

WINDOW SERVER REFERENCE

Version 2.30

(C) Copyright Psion PLC 1990-98

All rights reserved. This manual and the programs referred to herein are copyrighted works of Psion PLC, London, England. Reproduction in whole or in part, including utilization in machines capable of reproduction or retrieval, without express written permission of Psion PLC, is prohibited. Reverse engineering is also prohibited.

The information in this document is subject to change without notice.

Psion and the Psion logo are registered trademarks, and Psion, Psion MC, Psion HC, Psion Series 3, Psion Series 3a, Psion Series 3c, Psion Siena and Psion *Workabout* are trademarks of Psion PLC.

TopSpeed is a registered trademark of Clarion Software Corporation. IBM, IBM XT and IBM AT are registered trademarks of International Business Machines Corp. Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Apple and Macintosh are registered trademarks of Apple Computer Inc. VAX and VMS are registered trademarks of Digital Equipment Corporation. Brief is a registered trademark of Underware Inc. Psion PLC acknowledges that some other names referred to are registered trademarks.

Contents

1 Introduction	1-1
Versions of the window server.....	1-1
Version 4	1-2
Version 3.5:window server	1-5
Version 3:window server	1-7
Specific to the MC.....	1-7
WLIB:library using	1-8
The WLIB header.....	1-8
Connecting to the window server.....	1-8
Using the CLIB startup module on the HC, S3, S3a or <i>Workabout</i>	1-8
Using the PLIB startup module on the HC, S3, S3a or <i>Workabout</i>	1-9
Using the CLIB startup module on the MC.....	1-10
Using the PLIB startup module on the MC	1-10
Error handling	1-11
How errors are signalled.....	1-11
Errors in blind operations.....	1-11
Cleaning up after an error	1-11
Panic numbers.....	1-12
Series 3 compatibility mode.....	1-12
Clients and the window server.....	1-13
Client-side functions	1-13
Client-side buffer and flushing	1-13
Testing for a connection	1-13
Foreground and background clients.....	1-14
Events	1-14
Task switching;.....	1-14
Task switching on the HC	1-15
Task switching on the <i>Workabout</i>	1-15
Task switching on the S3 and the S3a	1-15
Task switching on the MC.....	1-16
Iconised clients.....	1-16
Client priorities	1-16
System-modal clients.....	1-17
Client management	1-17
Windows.....	1-18
Window trees	1-18
Ownership of windows	1-18
Background client drawing.....	1-18
Drawing region	1-19
Backed-up windows.....	1-19
No-redraw windows	1-20
Bitmap sequences.....	1-21
Using an attached bitmap to avoid redraws.....	1-22
Sprites.....	1-23
Redrawing.....	1-24
Redraw events.....	1-24
Update region.....	1-24
Invalidating rather than drawing	1-24
Validating before drawing or redrawing	1-25
Drawing.....	1-25
Redrawing.....	1-25
Going deaf	1-26
Redraw priority	1-27

More about windows	1-27
Creating and initialising a window system.....	1-27
Destroying a window system.....	1-27
Event sources other than the window server.....	1-28
Visibility of windows.....	1-28
Scrolling	1-28
Continuous scrolling	1-29
Text cursor	1-29
Bitmaps.....	1-30
Drawing to the screen from a bitmap.....	1-30
Creation and storage of bitmaps.....	1-30
Drawing to bitmaps	1-31
Bitmap files.....	1-31
Using wpcx.....	1-31
Capturing the screen to a bitmap file	1-32
Screen capture program for the HC.....	1-33
Capturing the screen directly to a PCX file.....	1-33
The physical structure of bitmap files and bitmaps.....	1-36
Embedded bitmap files	1-37
Text fonts.....	1-38
Fast fonts.....	1-38
ROM-based fonts.....	1-38
HC fonts.....	1-39
S3 fonts	1-40
MC fonts	1-41
S3a and Workabout fonts	1-41
Font files.....	1-42
Using wsfcomp.....	1-43
Font information	1-44
Text drawing.....	1-44
The physical structure of font files.....	1-46
Embedded font files.....	1-47
Graphics output.....	1-47
Pixel coordinates	1-47
Graphics structs.....	1-48
Graphics output.....	1-49
Graphics output other than to the current graphics context	1-49
Graphics contexts.....	1-50
Current graphics context	1-50
Permanent and temporary graphics contexts	1-50
Creating a permanent graphics context.....	1-50
Creating a temporary graphics context.....	1-51
Keyboard input.....	1-51
Key press events.....	1-51
Capturing keys	1-51
Pausing the client	1-52
Setting task keys.....	1-52
Mouse input	1-52
Mouse icons	1-52
Mouse events.....	1-52
Capturing the mouse	1-53
Grabbing the mouse.....	1-53
Input-only windows.....	1-53
Inactive windows.....	1-53
The rubber band	1-53
System start-up.....	1-54
Using the window server's notifier on the HC	1-54
The \$WS_FL environment variable on the HC	1-55
S3a.....	1-56
Replacing the shell on the HC	1-56
PC EPOC	1-59

