts	-$0e	word	y outline size
sd_xouto	-$0c	word	x outline origin (screen coordinates)
sd_youto	-$0a	word	y outline origin (screen coordinates)
sd_prwlb	-$08	long	primary link list bottom up (primary window)
sd_pprwn	-$08	long	pointer to primary window (secondary window)
sd_prwlt	-$04	long	primary link list top down (primary window)
sd_sewll	$00	long	secondary window link list pointer
sd_wsave	$04	long	window save area base
sd_wssiz	$08	long	size of window save area
sd_wwdef	$0c	long	pointer to window working definition
sd_wlstt	$10	byte	window lock status -1 locked, 0 unlocked, 1 no lock
sd_prwin	$11	byte	bit 7 set for primary window, bit 0 set if managed (IOP.OUTL called)
sd_wmode	$12	byte	mode of this window
sd_mysav	$13	byte	true if save area is mine
sd_wmove	$14	byte	window move / query flag (D2 from IOP.RPTR)

Graphics objects

These base level data structures are used to pass information to the base level pointer IO calls. All these structures represent visual information. These structures have various forms, there is a canonical form and a screen mode dependent form. To simplify application programs, variations on the objects for various display modes can be linked into lists which future versions of the pointer traps will scan for the most suitable form. In current versions the pointer traps require the objects to be specified in the actual display mode for the window.

The file `QDOS_PT` contains symbol definitions suitable for use in programs that manipulate graphics objects.

All the structures are made from a limited set of basic elements.

Form

The form is a word which describes the screen dependent mode of the following patterns, followed by two bytes describing the mode adaption rules. The first of these is relevant only when the object is a sprite used as a pointer, and defines how it changes with time: the second defines how the object may be adapted to fit the display aspect ratio.

Dynamic pointers, that change shape with time, are used by setting the time byte to a non-zero value: by linking several sprite definitions together with increasing time values (Tn), the sprite will appear in the lowest numbered form for T1 "ticks", then change to the second form for T2-T1 ticks, then the third for T3-T2, and so on. When no sprite can be found with a Tn greater than the elapsed time, the counter is reset to zero and the first form appears again. The maximum value of Tn being 255, and the count being incremented (roughly) every 20ms, the sprite may have a period of up to 5 seconds or so.

Form		
	00fc	canonical, aspect ratio 1:.50
	00fd	canonical, aspect ratio 1:.60
	00fe	canonical, aspect ratio 1:.71
	00ff	canonical, aspect ratio 1:.83
	0000	canonical, aspect ratio 1:1.0
	0001	canonical, aspect ratio 1:1.2
	0002	canonical, aspect ratio 1:1.4
	0003	canonical, aspect ratio 1:1.7
	0004	canonical, aspect ratio 1:2.0
	0100	QL 4 colour
	0101	QL 8 colour
Time		
	00	static
	1..FF	used for time<n
Adaption		
	00	translate pixel to pixel
	+01	expand x if required
	+02	contract x if required
	+04	expand y if required
	+08	contract y if required

Size

The size of an object is defined by two words, the number of pixels in the x direction, and the number of pixels in the y direction. The only limit on the size is that it must be positive non zero in both directions.

Repeat

Some types of information have a repeat attribute. This is two words, the repeat distance (in pixels) in the x direction, and the repeat distance (in pixels) in the y direction. The y repeat must be positive non zero, the x repeat must be a positive non zero multiple of the number of pixels in a 16 bit word.

Origin

The base level structures assume a pixel coordinate system with the origin at the top LHS with x increasing to the right, y increasing downwards. Objects may have their own origin which is defined as two words, x origin and y origin. A negative origin is outside the object to the left (x) or above (y). A zero origin is the top left pixel of the object.

Colour

For the canonical forms (and possibly some other forms) it is assumed that colours are represented by a maximum of 15 bits (32768 colours). Notionally these are regarded as 5 bit resolution for each of the 3 primary colours. The 16th bit is used to indicate the opacity of the object. The order of bits is (MSB) green, red, blue, green/2, red/2, ... red/16, blue/16, opaque (LSB). For monochrome, the 15 most significant bits represent the display brightness.

Pattern

Canonical patterns are defined as colour planes. A canonical pattern starts with a word which defines the number of planes that will follow. The block defining each plane is preceded by a colour word defining the contribution of the following block to the complete colour. In every block of a canonical pattern each bit represents a pixel, the most significant bit in the first word is the top left pixel. Unused parts of words should be filled with zeros.

E.g. canonical form of yellow block (5x4) enclosing a black block (3x2)

```
    dc.w        2                           two blocks required
    dc.w        %1100000000000000           define yellow
    dc.w        %1111100000000000
    dc.w        %1000100000000000
    dc.w        %1000100000000000
    dc.w        %1111100000000000
    dc.w        %0000000000000001           define opaque
    dc.w        %1111100000000000
    dc.w        %1111100000000000
    dc.w        %1111100000000000
    dc.w        %1111100000000000
```

Specific form patterns are stored using the standard screen representation of the pattern. For this reason, there are two types of specific form pattern, the colour pattern, which is the colour representation, and the pattern mask which is white for opaque, and black for transparent. The base level routines require specific form patterns.

Sprite Definition

A sprite definition has form, size, origin, colour pattern and pattern mask.

form	2 words
size	2 words
origin	2 words
colour pattern	long word relative pointer
pattern mask	long word relative pointer
next definition	long word relative pointer

Blob Definition

A blob is used to provide a mask through which a pattern is dropped into the screen. The critical distinction is that while the pattern formed by a sprite moves with the sprite, the pattern used with a blob is stationary. The effect is akin to removing a bit of the screen to reveal the pattern underneath.

A blob definition, therefore, has only form, size, origin and pattern mask.

form	2 words
size	2 words
origin	2 words
colour pattern	long word zero
pattern mask	long word relative pointer
next definition	long word relative pointer

Pattern Definition

A pattern definition allows the specification of any pixel in the pattern to be any colour or transparent. The pattern repeats both horizontally and vertically. The pointer to the pattern mask may be given as zero, in which case the pattern is solid.

A pattern definition has form, repeat, colour pattern and pattern mask.

form	2 words
repeat	2 words
origin	2 words zero
colour pattern	long word relative pointer
pattern mask	long word relative pointer (or 0)
next definition	long word relative pointer

Area Mask

An area mask defines the limits of an area operation. The form is a table of x (horizontal) limits for each y coordinate. There may be more than one table. The total storage required is:

2 + 6*x_size + 4*(sum of y_sizes) bytes

The form of the definition is

x_size	number of tables
y_size	length of this table
x_origin	origin of sub-area within window
y_origin	
table	2*y_size words lower limit, upper limit pairs
....	(relative to x_origin)

The format of a partial save area is as follows:

spare	long	may be used by the application
flag	word	$4afc if this is a save area
x_size	word	width of save area in pixels
y_size	word	height of save area in pixels
increment	word	distance in bytes from one row to next
mode	byte	mode of saved image
spare	byte	zero
image	increment*y_size bytes	bit image

Window Definition

Structure

The window definition is split into several levels: at the top there is the window definition. Below this, there are the definitions of any loose menu items or sub-windows. Below these, there are the definitions of the object lists.

This section gives the standard meanings of the window definition structures. However, as it is the responsibility of the application's code to interpret the structures, the meanings may vary.

The file `WMAN_WDEF` contains definitions of the symbols used in this section: it may be INCLUDEd in any assembler files that manipulate window definitions.

Within these definitions all pointers are word length relative pointers. Where reference is to be made to an address which is more than a word offset away, the least significant bit is set. This (after clearing the bit) is then a pointer to a long word containing a relative address. All addresses are even. A zero pointer implies that the structure pointed to is absent.

In the following definitions, coordinates and sizes are specified as a pixel position or number of pixels. To allow for continuously variable window sizes, some coordinates and sizes can include terms to indicate the scaling of the coordinate or size with the variation in the appropriate dimension of the window. This is masked into the top nibble of the coordinate or size:

0000	invariant
0001	1:4 scaling wrt dimension
0010	1:2 scaling wrt dimension
0011	3:4 scaling wrt dimension
0100	directly coupled to dimension.

The rest of the word has the coordinate or size corresponding to the minimum allowable window dimension.

To allow for a variety of different layouts within the window as the size of the window varies, part of the window definition may be repeated several times. The definitions should be made in order of decreasing window size. The last definition, which defines the smallest allowable window, should be followed by a word containing -1. If the top nibble of a layout size word is zero, then the layout may not be scaled: if it is 0100 then it may.

Fixed part of window definition

wd_xsize	$00	word	default window x size (width) in pixels
wd_ysize	$02	word	default window y size (height) in pixels
wd_xorg	$04	word	pointer x origin in window
wd_yorg	$06	word	pointer y origin in window
wd_wattr	$08		window attributes
wd_psprt	$10	word	pointer to pointer sprite for this window
wd_lattr	$12		loose menu item attributes
wd_help	$2e	word	pointer to help window
wd_rbase	$30		base of repeated part of window definition

Repeated part of window definition

wd_xmin	$00	word	x (minimum) size for this layout + scaling flag
wd_ymin	$02	word	y (minimum) size for this layout + scaling flag
wd_pinfo	$04	word	ptr to information sub-window definition list
wd_plitm	$06	word	pointer to loose menu item list
wd_pappl	$08	word	ptr to application sub-window definition list
wd.elen	$0a		repeated entry length

The origin of the window is the initial pointer position within the window. This will usually also determine the position of the window itself as the window management level will try to avoid moving the pointer. If the origin is given as zero, then the origin will be calculated from the position of the current item.

The window width and height exclude the border and shadow, i.e. they refer to the inside of the window.

The XMIN and YMIN sizes are actual sizes of the window, unless the most significant bit is set in which case they are the minimum sizes.