2 General Window Server Functions.....	2-1
The connection to the window server.....	2-1
Connect and initialise a window (wStartup).....	2-1
Connect to the window server (wConnect).....	2-3
Screen dimensions.i.pixel:pitch dimensions.....	2-5
Disconnect from the window server (wDisconnect).....	2-5
Flush buffered commands (wFlush).....	2-5
Series 3 compatibility modes.....	2-6
Set or cancel compatibility mode (wCompatibilityMode)	2-6
Inquire state of compatibility flags (wInquireCompatibility)	2-7
Error handling	2-7
Return to defined state (wCleanup).....	2-7
Check for an error (wCheckPoint)	2-7
Disable/enable leaves (wDisableLeaves)	2-7
Priority changing	2-7
Set process priority handling on or off (wSetPriorityControl).....	2-8
Enter compute mode (wStartCompute)	2-8
Leave compute mode (wEndCompute).....	2-8
General client functions	2-8
Get information about a client (wClientInfo)	2-8
Position client in task order (wClientPosition)	2-8
Mark client as iconised (wClientIconised)	2-9
Make client system modal (wSystemModal)	2-9
Cancel system modal state (wCancelSystemModal)	2-9
Enable pause key (wEnablePauseKey)	2-9
Disable pause key (wDisablePauseKey)	2-10
Get client list (wGetProcessList).....	2-10
Screen-based output	2-10
Present an information message (wInfoMsgCorner).....	2-11
Present an information message (wInfoMsg)	2-11
Present a flashing busy message (wSetBusyMsg).....	2-11
Cancel a flashing busy message (wCancelBusyMsg).....	2-12
Alerts.....	2-12
Present and wait for an alert (wsAlertW).....	2-14
Present an alert (wsAlertA)	2-18
Update a pending alert (wsAlertUpdate).....	2-19
Status windows	2-19
Enable the permanent status window (wsEnable).....	2-21
Set the state of the status window (wsStatusWindow)	2-21
Get screen extent for tile with status window (wsScreenExt).....	2-21
Update the permanent status window (wsUpdate)	2-22
Disable the permanent status window (wsDisable).....	2-22
Enable temporary status windows (wsEnableTemp).....	2-22
Disable temporary status windows (wsDisableTemp).....	2-22
Set list of modes to display in status window (wsSetList)	2-23
Set select position in status window mode list (wsSelectList)	2-23
Inquire state and extent of status window (wInquireStatusWindow).....	2-23
Configuring the window server	2-24
Configure the window server (wSystem).....	2-24
Attached Clients.....	2-25
Attach to client (wAttachToClient).....	2-26
Attach to foreground client (wAttachToForegroundClient).....	2-26
Detach from client (wDetachClient)	2-26
Miscellaneous	2-26
Get information on supported features (wSupportInfo)	2-26
Set or cancel key click disable state (wDisableKeyClick)	2-26

3 Windows	3-1
Creating and initialising a window	3-1
Window attributes	3-1
Create a window (wCreateWindow).....	3-4
Set window attributes (wSetWindow)	3-5
Get window attributes (wInquireWindow)	3-5
Initialise window tree (wInitialiseWindowTree)	3-5
Destroy a window and its descendants (wCloseWindowTree)	3-6
Get window to window offset (wInquireWindowOffset).....	3-6
Reassign the root window (wReassignRootWindow).....	3-6
Visible and invisible windows	3-6
Make window invisible (wMakeInvisible).....	3-6
Make window visible (wMakeVisible)	3-7
Sibling positions.....	3-7
Change position in sibling list (wWindowPosition).....	3-7
Get position in sibling list (wGetWindowPosition)	3-7
Scrolling.....	3-7
Copy a rectangle (wScrollRect).....	3-7
Scroll a window (wScrollWin).....	3-8
Redrawing.....	3-8
Start a partial redraw (wBeginRedraw).....	3-9
Start a full redraw (wBeginRedrawWin).....	3-9
Start a partial redraw (GC) (wBeginRedrawGC).....	3-9
Start a partial redraw (GC0) (wBeginRedrawGC0).....	3-10
Start a full redraw (GC) (wBeginRedrawWinGC).....	3-10
Start a full redraw (GC0) (wBeginRedrawWinGC0).....	3-10
End a redraw (wEndRedraw).....	3-11
Validating.....	3-11
Validate a rectangle of a window (wValidateRect).....	3-11
Validate a whole window (wValidateWin).....	3-11
Invalidating.....	3-12
Invalidate a rectangle (wInvalidateRect).....	3-12
Invalidate a window (wInvalidateWin)	3-12
Text cursor	3-12
Draw a text cursor (wTextCursor)	3-12
Draw a text cursor (wDrawTextCursor)	3-13
Erase a text cursor (wEraseTextCursor).....	3-13
Bitmap sequences.....	3-14
Attach bitmap sequence to window (wSetWinBitmap).....	3-14
Change a bitmap (wChangeWinBitmap)	3-15
Free a bitmap sequence (wFree).....	3-15
Sprites.....	3-15
Create a sprite (wCreateSprite).....	3-15
Change a sprite's bitmaps and position (wSetSprite).....	3-16
Free a sprite (wFree).....	3-16
Clocks.....	3-16
Create a clock (wsCreateClock)	3-17
Create a clock - Enhanced version (wsCreateClock2)	3-19
Set the clock offset (wsSetClock)	3-22
Free a clock (wFree)	3-22
Mouse icons	3-22
Load a mouse icon (gOpenMouseIcon)	3-23
Free a mouse icon (wFree).....	3-23