Window Attributes

The window attributes for the window definition are four words defining a window clear flag, the shadow depth, the border and paper. For sub-windows, the shadow depth should be zero. For the main window the typical shadow depth will be 2, the actual x and y shadows will be derived from this. The top bit of the clear flag is used to define whether or not the (sub-)window should be cleared when it is (re-)drawn: if it is set then the window is not cleared.

wda_clfg	$00	byte	MSbit clear to clear window
wda_shdd	$01	byte	shadow depth
wda_borw	$02	word	border width
wda_borc	$04	word	border colour
wda_papr	$06	word	paper colour

Menu Item Attributes

To bring some semblance of order to the window organisation, all menu items within any one window or sub-window are constrained to have the same attributes. There is one set of attributes for each of the each of the three possible states of the item, and there is a border attribute to indicate the item currently pointed to.

wda_curw	$00	word	current item border width
wda_curc	$02	word	current item border colour
wda_unav	$04		item unavailable
wda_aval	$0c		item available
wda_selc	$14		item selected
wda.elen	$1c		menu item attribute entry length

attribute record

wda_back	$00	word	item background colour
wda_ink	$02	word	text object ink colour
wda_blob	$04	word	pointer to blob for pattern
wda_patt	$06	word	pointer to pattern for blob

Lower Level Definitions

Loose Menu Items List

Loose menu items can be positioned anywhere within the window. The loose menu item list is just a list of object types, positions, actions and pointers. The list is terminated by a word containing -1.

wdl_xsiz	$00	word	hit area x size (width) + scaling
wdl_ysiz	$02	word	hit area y size (height) + scaling
wdl_xorg	$04	word	hit area x origin + scaling
wdl_yorg	$06	word	hit area y origin + scaling
wdl_xjst	$08	byte	object x justification rule
wdl_yjst	$09	byte	object y justification rule
wdl_type	$0a	byte	object type (0=text, 2=sprite, 4=blob, 6=pattern)
wdl_skey	$0b	byte	selection keystroke (upper case)
wdl_pobj	$0c	word	pointer to object
wdl_item	$0e	word	item number
wdl_pact	$10	word	pointer to action routine
wdl.elen	$12		loose menu item list entry length

The selection keystroke should be the 'upper case' value for letters and the event code (not the event number) for the event keystrokes. The event code is the event number less 14. It may also be convenient for the item number to be the same as the selection keystroke/event code for these items. If the selection keystroke should be underscored (which is for text items possible), then the type is text-position. Thus, if you wish to underscore the third character, type is 0-3, giving -3.

Information Sub-Window

An information sub-window is set up when the menu is set up, but has no further significance. The definition of information sub-windows is in the form of a list terminated by a word containing -1.

wdi_xsiz	$00	word	sub-window x size (width) in pixels + scaling
wdi_ysiz	$02	word	sub-window y size (height) in pixels + scaling
wdi_xorg	$04	word	sub-window x origin + scaling
wdi_yorg	$06	word	sub-window y origin + scaling
wdi_watt	$08		sub-window attributes
wdi_pobl	$10	word	pointer to information object list
wdi.elen	$12		information list entry length

The information sub-window origin is the pixel position of the top left hand corner of the inside of the sub-window with respect to the top left hand corner of the window.

Information Object List

Each object in an information object list has only a limited set of attributes, and these may be different for each object. The list for each information sub-window is terminated by a word containing -1.

	wdo_xsiz	$00	word	object x size (width) in pixels + scaling
	wdo_ysiz	$02	word	object y size (height) in pixels + scaling
	wdo_xorg	$04	word	object x origin + scaling
	wdo_yorg	$06	word	object y origin + scaling
	wdo_type	$08	byte	object type (0=text, 2=sprite, 4=blob, 6=pattern)
	wdo_spar	$09	byte	spare = 0
(	wdo_ink	$0a	word	text ink colour type=0
(	wdo_csiz	$0c	word	text character size (two bytes)
or				
(	wdo_comb	$0a	word	pattern or blob to combine type=4,6
	wdo_pobj	$0e	word	pointer to object
	wdo.elen	$10		information object list entry length

Application Sub-window List

Because the size of an application sub-window definition is dependent on the usage of the definition, the application sub-window list is just a list of pointers to individual application sub-window definitions. The list is terminated with a zero word.

Menu Object Lists

Because menus are of indefinite size, the descriptions of the objects in a menu are put into lists so that these may be set up at execution time.

It is assumed, by the menu interface, that the objects are arranged in a rectangular grid. Each column of the grid has a fixed width, each row a fixed height. The interface also allows for an index to the columns and an index to the rows to be placed above and to the left of the grid.

There are two dimensions, the first is the actual number of columns, the second is the number of rows. All of the lists have either one dimension or the other.

Each of the object spacing lists consists of pairs of numbers. The first word is the hit area width or height. the second number is the distance from the start of this hit area to the start of the next. Both spacings are in pixels. There must be sufficient gap between the objects to allow the current item border to be drawn.

Each of the object index lists has the same form as the object list described below. The item numbers within these lists should be set to -1 and the action routine pointers to zero.

The object item lists consist of a set of list entries, one for each column in a row. Each object list entry contains the item number for the object, the object type (text, sprite etc.), the justification (left, right or centre, top, bottom or centre), a pointer to the actual object and a pointer to an action routine to be called when the object is hit. Note that it is possible to have just one large object list, which is 'cut up' into rows by making each row list start pointer equal to the previous row list end pointer.

The justification rule bytes are zero for a centered object, positive for left or top justified and negative for right or bottom justified. The value indicates the distance of the object, in pixels, from the edge of the hit area.

The row list consists of pairs of pointers to the start and end of each object list.

Application sub-window definition

wda_xsiz	$00	word	sub-window x size (width) in pixels + scaling
wda_ysiz	$02	word	sub-window y size (height) in pixels + scaling
wda_xorg	$04	word	sub-window x origin + scaling
wda_yorg	$06	word	sub-window y origin + scaling
wda_watt	$08		sub-window attributes
wda_pspr	$10	word	pointer to pointer sprite for this sub-window
wda_setr	$12	word	ptr to application sub-window setup routine
wda_draw	$14	word	ptr to application sub-window draw routine
wda_hit	$16	word	pointer to application sub-window hit routine
wda_ctrl	$18	word	ptr to application sub-window control routine
wda_nxsc	$1a	word	maximum number of x control sections
wda_nysc	$1c	word	maximum number of y control sections
wda_skey	$1e	byte	application sub-window selection keystroke
wda_ext	$1f	byte	zero
wda.blen	$20		application sub-window basic definition length

pannable and scrollable sub-windows only (wda_nxsc or wda_nysc <>0)

wda_part	$00	word	ptr to the part window control block (or 0) for pan, scroll and split definitions
wda_insz	$02	word	index hit size + scaling
wda_insp	$04	word	index spacing left or above sub-wind.+scaling
wda_icur	$06	long	index current item attr. (border width, colour)
wda_iiat	$0a		index item attribute record
wda_psac	$12	word	pan or scroll arrow colour
wda_psbc	$14	word	pan or scroll bar background colour
wda_pssc	$16	word	pan or scroll bar section colour
wda.clen	$18		applic. sub-window control definition length

menu sub-windows only (processed by WM.SMENU called from application setup)

wda_mstt	$00	word	pointer to menu status block
wda_iatt	$02		item attributes
wda_ncol	$1e	word	number of actual columns
wda_nrow	$20	word	number of actual rows
wda_xoff	$22	word	x offset to start of menu (section)
wda_yoff	$24	word	y offset to start of menu (section)
wda_xspc	$26	word	pointer to x (column) spacing list
wda_yspc	$28	word	pointer to y (row) spacing list
wda_xind	$2a	word	pointer to x (column) index list
wda_yind	$2c	word	pointer to y (row) index list
wda_rowl	$2e	word	pointer to menu row list
wda.mlen	$30		sub-window menu definition length

The application sub-window origin is the pixel position of the top left hand corner of the inside of the sub-window with respect to the top left hand corner of the window.

The pointers to the sub-window pan and scroll control blocks and the menu status block are relative to the start of the window status area.

If a window is both pannable and scrollable, then there should be two complete sub-window control definitions.

If a spacing list consist of items of the same size, then the pointer to the spacing list may be replaced by the negative spacing values.

menu object spacing list

wdm_size	$00	word	object hit size + scaling
wdm_spce	$02	word	object spacing + scaling
wdm.slen	$04		object spacing list element length

menu row list

wdm_rows	$00	word	pointer to object row list start
wdm_rowe	$02	word	pointer to object row list end
wdm.rlen	$04		menu row list element length

menu object / index list entry

wdm_xjst	$00	byte	object x justification rule
wdm_yjst	$01	byte	object y justification rule
wdm_type	$02	byte	object type (0=text, 2=sprite, 4=blob, 6=pattern)
wdm_skey	$03	byte	selection keystroke (upper case)
wdm_pobj	$04	word	pointer to object
wdm_item	$06	word	item number (-1 for index)
wdm_pact	$08	word	pointer to action routine (zero for index)
wdm.olen	$0a		menu object / index list entry length

Menu Macros

This section documents the action of the utility macros supplied in the file WMAN_MENU_MAC. These macros assist in the generation of standard format Window Definitions by automatically generating the XDEF and XREF directives required to use the definition: they also relieve the programmer of the burden of remembering the size of each data item.

Most symbols generated by these macros have a four character prefix showing their type. This means that in the user-supplied symbol, usually referred to as the name, only the first four characters will be significant.

There is, of course, no need to use these macros to generate Window Definitions: in particular, any constraint of size and label name is imposed only by these macros, and not by the data structures themselves. Modification of the macros, or direct generation of the definition, is definitely recommended if you can't get the effect you want.

Structure

The major data structure produced by the macros is the Window Definition. This is of the form documented in the previous section of this manual, and is thus appropriate for conversion to its Working Definition by the WM.SETUP routine of the Window Manager. Each of an application's Window Definitions has a unique name, and may be referred to by using the label MEN_name which is XDEFfed by the WINDOW macro, and may be XREFfed where required.

A Window Definition consists of one or more layouts, each appropriate for a different size of window. One of these is selected by the WM.SETUP routine for copying into the Working Definition, depending on the size requested. Each layout is given a unique letter when introduced by the SIZE_OPT macro: when the SETWRK macro is invoked at the end of the menu assembly, symbols of the form WWletter.name are XDEFfed, defining the space required for the Working Definition for each layout. These may be referred to in other modules by declaring the symbol with an XREF.S directive. Different layouts for a window may be put in different files: the main definition is introduced with the WINDOW macro, and has the various layouts introduced with the SIZE_OPT macro: the external layout definition(s) start with the XLAYOUT macro, and define the layouts specified by calls to the LAYOUT macro.

In addition to creating the Window Definition, the macros also keep track of the size of Status Area required. In principle, the statuses of the items in a window may be static, so that when the window is pulled down again previously selected options are still selected. To cater for this, the status blocks for a given window are defined as COMMON blocks of the required size: each layout defines its own blocks, but with the same name, so that when linked the largest version of each COMMON block is used. One COMMON block is defined for the base area and loose item status block, one for each menu status block and control block, and one for each item allocated space with a call to the ALCSTAT macro. By using the COMMON DUMMY option in the linker command file, no space is allocated in the application for the status areas, resulting in ROMable code. The global status area for all windows may then be put in the application's data space, if this is big enough, or in a suitably-sized piece of heap allocated when the application starts. If this area is always pointed to by Ax, then the status area for a given window will be found at WST_name(Ax), this label having been defined by an XREF.S directive. Note that this limits you to a maximum global status area size of 32k. Often A5 or A6 will be used to point to the global status area, as they are not used by the Window Manager.

Rules and reserved symbols

Within the body of a description, the macro substitution syntax of `[name]` is used where the value of the variable or macro parameter `name` is meant: in general, macro parameters are in `Courier` and global variables in `UPPER_CASE`. New variables and labels may be created from global and local variables: for instance, the `ACTION` macro is of the form:

```
ACTION   MACRO    name
         ...
         XREF     MEA_[name]
         ...
         ENDM
```

An invocation of this macro might be:

```
         ACTION   QUIT
```

producing the expansion:

```
         ...
         XREF     MEA_QUIT
         ...
```

At the start of a definition, the square brackets take their usual meaning of defining an optional parameter.