4 Graphics Output.....	4-1
Graphics contexts.....	4-1
Creating a permanent graphics context.....	4-4
Create a permanent GC (gCreateGC)	4-4
Create a permanent GC with default values (gCreateGC0)	4-5
Free a permanent GC (wFree)	4-5
Creating a temporary graphics context	4-5
Create a temporary GC (gCreateTempGC)	4-5
Create a temporary GC with default values (gCreateTempGC0)	4-6
Free a temporary GC (gFreeTempGC).....	4-6
Setting a graphics context	4-6
Set a graphics context (gSetGC).....	4-6
Make a permanent GC current (gSetGC0).....	4-7
Line drawing.....	4-7
Draw a line (gDrawLine).....	4-7
Draw a sequence of lines (gDrawPolyLine).....	4-7
Draw a box (gDrawBox).....	4-8
Border a rectangle (gBorderRect)	4-8
Border a drawable (gBorder)	4-10
Draw a 'shadowed' border (gBorder2Rect)	4-10
Draw a 'shadowed' border (gBorder2).....	4-12
Draw a graphics object (gDrawObject)	4-12
Area filling	4-12
Change a rectangle (gClrRect).....	4-12
Invert an obloid (gInvObloid).....	4-13
Fill a rectangle with a bitmap (gFillPattern)	4-13
Text fonts.....	4-14
Open a font (gOpenFont).....	4-15
Set pos to open font/bitmap/mouse icon (gSetOpenAddress).....	4-15
Free a font (wFree).....	4-16
Get font information (gFontInfo).....	4-16
Get text width (gTextWidth)	4-16
Clip text to pixel width (gTextCount).....	4-17
Get a font width table (gGetWidthTable).....	4-17
Set the system and internal fonts (wSetSystemFont)	4-17
Open a font from a multiple font file (gOpenFontIndex).....	4-18
Configure a font group (gConfigureFonts).....	4-18
Read a font header from a file (gReadFontHeader)	4-19
Read a font group header from a file (gReadFontGroupHeader).....	4-19
Text output functions	4-19
Print text (gPrintText)	4-20
Print clipped text (gPrintClipText)	4-20
Print text in a box (gPrintBoxText)	4-20
Print text with embellishment (gXPrintText).....	4-21
Print shadowed text (gShadowText)	4-23
Draw a text button (wDrawButton).....	4-24
Draw a text button (wDrawButton2).....	4-25
Bitmaps.....	4-26
Create a bitmap (gCreateBit)	4-26
Load a bitmap (gOpenBit).....	4-27
Free a bitmap (wFree)	4-28
Save a bitmap (gSaveBit)	4-28
Save part of a bitmap (gSaveRect)	4-28
Copy a bitmap to a window (gCopyBit)	4-29
Copy a bitmap onto itself (gCopyRect).....	4-30
Read a bitmap (gPeekBit).....	4-30
Check if a bitmap is valid (gCheckBitmapID)	4-30
Multiple bitmaps	4-30
Open a bitmap file (gInitBit)	4-31
Load a bitmap from an open file (gGetBit)	4-31
Draw a bitmap from an open file (gDrawBit).....	4-31
Query the size of a bitmap (gQueryBit).....	4-31
Close an open bitmap (wFree)	4-32

Initialise a multiple bitmap file (gInitMultiSave)	4-32
Save a bitmap to a multi bitmap file (gSaveMultibit)	4-32
Save part of bitmap to multi bitmap file (gSaveMultiRect)	4-32
End multiple bitmap save and close file (gEndMultiSave)	4-32
Inquire screen or bitmap checksum (gInquireChecksum)	4-32
Adding graphics output functions	4-33
Load a DYL (wLoadDYL)	4-33
Call a DYL function (wCallDYL)	4-33
Call a DYL enquiry function (wCallDYLReply)	4-33

5 Events.....5-1

Getting the next event	5-1
Wait for an event (wGetEventWait)	5-1
Asynchronously request an event (wGetEvent)	5-2
Asynchronously request selected events (wGetEventSpecial)	5-2
Change event types selected (wGetEventUpdate)	5-2
Event types	5-3
Key press event (WM_KEY)	5-3
Redraw event (WM_REDRAW)	5-6
Background event (WM_BACKGROUND)	5-6
Foreground event (WM_FOREGROUND)	5-6
Cancellation event (WM_CANCELLED)	5-6
User message event (WM_USER_MSG)	5-7
Machine switched on event (WM_ON)	5-7
Command received from another client (WM_COMMAND)	5-7
Inform shell of process termination (WM_TASK_UPDATE)	5-7
Inform application key handler (WM_TASK_KEY)	5-7
Change of date event (WM_DATE_CHANGED)	5-7
Escape-key event (WM_ESCAPE)	5-8
Deiconisation event (WM_DEICONISE)	5-8
Attachment event (WM_ATTACHED)	5-8
Detachment event (WM_DETACHED)	5-8
Keyboard state change event (WM_KEYBOARD_STATE_CHANGE)	5-8
Mouse events	5-9
Mouse event (WM_MOUSE)	5-9
Start rubber band (WM_RUBBER_BAND_INIT)	5-9
Complete rubber band (WM_RUBBER)	5-10
Activation event (WM_ACTIVE)	5-10
Event functions	5-10
Request a cancel event (wCancelGetEvent)	5-10
Request a user event (wUserMsg)	5-10
Send a command to another client (wSendCommand)	5-10
Get a command from another client (wGetCommand)	5-10
Enable the reception of WM_ON events (wInformOn)	5-11
Enable/disable the reception of WM_ON events (wInformOnAll)	5-11
Capturing keys	5-11
Capture a key (wCaptureKey)	5-11
Cancel key capture (wCancelCaptureKey)	5-12
Setting task switch keys	5-12
Set a task switch key (wSetTaskKey)	5-12
Cancel a task switch key (wCancelTaskKey)	5-12
Set a back task switch key (wSetBackTaskKey)	5-13
Cancel a back task switch key (wCancelTaskKey)	5-13
Capturing the mouse	5-13
Capture the mouse (wCaptureMouse)	5-13
Release the mouse (wReleaseMouse)	5-13
The rubber band	5-14
Rubber banding (wRubberBand)	5-14
WM_RUBBER events	5-15
Capturing mouse and keyboard events	5-15

6 Window Server Reference Update	6-1
Screen sizes.....	6-1
Keyboard.....	6-1
Status windows	6-3
Series 3c.....	6-3
Siena.....	6-3
Clocks.....	6-4
Siena.....	6-4
Series 3c.....	6-4
Window Server versions.....	6-4

CHAPTER 1

INTRODUCTION

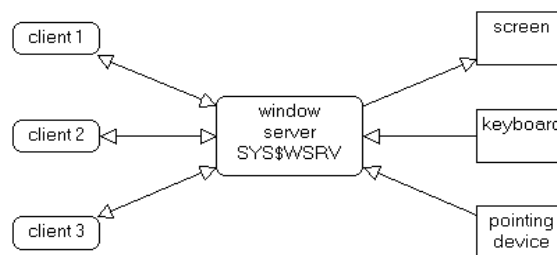
Before reading this manual you should read (at least) the introductory chapter of the *PLIB Reference* manual.

This manual describes the C functions in the WLIB library that access the services of the *window server*.

This manual completely supersedes previous versions and takes account of:

- version 4 of the window server
- the Series 3a and Series 3 corporate computers

The window server is a system process (with process name *SYS\$WSRV*) that runs under the EPOC operating system to provide "client" processes with shared access to the screen, keyboard and (if there is one) a pointing device.



A server process is required because EPOC is a *single-user* multi-tasking operating system as opposed to a multi-user system. On a single-user system, all the application processes must share the same screen and keyboard. Using a server process to share a resource is a common technique on multi-tasking operating systems¹ and EPOC has particularly good support for client-server architectures (see the chapter *Processes and Inter-Process messaging* in the *PLIB Reference* manual).