The variables `CLAYOUT`, `CURRA`, `CURRW`, `MAXITEM` and `WSIZES` are used by the macros, and should not be used for other purposes.

The prefixes shown overleaf are used by the macros, for the purposes specified. In general, you should avoid using any symbol with these prefixes in your own code. Those marked external are XDEFfed or XREFfed by the macros. Those marked var(iable) are used as assembler variables to keep track of which layout(s) the corresponding object is used in.

Prefix	External	Var	Use
MAD_			Label for application sub-window definition
		Y	Layouts using this sub-window
MAW_	Y		Label for application sub-window list
		Y	Layouts using this application sub-window list
MEK.	Y		Value of item select key
MEA_	Y		Label of externally defined code:
MEC_			this may be an Action/Hit, Control,
MED_			Drawing or Menu-setup routine.
MEM_			
MEB_	Y		Label of externally defined objects:
MEP_			these may be a Blob, Pattern,
MES_			Sprite or Text.
MET_			
MIO_			Label for an info. object list
		Y	Layouts using this list
MIW_			Label for an info. sub-window list
		Y	Layouts using this list
MLI_			Label for a loose item list
		Y	Layouts using this list
MOB_			Label for menu sub-window or (first) index object
		Y	Layouts using this object
MPS_	Y		Label for externally-accessible co-ordinates
MRW_			Label for menu sub-window row list
		Y	Layouts using this row list
MST_Y			Offset of menu sub-window status block from start of global status area
MSX_			Label for X or Y spacing list
MSY_		Y	Layouts using this spacing list
MV_	Y		Label for space in global status area allocated by ALCSTAT macro
NCX.		Y	Number of control sections in the
NCY.			X or Y direction for a menu sub-window
WAL_		Y	Start of ALCSTAT area in global status area Variable holds running total of space needed
WCX_	Y		Offset of X or Y section control block
WCY_			from start of global status area
WST_	Y		Offset of window status area from start of global status area
WWx.	Y		Size of Working Definition needed for layout x

The macros defined in the WMAN_MENU_MAC file are listed below.

ACTION name
Generates a relative pointer to an action routine. This is external to the menu definition, and should have the label MEA_[name].

ALCSTAT name,space
This reserves some extra space in the global status area, which can be accessed at the offset MV_[name] from the base of this global status area: this offset will always be even. The amount of space reserved is given by the value of the space parameter. The offset should be referred to in the code by using the XREF.S directive.

APPN name
Generates a relative pointer to the application sub-window list for this layout. This should have the label MAW_[name] and will have an XREF generated for it if CLAYOUT has the value "*", which implies an externally-defined layout.
If CLAYOUT does not have the value "*", then a variable with the name MAW_[name] is updated: if it already exists, then this application sub-window list is used by several layouts, and the value of CLAYOUT is appended to it. If the variable is undefined, then it is initialised to the current value of CLAYOUT.

ARROW colour
Define the colour of the arrows in the pan or scroll arrow rows.

A_CTRL name,dirn
Introduces an application sub-window control definition, defining a pointer, relative to the start of the window status area, where the section control block starts, and generating an externally accessible offset WC[dirn]_[name] which may be used by coding a suitable XREF.S directive in the code wishing to use it. The size of section control block is given by the maximum number of sections, which will have been previously defined by a call to the CTRLMAX macro, and kept in the variable NC[dirn]_[name].

A_END
This generates the termination for an application sub-window list: it is not interchangeable with I_END etc., as the terminators are different.

A_OBJE name
This marks the end of a menu sub-window object list, defining the label MOB_[name] so that the row list can point to the end of the list. It also defines a COMMON block for the menu item statuses, which may be found at the offset MST_[CURRA] from the base of the global status area: [CURRA] is the name of the application sub-window currently being defined.

A_MENU
Introduces the menu definition section of an application sub-window, and generates a relative pointer to the menu status block.

A_RLST name
This introduces a menu sub-window row list, and labels it MRW_[name]. It also sets the value of CLAYOUT to the value of the variable MRW_[name].

A_SLST name,dirn

This introduces a menu sub-window spacing list, and labels it MS[dirn]_[name]. It also sets the value of CLAYOUT to the value of the variable MS[dirn]_[name].

The parameter dirn may take the values "X" or "Y".

A_WDEF name

This introduces an application sub-window definition, and labels it MAD_[name]. It also sets the value of CLAYOUT to the value of the variable MAD_[name], and CURRA to [name].

A_WINDW name

This generates a relative pointer to an application sub-window definition, which must be internal to this layout. The label used is MAD_[name], this being generated by the A_WDEF macro. A variable MAD_[name] is also set to the current value of CLAYOUT.

A_WLST name

This macro introduces an application sub-window list. It generates a **label** MAW_[name] and reads a new value for the variable CLAYOUT from the **variable** MAW_[name], which will have been defined by a call to APPN or LAYOUT.

The effect of this is to ensure that the list can be pointed to from elsewhere in the definition, and that the space required for the application sub-windows can be added up in the appropriate layout variable.

BAR background,block

Define the colours of the "thermometer" bar to the right or bottom of an application sub-window. The visible part of the window is represented as a bar of the block colour, on a bar representing the whole height or width of the menu, of the background colour.

BLOB name

Generates a relative pointer to a blob definition. This is external to the menu definition, and should have the label MEB_[name].

BORDER size,colour

Generates the definition of a border to be put around an item when the pointer is pointing to it. Usually followed by one or three IATTR definitions defining the attributes of the item itself.

CSIZE xsize,ysize

This defines the character size for an information item: the usual range of xsize from 0 to 3 and ysize from 0 to 1 applies.

CTRL name

Generates a relative pointer to an application sub-window control routine. This is external to the menu definition, and should have the label MEC_[name].

`CTRLMAX xsects,ysects`
This defines the maximum number of sections into which an application sub-window may be split. It also keeps a record of these numbers in the variables `NCX.[CURRA]` and `NCY.[CURRA]`, so that when the control definition is encountered the correct amount of space can be allocated in the status area.

`DRAW name`
Generates a relative pointer to an application sub-window drawing routine. This is external to the menu definition, and should have the label `MED_[name]`.

`HELP label`
Generates a relative pointer to the help definition. Since the meaning of this pointer is dependent on the application, the `label` is used directly, without adding a prefix: the label is assumed to be external, so an XREF is generated.

`IATTR paper,ink,blob,pattern`
Generates part of a definition of the attributes to be used when drawing loose menu items, index items or menu sub-window items. The `blob` and `pattern` are external, with labels `MEB_[blob]` and `MEP_[pattern]` respectively. Loose and sub-window items should have three sets of attributes, one for each of the three possible statuses unavailable, available and selected. Index items do not have variable status, so only need one set of attributes. The object to be drawn is combined with one or more of the attributes, depending in its type:

Object type	Attribute			
	paper	ink	blob	pattern
TEXT	Y	Y		
SPRITE	Y			
BLOB	Y			Y
PATTERN	Y		Y	

`IBAR size,spacing[,szscale,spscale]`
Define the size and spacing of an index bar. Optionally these may be scaled. The spacing is measured above or to the left of the application sub-window.

`ILST name`
Generates a pointer to an index object list, which is internal to the definition and must be labelled `MOB_[name]`. The variable of the same name is given the value of `CLAYOUT`.

`INFO name`
Generates a relative pointer to the information sub-window list for this layout. This should have the label `MIW_[name]` and will have an XREF generated for it if `CLAYOUT` has the value "*".
If `CLAYOUT` does not have the value "*", then a variable with the name `MIW_[name]` is updated in the same way as in the `APPN` macro.

`INK colour`
This macro defines the ink colour for an information item.

`ITEM number`

Defines the item number for a loose or menu object: more than one object may share an item number, in which case they will share a status byte and therefore all be drawn with the same status.

If the value of the variable `CURRA` is not "*", then it is assumed that the object being defined is in a menu sub-window, and the maximum item number for that sub-window is updated if required, this being kept in the variable `MST_[CURRA]`: otherwise the variable `MAXITEM` is updated. In this way it is possible to have "holes" in the tiem numbers, but still get the correct size of status area allocated.

`I_END`

Generates an end-of-list marker for information sub-window and object lists.

`I_ITEM`

This introduces an information item: it is this macro that adds to the space requirements for the current layout(s), given by the value of the variable `CLAYOUT`.

`I_OLST name`

This introduces an information object list, generating a label `MIO_[name]`. The variable `CLAYOUT` is set to the value of the variable `MIO_[name]`.

`I_WINDW`

This introduces an information sub-window: it is this macro that adds to the space requirements for the current layout(s), given by the value of the variable `CLAYOUT`.

`I_WLST name`

This macro introduces an information sub-window list. It generates a **label** `MIW_[name]` and reads a new value for the variable `CLAYOUT` from the **variable** `MIW_[name]`.

The effect of this is to ensure that the list can be pointed to from elsewhere in the definition, and that the space required for the information sub-windows can be added up in the appropriate layout variable.

`JUSTIFY xjst,yjst`

Define the justification required for an item: an item may be centred in the area available or be positioned a fixed distance from either margin. A parameter value of zero requests a centred object, a positive non-zero value is an offset from the left or top, and a negative value an offset from the right or bottom.

`LAYOUT letter,[info],[loos],[appn]`

This specifies one of the layouts that is to be defined in this file, in a similar way to the `SIZE_OPT` macro, but is used in a separate layout file, after the `XLAYOUT`. It should not be used in a main definition file.

The names of the information sub-window list, loose item list and application sub-window list may be omitted if the layout does not contain such a list, but the commas must be coded so that the correct internal labels are generated.

`LOOS name`

Generates a relative pointer to the loose item list for this layout. This should have the label `MLI_[name]` and will have an XREF generated for it if `CLAYOUT` has the value "*".

If `CLAYOUT` does not have the value "*", then a variable with the name `MLI_[name]` is updated in the same way as in the `APPN` macro.

`L_END`

Terminates a loose item list, and generates a `COMMON` block definition for a window status area big enough for the maximum loose item number, given in the `MAXITEM` variable.

`L_ILST name`

This macro introduces a loose item list. It generates a **label** `MLI_[name]` and reads a new value for the variable `CLAYOUT` from the **variable** `MLI_[name]`. In addition, the variable `MAXITEM` is initialised to zero, and `CURRA` to "*".

The effect of this is to ensure that the list can be pointed to from elsewhere in the definition, and that the space required for the loose items will be added up in the appropriate variable.

`L_ITEM [name,number]`

This introduces a loose item: it is this macro that adds to the space requirements for the current layout, given by the value of the variable `CLAYOUT`. If `name` and `number` are supplied, a label `MLI.[name]` is defined and set to the value of `number`, also a label `MLO.[name]` which is the position of the item in the list, counting from 0.

`MENSIZ ncols,nrows`

This defines the size of a menu sub-window in terms of rows and columns, and therefore the sizes of the spacing lists, index item lists (if present), and row list.