Having connected to the window server, a client requests services by sending the window server an inter-process message. However, the application programmer does not program at the message passing level but uses the C function interface described in this manual.

Versions of the window server

This section summarises the differences between window servers that run on the following computers:

- the HC range of hand-held corporate computers
- the Series 3 palm-top computer (abbreviated to S3)
- the Series 3a palm-top computer (abbreviated to S3a)
- the *Workabout* corporate hand-held computer
- the MC range of notebook computers (MC400 and MC200)

¹File servers and printer servers are commonly used to share centralised mass storage devices and printers respectively on, for example, a network. A file server is also used in EPOC (with process name *SYS\$FSRV*, described in the *Files* chapter of the *PLIB Reference* manual).

As well as different versions on different machines, the process of continuous development has produced four generations of the window server:

version 2	found on MCs with ROM versions less than 2.40
version 3	found on HCs with ROM versions less than 1.50
version 3.5	found on S3s, HCs with ROM versions of 1.50 or greater and on MCs with ROM versions of 2.40 or greater
version 4	found on the S3a and <i>Workabout</i> .

You can obtain the machine type and whether it is version 2, 3 (which includes 3.5) or 4 of the window server from the information returned by `wConnect`.

Version 4

The following lists the changes for version 4 of the window server:

Task keys now cycle round all instances of applications	On the S3a, where more than one instance of an application is running, pressing the <code>TASK</code> key for that application (ie one of the membrane keys) causes control to cycle round from one instance to the next, bringing each one into foreground in turn; the <code>SHIFT+TASK</code> key cycles round in reverse. Where the S3a is running in S3 compatibility mode, the diamond key must now be used to switch modes. This key generates a <code>W_KEY_MODE</code> event.
Double pixel mode	All graphics commands now handle drawing with double sized pixels. This feature is designed for the S3a to run in S3 compatibility mode. It is also available in individual windows by setting the <code>W_WIN_DOUBLE_PIXEL</code> flag; this causes all graphics in that window to work in double pixel mode. A new flag <code>G_GC_FLAG_DOUBLE</code> can be set in a graphics context (GC) to draw in double pixel mode. Note that the <i>Workabout</i> does not use double pixel mode when running in S3 compatibility mode.
Bitmaps	A set of new bitmap functions are available that open a bitmap file and permit bitmaps within the file to be loaded or drawn directly. Multiple bitmaps can be loaded from a file without needing to close and reopen it for each bitmap. Part or all of a bitmap can be drawn directly from the file.
Merging of HC and S3 features	Some HC features from the later releases of version 3.5 of the window server have now been added to the S3a variant of the window server. These are: - the window server system flags set by calls to <code>wSystem</code> that were previously relevant to the HC but not the S3a, are now relevant to both. The initial value of these flags is loaded from the environment variable <code>\$WS_FL</code> as in the HC variant.
<code>\$WS_FNTS</code> environment variable	The environment variable <code>\$WS_FNTS</code> contains a series of words each of which contains the index of a font used by the window server. The fonts are as follows: • System font • Notifier/Alert font • Status Window font • Symbols font used for the status window diamond symbol • Medium 2 digital clock font • Medium 2 date font • Notifier/alert button font • Small status window clock font

Font ID indirection	Font IDs based on <code>WS_FONT_BASE</code> will be available for this and all future versions. Font IDs based on <code>WS_FONT_BASE2</code> are for internal use only. Developers may use these fonts but NO guarantee is given that they will be available in future versions.
Keyboard/event buffer size	The keyboard/event buffer size has been increased from 8 to 16 events per application.
Date changed event	Whenever the date changes, a <code>WM_DATE_CHANGED</code> message is delivered to any application in non-compatibility mode on the S3a while in foreground. Applications in non-compatibility mode in background will receive the message when they come to foreground. If this event occurs while the machine is off, the message will be delivered when the machine is next switched on.
Default icon ID	A new <code>#define</code> called <code>WS_DEFAULT_ICON</code> has been supplied to identify the built-in default icon; the grey part of the icon ID is <code>WS_DEFAULT_ICON+1</code> .
Grey	Grey is available on the S3a and <i>Workabout</i> (in one shade only). At a software level this has been implemented by introducing the concept of a plane. There are two planes to which drawing can be directed, both of which have the usual background attributes. The normal plane can be thought of as being the 'original' plane, handling black. The grey plane as its name implies handles the display of grey. Drawing is normally done to one or both planes. However, the two planes are not entirely independent; for example, to display grey, the normal plane should be clear and the grey plane set. If a pixel in the normal plane is set, it is displayed black regardless of the grey plane setting. This is best thought of as the normal plane 'overlying' the grey plane. Appropriate values in a new flags field in the <code>G_GC</code> struct cause graphics to be directed either to the grey plane or to both planes.
New window background modes	New background modes have been added to the normal plane: <code>W_WIN_BACK_CLR_NO_REDRAW</code>, <code>W_WIN_BACK_SET_NO_REDRAW</code> and <code>W_WIN_BACK_NONE_NO_REDRAW</code>. The <code>...NO_REDRAW</code> modes have the effect of telling the window server to set the background for the plane but that no drawing (or re-drawing) will be done to it. Note that <code>W_WIN_BACK_CLR</code> is the default background mode for the normal plane. Similarly, the grey plane has corresponding background modes: <code>W_WIN_BACK_GREY_CLR_NO_REDRAW</code>, <code>W_WIN_BACK_GREY_SET_NO_REDRAW</code> and <code>W_WIN_BACK_GREY_NONE_NO_REDRAW</code> as well as the expected <code>W_WIN_BACK_GREY_CLR</code>, <code>W_WIN_BACK_GREY_SET</code>, <code>W_WIN_BACK_GREY_NONE</code> and <code>W_WIN_BACK_GREY_BITMAP</code>. Note that the default background mode for the grey plane is <code>W_WIN_BACK_GREY_CLR_NO_REDRAW</code>.
Scrolling/Moving windows	Scrolling and moving windows has been optimised so that a plane is only moved if it has drawing enabled. Drawing to a plane is disabled by specifying any of the background modes: <code>...NO_REDRAW</code> .
Alert changes	All existing calls to <code>wsAlertw</code> appear in a new style. The maximum number of message lines can be increased to four by OR'ing the <code>WS_ALERT_B</code> flag into the mode parameter and by building an appropriate text string.
Load additional functions	Additional graphics primitives can be loaded from a dynamic library (DYL). The associated functions are <code>wLoadDYL</code> , <code>wCallDYL</code> , <code>wCallDYLReply</code> .
Sprites	These are animated sequences of bitmaps which are 'connected' to a window. Unlike the pre-version 4 animated sequences, a sprite is not displayed as part of the window background. The window server takes care of saving and restoring the contents of the underlying display. This can give the impression of the sprite 'floating' above the underlying display.

In addition to or in relation to the above mentioned topics, the following functions are new to version 4:

wCompatibilityMode	Sets or cancels S3 compatibility mode on the S3a and <i>Workabout</i> .
gInitBit	Opens a bitmap file ready for calls to gGetBit or gDrawBit.
gGetBit	Loads a bitmap from the bitmap file opened by gInitBit.
gDrawBit	Loads the relevant scan lines (one at a time) from an opened bitmap file for drawing. In effect, it performs a gGetBit, gCopyBit and wFree.
gQueryBit	Queries the size of a bitmap within a file opened by gInitBit.
wFree	Either: closes the bitmap file opened by gInitBit but leaves loaded any bitmaps that came from that file or: frees a sprite.
gInitMultiSave	Opens and initialises a multiple bitmap file ready for calls to gSaveMultiBit, gSaveMultiRect and gEndMultiSave.
gSaveMultiBit	Saves a bitmap to a multiple bitmap file.
gSaveMultiRect	Saves part of a bitmap to a multiple bitmap file.
gEndMultiSave	Ends a multiple bitmap save and closes the file which was opened by gInitMultiSave.
gInquireChecksum	Calculates the checksum of a screen, a bitmap or backed up window.
gOpenFontIndex	Opens a font from a multiple font file as created by version 2.00 upwards of the font compiler or opens a font from a single font file.
gConfigureFonts	Configures a font group.
gReadFontGroupHeader	Reads a font group header from a font file.
gReadFontHeader	Reads a font header from a font file.
wSetSystemFont	Sets the system and other internal fonts.
wsCreateClock2	This is an enhanced version of wsCreateClock and allows the creation of new clock types.
wStatusWindow	Sets the state of the permanent status window.
wsSetList	Sets the list of modes to be displayed in the status window.
wsSelectList	Specifies the new position for the diamond symbol in the status window list.
wInquireStatusWindow	Returns the state and extent of the various status windows available and the status window currently displayed.
wInquireCompatibility	Returns the current state of the compatibility flags of the calling client.
wSupportInfo	Returns information on supported features such as grey and compatibility mode.
wDisablekeyClick	Disables the key click for an application. After setting this state, the window server disables the key click while the application is in foreground and resets it when the application goes to background.
wGetEventSpecial	Asynchronously requests selected events. This is an enhanced version of wGetEvent and allows the selection of the type of event to be delivered.
wGetEventUpdate	Changes the event types previously enabled in an outstanding wGetEvent or wGetEventSpecial call.