`OBJEL [name]`

Introduces a menu sub-window object definition: if the `name` is supplied then the object is given the label `MOB_[name]` and `CLAYOUT` is given the value of the `MOB_[name]` variable.

`OLST name`

Generate a relative pointer to an information object list. This must be internal to the definition, and have the label `MIO_[name]`. A variable of the same name is defined to have the same value as the variable `CLAYOUT`, so that the space occupied for the object list can be attributed to the appropriate layout.

`ORIGIN xpos,ypos[,xscale,yscale]`

Generates a two word origin definition for a window, sub-window or object. A window's origin specifies the point within it where the pointer should be placed when the window is drawn - this will be combined with the current pointer position to decide the absolute origin of the window.

The origin of a sub-window or object is always specified relative to the window containing it.

Optionally a scale factor may be provided to specify how the origin should be changed if the window is bigger than expected. See the Window Definition section of the Data Structures for details on how scale factors work.

`PATTERN name`

Generates a relative pointer to a pattern definition. This is external to the menu definition, and should have the label `MEP_[name]`.

`POSN name,xsize,ysize[,xscale,yscale]`

Generates a scaled co-ordinate pair in the same way as the `ORIGIN` macro, and labels the data `MPS_[name]`. This label is XDEFfed so that the co-ordinates can be used from other parts of the program.

`ROWEL start,end`

Generate one element of a row list, consisting of a pair of relative pointers to the start and end menu sub-window objects: the `start` pointer points to the first object, the `end` points just after the last. The labels used must be internal to the definition, and have the symbols `MOB_[start]` and `MOB_[end]`. Two variables of the same names are given the current value of the `CLAYOUT` variable.

`RLST name`

Generates a relative pointer to a rowlist, which is internal to the definition and must be labelled `MRW_[name]`. A variable of the same name is given the current value of `CLAYOUT`.

`SELKEY [name]`

Generate a select key for a loose or menu item. The value of the select key is an external symbol `MEK.[name]`: this allows the programmer to have one file containing all select keys (and text), which is then the only file that needs to be changed to make foreign language versions of the program.

If `name` is not supplied, a select key of 0 is defined, which can never occur (it is trapped out by the Window Manager).

`SETR name`

Generates a relative pointer to an application sub-window setup routine. This is external to the menu definition, and should have the label `MEM_[name]`.

`SETWRK`

This macro must always be coded at the very end of a window or layout definition: it defines the external symbols giving the space required for the working definitions of the various possible size-dependent layouts. In addition it generates a `COMMON` section declaration and external definition for any extra space required in the global status area as a result of calls to `ALCSTAT`.

`SIZE xsize,ysize[,xscale,yscale]`

Generates a two-word size definition for a window, sub-window or object. The size of a window is the actual area that can be used, any border defined is added to the outside.

Optionally a scale factor may be provided to specify how the size should be changed if the window is bigger than expected. See the Window Definition section of the Data Structures for details on how scale factors work.

`SIZE_OPT letter | *`

This introduces an entry in the repeated part of the window definition: each entry gives a possible size that the window can have, and pointers to the various parts of the layout for this size.

The value of the parameter is kept in the variable `CLAYOUT` for future use.

If the `*` option is coded, the layout is assumed to be external, and XREFs will be generated for the pointers to the loose item list, information sub-window list, and application sub-window list.

If a `letter` is coded, then the layouts are assumed to be in the current file. In this case the variables `WW[letter].[CURRW]` and `WS[letter].[CURRW]` are initialised to suitable values: these are used during the later stages of the menu definition to calculate the sizes required for the working definition and status area for this layout. The `[letter]` is also appended to the `WSIZES` variable.

`SOFFSET xoff,yoff`

This defines the offset of the top left object from the top left of a menu sub-window, so you don't have to squash everything up into the top left corner.

`SPARE`

Generates a null byte to fill up spare space. Only required after the definition of an application sub-window's select key.

`SPCEL gap,size`

This generates one element of a row or column spacing list, defining the horizontal or vertical hit size of a column or row, and the gap between the column or row and the next.

`SLST xnam,ynam`

This generates two relative pointers to the X and Y spacing lists, which should be labelled `MSX_[xnam]` and `MSY_[ynam]`. Two variables of the same names are set to the current value of `CLAYOUT`.

`SPRITE name`

Generates a relative pointer to a sprite definition. This is external to the menu definition, and should have the label `MES_[name]`.

`S_END`

Terminates the list of layouts in the repeated part of a window definition.

`TEXT name`

Generates a relative pointer to a string. This must be external to the menu definition, and should have the label `MET_[name]`. This allows the programmer to have one file containing all text (and select keys), which is then the only file that needs to be changed to make foreign language versions of the program.

`TYPE code`

Specifies the type of a loose, information or menu object. The value of `code` may be 0 for a text item, 2 for a sprite and so on: suitable symbols are defined in the `WMAN_KEYS` file.

`WATTR shadow,border_size,border_colour,paper`

Generates data describing the overall colour of a window or sub-window. The `shadow` is ignored in the case of sub-windows. The `border_size` is added to the specified window size.

`WINDOW name`

Generates an externally accessible label `MEN_[name]` which points to the Window Definition.

The variable `CURRW` is set to `[name]` so that various unique symbols may be defined and XDEFfed at a later stage.

The variable `WSIZES` is set to the null string: this is added to by `SIZE_OPT`, and used in `SETWRK` to generate XDEFs for each possible size.

`XLAYOUT name`

This introduces a set of layout definitions in a similar way to `WINDOW` introducing the main part of a window definition. It is associated with the appropriate main definition by having the same `[name]`, which is assigned to the `CURRW` variable as in `WINDOW`.

Text Macros

The file `WMAN_TEXT_MAC` contains a set of macros which are used for defining text strings, often for use in menus. Several different flavours are provided, depending on the use to which the text is going to be put. The merit of this approach is that all text used in an application may be put into one file, and different versions of this file with the text in different languages linked with the rest of the application (all of which should be language-independent) to produce foreign language versions.

All the macros take one or more `string` parameters. Each of these should consist of of the characters you wish to appear in the text, enclosed in braces {}. This is a convention used by the GST Macro Assembler to allow the use of strings with spaces in them as macro parameters. All the macros use this parameter to generate a QDOS format string at an even address with a 1-word character count at the beginning.

Note that you cannot use the open square bracket character "[" either within a string or as a select key when you are using the GST Macro Assembler, as this character is `always` interpreted as the beginning of a macro substitution. If you do need to use the open square bracket, you will need to code the ASCII value (91 or $5B) in a DC.x directive of your own making.

The `MKTEXT` macro uses the variables `MKT.PRM` and `MKT.PRMX`, so you should avoid using these variables when using the text generating macros.

Label and variable prefixes used by these macros are as follows:

Prefix	External	Var	Use
MEK.	Y		Item select key definition
MET_	Y		Text string label
MET.	Y		Text string length/2 in pixels

In the following macro definitions, square brackets in the heading line enclose an optional parameter, braces enclose a parameter that may be repeated more than once. Within the body of a definition, the square brackets signify the `value` of a supplied parameter: see the beginning of the previous section for an example.

`MKSELK label,selkey`

Generates an external symbol `MEK.[label]` whose value is that given by the one character string passed in `selkey`. If the character was in the range "a" to "z" then the upper case equivalent is used, as select keys are required to be defined in upper case. This macro is of use when defining a select key for a graphics object such as a sprite.

`MKSTR string`

This is the simplest of the macros. It generates a QDOS string but no extra information.

`MKTEXT label{,string}`

This macro is used to generate a large block of text which has to be defined over many lines of source code. The resulting single string is labelled `MET_[label]`. All parameters after the `label` name should be strings enclosed in braces, and these are concatenated to produce the result. If you wish to force a newline at any point then you may code a backslash character "\" as the last character of any string - this will then be translated into a newline character (ASCII value 10 or $0A). A backslash within a string is not translated.

`MKTITL label,string`

Generates a string for use as a large title. Two external symbols are defined, `MET_[label]` labels the string itself and `MET.[label]` gives half the length of the string, in pixels, if written out with CSIZE 2,n. This symbol may be referred to by an XREF.S directive and used to centre the title in an information sub-window. Another macro is used for strings written with a smaller character size, as the GST Macro Assembler does not allow multiplication or division of externally-defined symbols.

`MKTITS label,string`

Generates a string for use as a small title. Two external symbols are defined, `MET_[label]` labels the string itself and `MET.[label]` gives half the length of the string, in pixels, if written out with CSIZE 0,n. This symbol may be referred to by an XREF.S directive and used to centre the title in an information sub-window. Another macro is used for strings written with a larger character size, as the GST Macro Assembler does not allow multiplication or division of externally-defined symbols.

`MKXSTR label,[selkey],string`

Generates a string for use as a loose menu item or menu object. The string itself is defined as usual, with the symbol `MET_[label]` being used to refer to it. Optionally a select key may be defined by specifying a non-null value for the `selkey` parameter. This should be a one character string, preferably enclosed in braces for consistency. If supplied, the symbol `MEK.[label]` is defined to have the value of this character: if the character is in the range "a" to "z" then the upper case equivalent will be used.

Index of macros

The macros are summarised in alphabetical order, together with which file they are defined in and a short description of the structure each generates. Those marked MENU are in the file `WMAN_MENU_MAC`, those marked TEXT are in the file `WMAN_TEXT_MAC`.

ACTION	MENU	pointer to action routine
ALCSTAT	MENU	space in global status area
APPN	MENU	pointer to application sub-window list
ARROW	MENU	arrow colour for pan/scroll bars
A_CTRL	MENU	start of control definition
A_END	MENU	end of application sub-window list
A_MENU	MENU	start of menu definition
A_OBJE	MENU	end of menu object list
A_RLST	MENU	start of menu row list
A_SLST	MENU	start of menu spacing list
A_WDEF	MENU	start of application sub-window
A_WINDW	MENU	pointer to application sub-window
A_WLST	MENU	start of application sub-window list
BAR	MENU	pan/scroll "thermometer" colours
BLOB	MENU	pointer to blob
BORDER	MENU	border size and colour for current item
CSIZE	MENU	character size for information text
CTRL	MENU	pointer to control routine
CTRLMAX	MENU	maximum number of control sections
DRAW	MENU	pointer to sub-window drawing routine
HELP	MENU	pointer to help definition
IATTR	MENU	item status attributes
IBAR	MENU	size and spacing of index items
ILST	MENU	pointer to index item list
INFO	MENU	pointer to information sub-window list
INK	MENU	ink colour for information text
ITEM	MENU	item number for loose or menu item
I_END	MENU	end of information window or object list
I_ITEM	MENU	start of information object
I_OLST	MENU	start of information object list
I_WINDW	MENU	start of information sub-window
I_WLST	MENU	start of information sub-window list
JUSTIFY	MENU	justification rules for loose or menu item
LAYOUT	MENU	start of external layout definition
LOOS	MENU	pointer to loose item list
L_END	MENU	end of loose item list
L_ILST	MENU	start of loose item list
L_ITEM	MENU	start of loose item
MENSIZ	MENU	size of menu in columns/rows
MKSELK	TEXT	item select keystrokes
MKSTR	TEXT	QDOS string, no label
MKTEXT	TEXT	multi-line text
MKTITL	TEXT	large title string
MKTITS	TEXT	small title string
MKXSTR	TEXT	external string with select keystroke
OBJEL	MENU	start of menu object definition
OLST	MENU	pointer to information object list
ORIGIN	MENU	origin of window or object
PATTERN	MENU	pointer to pattern
POSN	MENU	externally-accessible ORIGIN
ROWEL	MENU	row list element
RLST	MENU	pointer to row list
SELKEY	MENU	select keystroke for loose or menu item

SETR	MENU	pointer to setup routine
SETWRK	MENU	end of entire window definition
SIZE	MENU	size of window or object
SIZE_OPT	MENU	start of internal layout definition
SOFFSET	MENU	offset from top left of menu sub-window
SPARE	MENU	spare padding byte
SPCEL	MENU	spacing list element
SLST	MENU	pointers to spacing lists
SPRITE	MENU	pointer to sprite
S_END	MENU	end of layout list
TEXT	MENU	pointer to text
TYPE	MENU	object type
WATTR	MENU	overall window attributes
WINDOW	MENU	start of entire window definition
XLAYOUT	MENU	start of external layout definitions

Working Definition

To allow a very large degree of flexibility in the handling of windows and menus, the actual definition of a window used by the window management routines is set up during execution. Because this definition will usually be set up before pulling down a window, and discarded after throwing the window away, this is referred to as the working definition.