<code>wSetPriorityControl</code>	Sets process priority handling to on or off.
<code>wInformOnAll</code>	Enables or disables the reception of <code>WM_ON</code> events. If enabled, <code>WM_ON</code> events are delivered to a client whether it is in foreground or background.
<code>gShadowText</code>	Prints shadowed text.
<code>gBorder2Rect</code>	Draws a shadowed border within a specified rectangle.
<code>gBorder2</code>	Draws a shadowed border as for <code>gBorder2Rect</code> but lying within a rectangle which is set to the whole window or bitmap.
<code>gDrawObject</code>	Draws a graphics object scaled to fit within a specified rectangle.
<code>wDrawButton2</code>	Draws a button. The function can draw both the old S3 style buttons and the new S3a style buttons.
<code>wCreateSprite</code>	This function creates a sprite for a given window, a sprite being a sequence of up to 13 bitmap sets. Each bitmap set can have up to 6 bitmaps, 3 for the normal plane and 3 for the grey plane.
<code>wSetSprite</code>	Changes the bitmap sets and position of a sprite.

The following functions have been enhanced or changed:

<code>wSystem</code>	New flags have been added to allow certain features of the status window to be disabled: <code>WSERV_FLAG_SW_NO_LOW_BATTERY</code> , <code>WSERV_FLAG_SW_NO_PACKS</code> , <code>WSERV_FLAG_SW_NO_LINK</code> , <code>WSERV_FLAG_SW_NO_CAPS</code> .
<code>wTextCursor</code>	A new flag <code>W_CURSOR_GREY</code> can be set in the <code>flags</code> member of the <code>W_CURSOR</code> struct to make the text cursor appear grey.
<code>wSetWinBitmap</code>	A member of a bitmap sequence can be made to appear grey by OR'ing the <code>WS_WIN_BITMAP_GREY</code> flag into the <code>mode</code> member of the <code>WS_WIN_BITMAP</code> structure.
<code>gFillPattern</code> <code>gCopyBit</code>	Both functions now have a special way of handling grey when working on backed up windows with grey planes.
<code>gPeekBit</code>	Setting the most significant bit of the window/bitmap ID parameter causes the "peeking" to select the grey plane.
<code>gSaveBit</code>	This function will now save a double bitmap when saving the screen or a backed up window with a grey plane.

Version 3.5

The following lists the changes for version 3.5 of the window server:

Notifier can be handled by the window server	where the window server optionally hooks the <code>p_notify</code> and <code>p_notifyerr</code> calls (removing the need for the <code>SYS\$NTFY</code> process). Can also (optionally on the HC) take on the responsibility for automatically notifying the user of a process that terminates with a panic or a negative reason code. The window server also provides the notifier-like alert services <code>wsAlertw</code> and <code>wsAlertA</code> . Only available on HC and S3.
Going deaf	On the HC and the S3, the window server now presents a "Hung up" status window when the foreground client does not respond ("listen") to redraw events (within 5 seconds). On the MC (and previously on the HC), the window server "slimes" invalid areas with a grey pattern.
Setting task keys	where the window server can be set to respond to two keys that cycle through the tasks in the two directions (see <code>wSetTaskKey</code> , <code>wSetBackTaskKey</code> , <code>wCancelTaskKey</code> and <code>wCancelBackTaskKey</code>). These task keys are in addition to the normal system task keys on the HC and MC and this feature is provided for the S3 (which does not have a system task key on the keyboard).