The window definition is principally a definition of a pull-down window. It may, however, include definitions of menus within the window. The window working definition is a copy of the window definition, with the addition of the definitions of menus whose contents are defined at execution time. The form of the working definition is chosen to simplify menu handling.

Within a window, it is likely that sub-windows will exist which are either menus in a non-standard form, or not menus at all. In either of these cases the corresponding part of the window working definition may be absent or of non-standard form.

Within the working definition all pointers are long word absolute pointers. All addresses are even. A zero pointer implies that the structure pointed to is absent.

The file `WMAN_WWORK` contains definitions of the symbols used in this section: it may be INCLUDEd in any assembler files that manipulate working definitions.

The working definition starts with a header block. This has three functions: the first is to save the window channel ID, the original window definition address and the window status area address; the second is to point to the pointer record, to save the pointer position as it was before the window was opened, and to flag whether the window is a primary or a pull-down (secondary); the third is to provide the sprite list for the base level of the pointer interface.

ww_wstat	$00	long	pointer to window status area
ww_wdef	$04	long	pointer to window definition
ww_chid	$08	long	channel ID for window
ww_pprec	$0c	long	pointer to pointer record (24 bytes)
ww_psave	$10	long	saved pointer position (absolute coordinates)
ww_spar1	$14	long	window spare 1
ww_spar2	$18	word	window spare 2
ww_spar3	$1a	byte	window spare 3
ww_pulld	$1b	byte	flag, <>0 if pulled down
ww_splst	$1c	long	pointer to sub-window sprite list

The channel ID is set when the window is opened by the window open routine.

The pointer position is saved when the window is opened, and restored when the window is thrown away.

The header block is immediately followed by the window definition block:

ww_xsize	$20	word	window x size (width) in pixels
ww_ysize	$22	word	window y size (height) in pixels
ww_xorg	$24	word	pointer x origin in window
ww_yorg	$26	word	pointer y origin in window
ww_wattr	$28		window attributes
ww_psprt	$30	long	pointer to pointer sprite for this window
ww_lattr	$34		loose menu item attributes
ww_help	$5c	long	pointer to help definition
ww_head	$60		end of header
ww_ninfo	$60	word	number of information sub-windows
ww_ninob	$62	word	number of information sub-window objects
ww_pinfo	$64	long	ptr to information sub-window definition list
ww_nlitm	$68	word	number of loose menu items
ww_plitm	$6a	long	pointer to loose menu item list
ww_nappl	$6e	word	number of application sub-windows
ww_pappl	$70	long	ptr to application sub-window definition list
ww_lists	$74		start of definition lists

The window width and height exclude the border and shadow, i.e. they refer to the inside of the window.

The origin of the window is the position of the top left hand corner of the inside of the window is display coordinates.

Window Attributes

The window attributes for the working definition are identical to those for the window definition.

wwa_clfg	$00	byte	MSbit set to clear window
wwa_kflg	$00	byte	Bit 0 set disables keys moving the mouse
wwa_shdd	$01	byte	shadow depth
wwa_borw	$02	word	border width
wwa_borc	$04	word	border colour
wwa_papr	$06	word	paper colour

Menu Item Attributes

The menu item attributes for the working definition are similar to those for the window definition. They occupy rather more space as they use long word pointers.

wwa_curw	$00	word	current item border width
wwa_curc	$02	word	current item border colour
wwa_attr	$04		attribute records
wwa_unav	$04		item unavailable
wwa_aval	$10		item available
wwa_selc	$1c		item selected
wwa.elen	$28		menu item attribute entry length

attribute record

wwa_back	$00	word	item background colour
wwa_ink	$02	word	text object ink colour
wwa_blob	$04	long	pointer to blob for pattern
wwa_patt	$08	long	pointer to pattern for blob
wwa.alen	$0c		attribute record length

Loose Menu Items List

Loose menu items can be positioned anywhere within the window. The loose menu item list is just a list of object types, positions, actions and pointers. The list is terminated by a word containing -1. Apart from the use of long word pointers, the loose menu item list is the same as in the window definition.

wwl_xsiz	$00	word	hit area x size (width)
wwl_ysiz	$02	word	hit area y size (height)
wwl_xorg	$04	word	hit area x origin
wwl_yorg	$06	word	hit area y origin
wwl_xjst	$08	byte	object x justification rule
wwl_yjst	$09	byte	object y justification rule
wwl_type	$0a	byte	object type (0=text, 2=sprite, 4=blob, 6=pattern)
wwl_skey	$0b	byte	selection keystroke (upper case)
wwl_pobj	$0c	long	pointer to object
wwl_item	$10	word	item number
wwl_pact	$12	long	pointer to action routine
wwl.elen	$16		loose menu item list entry length

The selection keystroke should be the 'upper case' value for letters and the event code (not the event number) for the event keystrokes. The event code is the event number less 14.

Information Sub-Window

An information sub-window is set up when the menu is set up, but has no further significance. The definition of information sub-windows is in the form of a list terminated by a word containing -1. Apart from the use of long word pointers, the information sub-window list is the same as in the window definition.

wwi_xsiz	$00	word	sub-window x size (width) in pixels
wwi_ysiz	$02	word	sub-window y size (height) in pixels
wwi_xorg	$04	word	sub-window x origin
wwi_yorg	$06	word	sub-window y origin
wwi_watt	$08		sub-window attributes
wwi_pobl	$10	long	pointer to information object list
wwi.elen	$14		information list entry length

The information sub-window origin is the pixel position of the top left hand corner of the inside of the sub-window with respect to the top left hand corner of the window.

Information Object List

Each object in an information object list has only a limited set of attributes, which may be different for each object. The list for each information sub-window is terminated by a word containing -1.

	wwo_xsiz	$00	word	object x size (width) in pixels
	wwo_ysiz	$02	word	object y size (height) in pixels
	wwo_xorg	$04	word	object x origin
	wwo_yorg	$06	word	object y origin
	wwo_type	$08	byte	object type (0=text, 2=sprite, 4=blob, 6=pattern)
	wwo_spar	$09	byte	spare
(	wwo_ink	$0a	word	text ink colour type=0
(	wwo_csiz	$0c	word	text character size (two bytes)
or				
(	wwo_comb	$0a	long	pattern or blob to combine type=4 or 6
	wwo_pobj	$0e	long	pointer to object
	wwo.elen	$12		information object list entry length

Application Sub-window List

Because the size of an application sub-window definition is dependent on the usage of the definition, the application sub-window list is just a list of long word pointers to individual application sub-window definitions. The list is terminated with a zero long word.

Application sub-window definition

wwa_xsiz	$00	word	sub-window x size (width) in pixels
wwa_ysiz	$02	word	sub-window y size (height) in pixels
wwa_xorg	$04	word	sub-window x origin
wwa_yorg	$06	word	sub-window y origin
wwa_watt	$08		sub-window attributes
wwa_pspr	$10	long	pointer to pointer sprite for this sub window
wwa_draw	$14	long	ptr to application sub-window draw routine
wwa_hit	$18	long	pointer to application sub-window hit routine
wwa_ctrl	$1c	long	pointer to sub-window control routine (or 0)
wwa_nxsc	$20	word	maximum number of x control sections
wwa_nysc	$22	word	maximum number of y control sections
wwa_skey	$24	byte	application sub-window selection keystroke
wwa.blen	$28		application sub-window basic definition length

Two control definitions, of the following structure, will always be present. The first will only be set up (non-zero) for pannable sub-windows, the second only for scrollable sub-windows.

wwa_part	$28	long	ptr to the part window control block (or 0) for pan, scroll and split definitions
wwa_insz	$2c	word	index hit size + scaling
wwa_insp	$2e	word	index spacing left or above sub-window+scaling
wwa_icur	$30	long	index current item attr. (border width, colour)
wwa_iiat	$34		index item attribute record
wwa_psac	$40	word	pan or scroll arrow colour
wwa_psbc	$42	word	pan or scroll bar colour
wwa_pssc	$44	word	pan or scroll bar section colour
wwa.clen	$1e		applic. sub-window control definition length

menu sub-windows only

wwa_mstt	$64	long	pointer to the menu status block
wwa_iatt	$68		item attributes
wwa_ncol	$90	word	number of actual columns
wwa_nrow	$92	word	number of actual rows
wwa_xoff	$94	word	x offset to start of menu (section)
wwa_yoff	$96	word	y offset to start of menu (section)
wwa_xspc	$98	long	pointer to x (column) spacing list
wwa_yspc	$9c	long	pointer to y (row) spacing list
wwa_xind	$a0	long	pointer to x (column) index list
wwa_yind	$a4	long	pointer to y (row) index list
wwa_rowl	$a8	long	pointer to menu row list
wwa.mlen	$48		length of menu working definition

The application sub-window origin is the pixel position of the top left hand corner of the inside of the sub-window with respect to the top left hand corner of the window.

If you supply a negative spacing value instead of a pointer to the spacing list, then all rows or columns are treaded as being of the same size.

The two control definitions must be present for all application sub-windows, but need only be set up if the sub-window is pannable (wwa_nxsc<>0) or scrollable (wwa_nysc<>0).

Menu Object Lists

It is assumed, by the menu interface, that the objects are arranged in a rectangular grid. Each column of the grid has a fixed width, each row a fixed height. The interface also allows for an index to the columns and an index to the rows to be placed above and to the left of the grid.

There are two dimensions, the first is the actual number of columns, the second is the number of rows. All of the lists have either one dimension or the other.

Each of the object spacing lists consists of pairs of numbers. The first is the hit area width or height, the second is the distance from the start of this hit area to the start of the next. Both spacings are in pixels. There must be sufficient gap between the objects to allow the current item border to be drawn.

Each of the object index lists has the same form as the object item list described below. The item numbers within these lists should be negative, and the action routine pointers zero.

The object item lists consist of a set of list entries, one for each column in a row. Each object list entry contains the item number for the object, the object type (test, sprite etc.), the justification (left, right or centre, top, bottom or centre), a pointer to the actual object and a pointer to an action routine to be called when the object is hit. Note that it is possible to have just one large object list, which is 'cut up' into rows by making each row list start pointer equal to the previous row list end pointer.

The justification rule bytes are zero for a centered object, positive for left or top justified and negative for right or bottom justified. The value indicates the distance of the object, in pixels, from the edge of the hit area.

The row list consists of pairs of pointers to the start and end of each object list.

menu object spacing list

wwm_size	$00	word	object hit size
wwm_spce	$02	word	object spacing
wwm.slen	$04		object spacing list element length

menu row list

wwm_rows	$00	long	pointer to object row list start
wwm_rowe	$04	long	pointer to object row list end
wwm.rlen	$08		menu row list element length

menu object / index list entry

wwm_xjst	$00	byte	object x justification rule
wwm_yjst	$01	byte	object y justification rule
wwm_type	$02	byte	object type (0=text, 2=sprite, 4=blob, 6=pattern)
wwm_skey	$03	byte	selection keystroke (upper case)
wwm_pobj	$04	long	pointer to object
wwm_item	$08	word	item number (-ve for index)
wwm_pact	$0a	long	pointer to action routine (zero for index)
wwm.olen	$0e		menu object / index list entry length

Working Definition Organisation

As the working definition is held together with pointers, it is not necessary for the data to be contiguous, or even in related parts of the memory. The window management setup routine, however, does transfer the data from the window definition to the working definition in an orderly manner.

	header
ww_lists (116)	
	information window list
wwi.elen (20) x ww_ninfo + 2	
	information object lists
wwo.elen (18) x ww_ninob + 2 x ww_ninfo	
	loose menu item list
wwl.elen (22) x ww_nlitm + 2	
	application window list
4 x ww_nappl + 4	
	application window definitions

The application sub-window definition set up by the window management routine WM.SETUP is $64 bytes long. This definition may be extended by either an application setup routine or the menu management setup routine.

An application sub-window definition set up by the menu management setup routine WM.SMENU has the following structure:

	application window definition
wwa.blen + 2 x wwa.clen + wwa.mlen (172)	
	column spacing list
wwm.slen (4) x wwa_ncol	
	row spacing list
wwm.slen (4) x wwa_nrow	
	column index index (optional)
(wwm.olen (14) x wwa_ncol)	
	row index list (optional)
(wwm.olen (14) x wwa_nrow)	
	menu row list
wwm.rlen (8) x wwa_nrow	
	menu object lists
wwm.olen x nr of objects	

Window Status Area

The window status area is used for communication between the application and the window and menu management routines. The window status area contains the pointer record, the tables giving the current sub-window and menu item, the control blocks for the pan, scroll and split status of a sub-window and the tables giving the status of all menu items.

The file `WMAN_WSTATUS` contains definitions of the symbols used in this section: it may be INCLUDEd in any assembler files that manipulate window status areas.

There is a fixed size base area which is pointed to from the window working definition header:

window linkage area

ws_work	$00	long	pointer to window working definition
ws_wdef	$04	long	pointer to window definition

window working area

ws_point	$08		pointer record (24 bytes)
wsp_chid	$08	long	channel ID of window enclosing the pointer
wsp_swnr	$0c	word	sub-window number enclosing pointer (or -1)
wsp_xpos	$0e	word	pointer x pixel position (sub-window)
wsp_ypos	$10	word	pointer y pixel position (sub-window)
wsp_kstr	$12	byte	key stroke (or 0)
wsp_kprs	$13	byte	key press (or 0)
wsp_evnt	$14	long	event vector
wsp_weve	$15	byte	window byte of event vector
wsp_seve	$16	byte	sub-window byte of event vector
wsp_peve	$17	byte	pointer byte of event vector
ws_subdf	$18		sub-window area definition (4 words)
wsp_xsiz	$18	word	sub-window x size (width)
wsp_ysiz	$1a	word	sub-window y size (height)
wsp_xorg	$1c	word	sub-window x origin
wsp_yorg	$1e	word	sub-window y origin
ws_ptpos	$20	long	pointer position (absolute)
ws_wmode	$24	word	display mode for this window
ws_ciact	$2c	long	pointer to current item action routine
ws_citem	$30	word	current item in sub-window
ws_cibrw	$32	word	current item border width
ws_cipap	$34	word	paper colour behind current item
ws_cispr	$36	word	spare
ws_cihit	$38		current item hit area (absolute coordinates)
ws_cihxs	$38	word	hit area x size
ws_cihys	$3a	word	hit area y size
ws_cihxo	$3c	word	hit area x origin in sub-window
ws_cihyo	$3e	word	hit area y origin in sub-window

The current item action routine is called whenever the the pointer is moved, or may be moved, while the current item is zero or positive. If this pointer is zero the internal current item routines are called: these require all the rest of the current item status area to be correctly set. If an action routine is supplied, then the $10 bytes after the action routine may be redefined as required.

The fixed size area is followed immediately by the loose menu item status block which gives the status of all the loose menu items. The block is indexed by the loose menu item number. The status block should be preset by the application: thereafter it is maintained by the window management routines.

loose menu item status block

ws_litem	$40	one byte per loose item ($10 is unavailable, $00 is available, $80 is selected)

The rest of the status area is in a free format. It may contain status blocks for the application sub-window menus (if any) and pan and scroll control blocks. Since there is a pointer from the window working definition to each of these blocks, they need not be contiguous and may be in completely unrelated parts of the memory.

For each standard format sub-window, there is a status block giving the status of each item in the sub-window menu.

sub-window menu item status block

wss_item	$00	one byte per menu item ($10 is unavailable, $00 is available, $80 is selected)

The status bytes in the item status blocks are used for communication between the application and the menu handling routines. Initially, the status of each item is set by the application. The window and menu drawing routines will draw each item using the appropriate colours patterns and blobs. The byte is divided into two nibbles: the upper nibble contains the required (or actual status), the lower nibble is zero except when an action routine requires an item to be redrawn.

If an item is "hit", or selected by keystroke, then, if the item is available, the status is changed. If an item is hit by a "do" then, if the item is available, the status is set to selected. If an action routine requires the status of any items to be redrawn, then the new status should be set in the upper nibble, and the least significant bit set.

Status	Normal	Redraw
unavailable	$10	$11
available	$00	$01
selected	$80	$81

For each sub-window, there may be an optional pan or scroll and split control block for horizontal and vertical control of a window.

This block starts with the number of pannable or scrollable sections, followed by a list of the start and end row or column number of each section. As usual, the start row or column is included in the section, the end row or column is excluded.

sub-window section control block header

wss_nsec	$00	word	number of sections

sub-window section control block record

wss_spos	$00	word	section start pixel position
wss_sstt	$02	word	section start column or row
wss_ssiz	$04	word	section size (number of columns or rows)
wss.ssln	$06		section status list entry length

If there is not a minimum size of ww.scarr for scroll arrows or ww.pnarr for pan arrows, they are not drawn at all.

Pointer Environment Changes

You are supplied with two versions of both the Pointer Interface and the Window Manager, of different vintages. Those loaded by the BOOT file are the more recent versions, and have more features than the old versions. The older versions are as shipped with QRAM v1.07, and are thus typical ofthe versions used by the majority of owners of QRAM. Should you wish to software for sale, you can either write for these older versions, accepting their restrictions, or for the newer versions, in which case some existing users of the Pointer Environment will be unable to use your software. A third option is to enter into a licensing agreement with QJUMP which would allow you to include the upgraded version of the Pointer Environment with your software in return for a suitable fee: as we intend the Pointer Environment to set a new standard for QL software, such a fee is unlikely to be excessive (end of sordid commercial!).

The following lists summarise the changes in the Pointer Toolkit, the Pointer Interface and the Window Manager.

Pointer Toolkit Changes

v0.01	Original released version.
v0.02	RD_PTR of window with no loose or menu items in allowed. MK_LIL with exactly one sprite/blob/pattern type item now works.
v0.03	CH_WIN now returns size change correctly. RD_PTR of window with more than one menu sub-window works (used to return as if an error had occurred, with D0=0).
v0.04	MS_SPD doesn't smash memory.
v0.05	Timeout set in MS_SPD, MS_HOT. CH_ITEM works for menu sub-windows.
v0.06	SWDEF doesn't reference address -4. WREST added.
v0.07	Correct number of procs.

Pointer Interface Changes

v1.06	Key debounce improved.
v1.07	First internal mouse version. Closing last window in particular mode now restores all windows in other mode.
v1.08	Avoids problems with closing unused consoles (It used to be able to lose the keyboard queue.) Improvements to screen restoration on window close.
v1.09	Prevents channel 0 from being closed. Mouse movement stuffs cursor keystrokes into keyboard queue. SD.WDEF (WINDOW from SuperBASIC) now resets cursor position. Multicolour patterns for blobs made usable.
v1.10	"Top" secondary is now the most recent one, not the first one. New TRAP IOP.FLIM, D0=$6C to find permissible limits for window. New TRAPs IOP.SVPW/RSPW D0=$6D/6E to save/restore part windows. IOP.RPXL now implemented: new spec. includes scanning. FWIND now only detects managed secondaries of managed primaries. IOP.OUTL can now move a secondary. IOP.OUTL now deals with secondaries that fall outside a re-defined primary (now set to primary's hit area). Odd shadow widths evened up. IOP.SPTR now only sets new position, so it works properly. Unmanaged secondaries now limited to managed primary outline, not whole screen. IOP.PICK ignores lock. IOP.PICK allows keyboard queue to be grabbed, so cursor appears OK. Hitting DO mouse button in keyboard window stuffs an ENTER. Both buttons on mouse stuffs one or two character string. Dropping blobs under sprite in MODE 8 fixed. Dynamic sprites implemented. Pattern outside sprite mask is now XORed into screen, not ORed. Extending an unmanaged locked primary's outline by opening a larger secondary now works.
v1.12	First PTR_GEN.
v1.13	Move window on odd pixel boundary/odd width now permitted (MODE 4).
v1.14	ESC while doing special RPTR now gets through (got lost in vv1.xx-1.13).
v1.15	FWIND gets X size of sub-window correct, it was one too big
v1.16	RPTR signals SCHED to make pointer visible.
v1.17	First Atari ST Pointer Interface.
v1.18	Patched to enable dropping of sprites and blobs which are larger than the pointer sprite. Save areas now owned by the same job as the channel, with null driver. Dummy CON is ROM CON, not current CON.
v1.19	CTRL F5 during MODE now works (!) RPIXL can now scan left/right for a given colour correctly. Mode change between window open and use is now OK. Other dummy channels diverted via our linkage block, so MODE doesn't spot them. Cursor status cleared before MODE window redraws. RPTR does not signal SCHED so much - see V1.16.
v1.20	All PICKs now move pointer to primary centre, not just CTRL C.
v1.21	Window and border changes clear scheduler flag.
v1.22	Pick windowless JOB now OK on ATARI.
v1.23	CKEYON and CKEYOFF added to control action of cursor keys. WWA.KFLG added to window attributes for the same reason. Type ahead enabled within a window. Third attempt at Thor 1 version. Keypress suppressed on window change.
v1.24	Thor 1 version allows Thor patch to be used before loading Lightning.
v1.25	Thor 1 version supports all three buttons on the Thor mouse. Failure of DEL_DEFB introduced in Thor mods to v1.23 fixed.
v1.26 (internal)	Wake event generated when 'picking' with DO button or if required in IOP.PICK trap (D2=K.WAKE).
v1.27 (internal)	Wake events improved. Keyboard queue of locked, busy or no window stripped.
v1.28	Escape from window identify restored (problem in 1.27 only)
v1.29	CTRL C spurious wake removed. Problem with rapid "CTRL C"s removed (introduced in version 1.23). HIT while moving restored (missing since V1.23)

v1.30	PICK to center of top secondary. Pointer movement slowed while disk etc busy.
v1.31	Bad driver for save area corrected. No wake-up on cursor key strokes.
v1.32	Allocates enough room for a 64x48 pointer sprite. Improvements to out of window keystrokes.
V1.33	Improved dragging. Pointer movement restored from v1.30. Checks for cursor overlap on RHS.
V1.34	Pointer movement slowed again.
V1.35	Cursor suppression algorithm improved.
V1.36	Corrected a fault in the V1.35 cursor suppression algorithm. Pointer limiting introduced for dragging.
V1.37	Option to Freeze jobs on locking window.
V1.38	Close removes Fill buffer. Both ENTER keys on ST cause DO.
V1.39	IOP.RPXL removes pointer sprite.
V1.40	Higher RES mode supported
V1.41	Higher RES corrections
V1.42	Sprite / Blob dropping problems introduced in V1.40 fixed.
V1.43	Window area for non-well behaved windows can exceed 512x256.
V1.44	Some changes to sprite suppression / appearance
V1.45	More changes to sprite suppression / appearance
V1.46	IOW.SSIZ accepts -1,-1 for no change in size (size enquiry)
V1.47	Window Move $84 has invisible sprite
V1.48	Partial save / restore corrected for non-QL screen sizes. Dragging restored (V1.45) even when pointer is being reset
V1.49	Sprite remove checks updated for wider screens.
V1.50	Partial save/restore updated for monochrome mode.
V1.51	Sprite suppression / appearance restored to old style.
V1.52	Open CON (copyc) "out of memory" error recovery fixed.
V1.53	Initialisation works even if no RTC.
V1.54	Modification of Atari polling routine.
V1.55	IOP.RPXL corrected for non QL screens.
V1.56	IOP.SVPW memory allocation modified - should have no effect.
V1.57	Corrections to V1.56 for QDOS. MODE improvements.
V1.58	Corrections to V1.55.

Window Manager Changes

V1.04	WM.DRBDR added.
V1.05	Zero text pointers allowed, information blobs corrected.
V1.06	CHWIN returns size change. Initial pointer set rel hit area. Fixed window sizes accepted by SETUP. BREAK detected. Pending newline problems in information windows removed. Menu sub-window paper set before scrolling.
V1.07	Lots of new routines. SSCLR and ARROW made regular fmt.
V1.08	CHWIN fixed for secondaries/cursor keys.
V1.09	Non-cleared info windows allowed. Vectors $48 to $74 added.
V1.10	Setup correct number of columns > 3. Set pointer position correctly in odd position application sub-window.
V1.11	Returns user defined message if no PTR.
V1.12	Fractional scaling bug fixed.
V1.13	WM.MSECT extended to accept cursor keys, SPACE and ENTER in arrow rows. DO/ENTER in arrow row of single item menu section made equivalent to HIT/SPACE. Cosmetic improvements to menu and current item handling.
V1.14	WM.DROBJ updated to draw sprites 1 to 7 in the right place.
V1.15	Sleep and wake event keystrokes added to WM.RPTR. Characters in the range $09 to $1f recognised in WM.RPTR.
V1.16	Improved wake. Control codes less than 9 accepted. WM.RNAME, WM.ENAME return terminator in D1 as it should.
V1.17 (int)	Pan/scroll bars at last.
V1.18 (int)	Pan/scroll arrows made optional, bars tidied up.
V1.19 (int)	WM.SWAPP corrected for application windows >0. Improvements to out of window keystrokes. WM.CHWIN now allows cursor keys for pull down window moves - regardless of circumstances.
V1.20 (int)	Out of window wake accepted again (went in 1.19).
V1.21 (int)	DO anywhere in window accepted.
V1.22 (int)	Constant Spacing in menus.
V1.23 (int)	Repeated selection key handled. Dragging on pan/scroll bars implemented.
V1.24	Improvements to FSIZE for windows variable in two dims.
V1.25	Further improvements to pan/scroll bars.
V1.26	WM.STLOB status set OK. WM.UPBAR added.
V1.27	WM.SWLIT now sets cursor position using justification. WM.RNAME WN.ENAME start from cursor position.
V1.28	Pan/scroll bars with no sections cleared (V1.26, V1.27)
V1.29	Sub-window select keystroke (-1 in D2) re-introduced.
V1.30	Sub-window control routine called only on move or hit. Window origin scaleable. DRBAR can draw full length bar (V1.26-V1.29). Event with no loose item accepted anywhere in window.

V1.31	Underline nth character of text type -n. WM.MHIT returns D4=0 if action or control routine called.
V1.32	DO item action routine called on DO in window
V1.33	Text position set before character size set (prevents spurious scroll.
V1.34	Split cannot generate empty sections.
V1.35	Character size only set if non-standard. Requires 1.46 Pointer.
V1.36	WDRAW corrected so as not to smash d5/d6 (error in 1.35).
V1.37	Scaling of menu spacing. Fixed menu spacing (first spacing negative) allowed in definition.
V1.38	Minimum limit for window rounded up to 4 pixel boundary.
V1.39	CHWIN does not smash D4 and D7 on move.
V1.40	Underline permitted for text starting with spaces.
V1.41	WM.RNAME WM.ENAME does not edit text longer than window.
V1.42	Extended WM.RNAME WM.ENAME.
V1.43	Set window resets character size to 0,0.
V1.44	Pan and scroll bars corrected for border >1.
V1.45	V1.44 corrected.
V1.46	CSIZE reset when no info text item.
V1.47	Corrects V1.46.

Utilities

Two utility programs are provided: they are ordinary EXECutable programs which may be started from SuperBASIC or Qram's FILES menu.

CVSCR

This utility converts a screen image file into a format suitable for loading into the PAINT demonstration program. It requests an input filename, and checks that it is exactly 32k long, and of an appropriate type (not executable). If the input file passes these tests, an output filename is requested, into which the processed file will be written: if this already exists then you are asked whether it is OK to overwrite it. Finally the program asks which screen mode the screen image was in, there being no way to determine this from the file, and writes out the converted file.

The conversion process adds a 10-byte header onto the start of the screen image data, consisting of a flag, X and Y sizes (in pixels), line length in bytes, and the mode flag.

STKINC

This utility is used to process SuperBASIC programs which use the Window Manager facilities of the Pointer Toolkit, and have been compiled using v3.12 or earlier of the Q_Liberator compiler. It is not required with v3.21 onwards - if you have this or a later version then you can compile and run a program using the Window Manager in exactly the same way as any other. STKINC fixes the problem caused by the Window Manager using more stack than Q_Liberator provides, by increasing the provision. This modification needs to be done in the file header, the compiled code and the run-time system, so the run-time system must have been included in the object file. One filename is requested, and the file is converted in place as no size change is involved. The program will usually notice if the file is not a Q_Liberated object file including the run-time system, and complain.

FIXPF

This utility takes the form of a SuperBASIC procedure, and may be used to restore the ROM version of any built-in procedure or function. If required, it should be loaded into the resident procedure area by your BOOT file, as described on page 5.

FIXPF should never be needed! Unfortunately some packages "fix bugs in" or "improve on" the way SuperBASIC works by re-defining existing ROM routines, and in the process cause more problems than they cure. An example is the way the RESPR function can be re-defined to allocate space in the common heap, which "avoids the problem" of not being allowed to reserve more space in the resident procedure area once jobs are running. It is also very dangerous, as the heap space could be returned and re-used, resulting in a crash when procedures which were in that space are called. We have also seen examples of RESPR being re-defined within a program: when that program goes away, taking the new RESPR with it, you get problems.

You can even use FIXPF on SuperToolkit commands if you like! If you find that the "improved" versions of SAVE and LOAD keep using the defaults to save or load from the wrong device, you could FIXPF them so they need an exact filename, as before. This would also get rid of the "File already exists - OK to overwrite?" message.
The syntax of the procedure is:

```
FIXPF 'name'
```

The quotation marks are required, as you can't use procedures as parameters. The procedure or function `name` should be an original QL ROM routine. You can FIXPF a routine as often as you like without causing problems.

Known candidates for being FIXPFed are any re-defined versions of RESPR, and the SPEEDSCREEN version of MODE when the Pointer Interface is installed. The Pointer Interface takes care of all MODE calls, not just SuperBASIC ones as SPEEDSCREEN does, so the new version of MODE is unnecessary: in fact it can be dangerous - we have seen "total lockups" resulting from trying to pop up QRAM after the SPEEDSCREEN MODE has been used. This problem may be cured in future versions.

Troubleshooting

You may encounter problems with the Pointer Toolkit: the following list is by no means exhaustive, but covers some of the most likely causes of error.

My program (or one of the demos) worked OK yesterday, but it doesn't work today. This is usually caused by changing your BOOT file, or some other aspect of your system not directly connected with the program itself. In particular, you must set SuperBASIC's outline with an OUTLN #0... call to use all but the simplest parts of the Toolkit: if you don't, then the Pointer Interface will assume that SuperBASIC is "unmanaged", and not bother to check for sub-windows, user-defined pointers and so on.

My program never returns from a "read pointer" call. You can only use a "managed" window for pointer input: if you use an unmanaged window then the pointer always seems to be outside it. A window can be made managed by a call to OUTLN or DR_PPOS from SuperBASIC, or to the IOP.OUTL TRAP or WM.PRPOS vector in machine code.

I don't get my special sprite, just the arrow. User-defined sprites appear in sub-windows as a result of a call to SWDEF or IOP.SWDF to set up the appropriate data structure. Sub-windows will be ignored if their "parent" window or its primary (or both) are "unmanaged". They will also be ignored if there is a gap in the sub-window list, as the list is terminated by a zero pointer so a zero in the middle of the list is interpreted as an end of list marker.

My program works when interpreted, but not when it's compiled. SuperBASIC programs using the Pointer Toolkit can't be compiled with the Super/Turbocharge compilers, as they can't cope with array parameters or results returned in the parameter list. If compiled with Q_Liberator then you will have problems if you have used Window Manager routines but have not used the STKINC utility on the resulting program. The program will not work if its outline has not been set: see above.

My compiled program starts off OK but then it crashes. This is usually caused by not using the STKINC utility where appropriate: it can also happen if you haven't specified enough heap, stack or buffer space for the program.

My machine code program crashes in the Window Manager. This is very often caused by an incorrect window definition, which causes the setup routine WM.SETUP to use more space, when creating the working definition, than was anticipated. If this space is in the common heap then the following heap header will be corrupted, resulting in a system crash instantly or half an hour later, depending.

One or more of the items doesn't get selected on its keystroke. When specifying a keystroke to select a menu item, remember that the character must be **specified** in upper case, although it doesn't matter if the key **pressed** is upper or lower case. Remember also that event keys such as HELP, CANCEL and so on are translated to have very low key values such as 4, 3 and so on.

I get an "out of range" error on a WINDOW command that worked before. Managed secondary windows, which are needed for most of the examples, may not be positioned, by a call to either OUTLN or WINDOW, outside the outline of their primary window. The examples provided in QPTR assume the use of the BOOT file provided, which sets the SuperBASIC's outline to the whole screen - if you use a different BOOT file setting another outline then they may stop working.

CONFIG

Configuration Information Specification

Many programs have the facility to configure themselves to set default working parameters. More usually the configuration is done by a separate program which modifies the working program file. Each program will have a different configuration program, and often different versions of the same program will have different configuration programs too. All this makes things very difficult for users.

It is proposed that a standard configuration system is used on all new programs and all new releases of existing programs. If this is done, a single configuration program can be used on any application software file even even when several application files are concatenated.

The advantages of this approach are obvious. There are two disadvantages. The first is that each program has to carry with it all the configuration information: this will make it larger. The second is that there is no simple means for doing this with compiled BASIC programs. The first will not usually be a problem as it seems unlikely that a 32k program would have more than about 20 configurable items and their associated descriptions, this would add at most 3% to the program size. The second can be overcome with a little will.

There are two parts to this system: the first is a standard for the format of a configurable file, the second is a program to process files. There can be any number of programs to process files, from any number of suppliers. If the standards for the configurable file are adhered to, then any supplier's configuration program can be used on any (other) supplier's software.

The configuration consists of the following information:

- Configuration ID
- Configuration level
- Software name
- Software version
- List of
    - Type of item (string, integer etc.) (byte)
    - Item Selection keystroke (byte)
    - Pointer to item
    - Pointer to item pre-processing routine
    - Pointer to item post-processing routine
    - Pointer to description of item
    - Pointer to attributes of item (item type dependent)
- End word (value -1)

As time goes by, additional types of item are likely to be added. This will mean that new versions of the configuration program will be required. These new versions will, of course, be able to configure all lower level configurable files. But, if a old configuration program is used, and the level specified in the configuration block is greater than the level supported by the configuration program, it will have to give up gracefully.

The configuration ID is word aligned and is the eight characters "<<QCFX>>", this is followed by two ASCII characters giving the configuration level (minimum "01"). The software name is a standard string and is followed by a word aligned version identification in a standard string (e.g. "1.13a"). The word aligned list of items follows.

`Types of item`

Level 01 supports 7 types of item. These are: string, character, code selection, code, byte, word and long word. Application specific types of item can be processed by treating them as strings which are handled entirely by an application supplied routine.

`String (type=0)`

The form of a configurable string is a word giving the maximum string length, followed by a standard string. There should be enough room within the application program for the maximum length string plus one character for a terminator. There is a single word of attributes with bits set to determine special characteristics.

bit 0	do not strip spaces

`Character (type=2)`

A character is a single byte, if it is a control character, it will be written out as a two character string (e.g. ^A = $01). There is a single word of attributes with bits set to determine the possible characters allowed.

bit 0	non printable characters
bit 1	digits
bit 2	lower case letters
bit 3	upper case letters
bit 4	other printable characters
bit 6	cursor characters
bit 8	control chars + $40, translated to control chars

Bit 8 is, of course, mutually exclusive with bits 0 to 7, although this is not checked. The configuration block in an application program must be correct.

`Code (type=4)`

A code is a single byte which may take a small number of values. The attributes is a list of codes giving a byte with the value, a byte with the selection keystroke and a standard string. The list is terminated with an end word (value -1). There are two forms. In the first, the selection keystrokes are set to zero. In this case, when a code is selected, the value will step through all possible values. This is best suited to items which can only have two or three possible codes. Otherwise the user may select any one of the possible codes, either from a list (interactive configuration programs) or from a pull down menu (menu driven configuration programs).

`Selection (type=6)`

A selection is in the same form as a code, but instead of a byte being set to the selected value, the value is treated as an index to a list of status bytes. When one is selected, it is set to wsi.slct ($80), the previous selection (if different) is set to wsi..avbl (zero). If any status bytes are unavailable (set to wsi.unav=$10), then they will be ignored. The first status byte in the list must not be unavailable.

`Values (types 8,10,12)`

Largely self explanatory. The attributes are the minimum and maximum values. All values are treated as unsigned.

Item Selection Keystroke

The item selection keystroke is an uppercased keystroke which will select the item in the main menu. The action of selecting the item will depend on the item type. For a code or select item a pull-down window may be opened to enable the user to select the appropriate code. For character item, a single keystroke will be expected. for all other types of item, the item will be made available for editing. For interactive configuration programs, the selection keystroke has no meaning.

Pointer to Item

The pointer to item, and all the other pointers in the definition, are relative addresses stored in a word (e.g. dc.w item-*).

Pointer to Item Pre-Processing Routine

It is possible to provide a pre-processing routine within the main program which will be called before an item is presented for changing. This will be when the item is selected in a menu configuration program, or before the prompt is written in an interactive configuration program. If there is no pre-processing routine, the pointer should be zero. The amount of pre-processing that application program can do is not limited. It could just set ranges, or it could do the complete configuration operation itself, including pulling down windows.

```
|                                                                       |
|   Pre-processing Routine                                              |
|                                                                       |
|   Call parameters                     Return parameters               |
|                                                                       |
|                                       D0   item set / error           |
|                                       D1+  scratch                    |
|   D7   0 / Window Manager vector      D7   scratch                    |
|                                                                       |
|   A0   pointer to item                A0   scratch                    |
|   A1   pointer to description         A1   (new) ptr to description   |
|   A2   pointer to attributes          A2   (new) ptr to attributes    |
|   A3   pointer to 4 kbyte space       A3   scratch                    |
|                                       A4+  scratch                    |
|                                                                       |
|   Error returns: set as D0                                            |
|        >0  item set, do not prompt or change                          |
|        =0  ok                                                         |
|        <0  error                                                      |
|                                                                       |
```

The space pointed to by A3 is not used by the configuration program and can be used by the application code. Initially it is clear. The application code may use up to 512 bytes of stack.

If D0 (and the status) is returned <0, then the Configuration program will write out an error message and stop.

Pointer to Item Post-Processing Routine

It is possible to provide a post-processing routine within the main program which will be called for each item before configuration starts, and for each item after any item is changed. It can be used to set limits or other dependencies.

```
|                                                                   |
|   Post-processing Routine                                         |
|                                                                   |
|   Call parameters                  Return parameters              |
|                                                                   |
|                                    D0    item set / error         |
|   D1.b  set this item jsut changed D1.b  item status (avbl/unav)  |
|                                    D2+   scratch                  |
|   D7    0 / Window Manager vector  D7    scratch                  |
|                                                                   |
|   A0    pointer to item            A0    scratch                  |
|   A1    pointer to description     A1    (new) ptr to description |
|   A2    pointer to attributes      A2    (new) ptr to attributes  |
|   A3    pointer to 4 kbyte space   A3    scratch                  |
|                                    A4+   scratch                  |
|                                                                   |
|   Error returns: set as D0                                        |
|         >0    bit 0 item reset                                    |
|               bit 1 description reset                             |
|               bit 2 attributes reset                              |
|         =0  ok                                                    |
|         <0  error                                                 |
|                                                                   |
```

The space pointed to by A3 is not used by the configuration program and can be used by the application code. Initially it is clear. The application code may use up to 512 bytes of stack. If an item description is changed, it should occupy the same number of lines as the original.

The returned values for D1 are WSI.AVBL ($00) if the item can be changed or WSI.UNAV ($10) if the item is not available for changing.

If D0 and the status are <0, A1 and A2 and the item status will not be updated, the error messsage will be written out, no further postprocessing routines will be called, and (for an interactive Configuration program) the item just set will be re-presented.

A post-processing routine can also be used to set up initial descriptions and attributes.

Description of Item

The description of an item is in the form of a string. Each description can have several lines, separated by newline characters. Each line should be no longer than 64 characters, except the last line must allow space for the longest item. Interactive programs may append a list of states or selections to the description.

Pointer to attributes

The attributes are item dependent. See item types for descriptions.

Latest improvements

Additional information on WM.ERSTR

This manual did not mention that there is a limit on the length of own error messages. An own error messages is easy to create:

```
LEA    own_msg,A0    ; get address
MOVE.LA0,D0          ; into our "error" register
BSET   #31,D0        ; an error is negative
```

Now the limit: the length of the string is limited to 40 ($28) characters. If it is longer, "unknown error" is returned instead!

Additional information on WM.LDRAW

WM.LDRAW clears the change bit in the status are of every item which is selectively redrawn.

Additions to the CONFIG standard

The attributes for strings have been extended, to allow menu-driven CONFIG programs better options for a selection, depending on the type. There are two additional bits used in the string attributes: 8 and 9. These define the type of string, so that the CONFIG program can treat these strings in a special way. The possible combinations are:

cfs.sspc	equ	%0000000000000001	string strip spaces
cfs.file	equ	%0000000100000000	string is filename
cfs.dir	equ	%0000001000000000	string is directory
cfs.ext	equ	%0000001100000000	string is extension

At present, these features are supported by the new MenuConfig, and ignored by the standard config.

Undocumented SuperBASIC Procedure

SPTR has never been documented. Easy to guess, it does the same as IOP.SPTR, i.e. moves the pointer to a given position. The syntax is:

```
SPTR [#channel], xpos, ypos [,key]
```

Option	Default	Meaning
xpos, ypos	none	new pointer position
key	-1	origin key

The origin key should be zero if the pointer coordinates are absolute. A key of -1 will set the position relative to the current window definition. A key of 1 will set it relative to the hit area.

Undocumented selection keystroke for SuperBASIC

It is possible to put an underscore under a selection key for text loose menu items and text info items. To do this, specify the type to be text minus twice the underscore position. This means, to underscore the first character, give 0-2 (=-2), to underscore the fifth position give -10 etc.