Password support	where the window server can be set to activate the password alert when the machine is switched on. Only available on HC and S3.
Faster text drawing	where a font that has no character wider than 8 pixels and is represented in a special fixed-width form is drawn significantly faster.
Mono and italic font styles	A proportional font may be drawn mono-spaced (G_STY_MONO). A font may be drawn italicised by shifting the top half of each character across by one pixel (G_STY_ITALIC).
Info/busy messages	Info messages (see <code>wInfoMsg</code> and <code>wInfoMsgCorner</code>) are transient messages displayed in a corner of the screen. A flashing busy message may be displayed using <code>wSetBusyMsg</code> . Can also (optionally on the HC) take on the responsibility for using an info message to inform the user of a low battery when the machine is switched on. Only available on HC and S3.
Status window drawing	where a task status window is drawn by the window server as controlled by calls to <code>wsEnable</code> , <code>wsDisable</code> , <code>wsEnableTemp</code> and <code>wsDisableTemp</code> . Intended primarily for the S3 but available also on the HC. Not available on the MC.
Clock drawing	where an analogue or a digital clock image is redrawn and updated automatically by the window server (see <code>wsCreateClock</code> and <code>wsSetClock</code>). Only available on HC and S3.
Inter-client commands	A client of the window server can send up to 127 bytes of data to another client (see <code>wSendCommand</code> , <code>wGetCommand</code> and the <code>WM_COMMAND</code> event). On the S3, this provides the mechanism by which the System task (SYS\$SHLL) sends commands to application tasks to, for example, save their data and exit.
Link paste handled by the window server	On the S3 and HC, the window server handles the services that are needed to link paste data from one application to another (provided by SYS\$SHLL on the MC).
Terminating clients	When a client disconnects, the window server makes the client's owner foreground (the owner is the last process to <code>p_presume</code> the disconnecting client). If the owner cannot be found, the shell is made foreground. Applies to the HC and S3 only.
SYS\$WSRV.\$05	The task that handled keyboard events no longer exists on the HC and S3 versions, causing most process numbers to start one lower than before.
In addition to those mentioned above, the following miscellaneous functions are new to version 3.5:	
<code>wSystem</code>	Changes the system wide behaviour of the window server. For example, to set whether the window server automatically reports a process terminating with a panic.
<code>gTextCount</code>	Calculates the number of characters that can be displayed in a given pixel width.
<code>wGetWidthTable</code>	Gets the font width table.
<code>wTextCursor</code>	An enhanced version of <code>wTextCursor</code> , supporting an obloid text cursor and the disabling of the cursor flash.
<code>wInformOn</code>	If the foreground client has called this function, the window server sends it a <code>WM_ON</code> event when the machine is switched on.
<code>wDrawButton</code>	Draws an S3 style button, depressed or otherwise. Not available on the MC.
<code>gSetOpenAddress</code>	Used immediately before <code>gOpenFont</code> , <code>gOpenBit</code> or <code>gOpenMouseIcon</code> to load the font, bitmap or mouse icon from anywhere within the file.
<code>wGetProcessList</code>	Returns an array of the process IDs of the clients of the window server.
<code>wAppKeyHandler</code>	Special handling of the S3 membrane keys. Intended for use by the S3 System task only.

The following functions have been enhanced:

gBorder gBorderRect	These functions take the following new flags: W_BORD_CORNER_1, W_BORD_TOP_ON, W_BORD_TOP_OFF, W_BORD_BOT_ON, W_BORD_BOT_OFF.
wConnect	Takes the new W_CONNECT_DISABLE_LEAVES flag. Equivalent to calling wDisableLeaves(TRUE) except that it also affects whether wConnect itself leaves or returns an error.
wDisableLeaves	Now returns the old state.

Version 3

The following lists the changes for version 3 of the window server:

Backed-up windows	where a window may optionally be created (as determined by a parameter to wCreateWindow) such that any drawing to it is also drawn to a back-up bitmap. Many of the functions that operate on bitmaps can also operate on the backup bitmap through the ID of the window it is backing up.
Capturing keys	where a client may capture specific keys regardless of whether it has the foreground or not; provided by wCaptureKey and wCancelCaptureKey.
System modal clients	where a client may alter its system modal state; provided by wSystemModal and wCancelSystemModal (a client can connect as a system modal client in any version).
Pause key	where the foreground client can be paused by the user pressing the pause key. The pause key may be enabled and disabled using wEnablePauseKey and wDisablePauseKey.
Windows	A function wGetWindowPosition gets the position of a window in its sibling list. An obscure function wReassignRootWindow reassigns the root window (only used for development).
Graphics output	Additional graphics output functions for improved screen aesthetics: gInvObloid, gXPrintText, gBorderRect, gBorder and gCheckBitmapID.

All the above are available on version 3 and 3.5 of the window server.

Specific to the MC

Console implemented by window server	On the HC, S3 and the S3a, the console device LDD CON: is implemented by connecting directly to the window server. On the MC, the console device is implemented by an intermediate SYS\$CONS display process that draws and redraws the console window.
Pointing device	The window server supports a pointing device only on the MC. The functions associated with a pointing device are: wCaptureMouse, wReleaseMouse, gOpenMouseIcon, wRubberBand.
View multiple tasks	On the MC only, the windows belonging to more than client process may simultaneously be visible on the screen. The functions associated with viewing multiple tasks are: wClientIconised, wAttachToClient, wAttachToForegroundClient, wDetachClient.
Load additional functions	On the MC only, additional graphics primitives can be loaded from a dynamic library (DYL). The associated functions are wLoadDYL, wCallDYL, wCallDYLReply.

Using WLIB

On all machines other than the MC, the WLIB library may be used in conjunction with:

- the CLIB library (with the CLIB startup module)
- with both CLIB and PLIB (with the CLIB startup module)
- the PLIB library (with the PLIB startup module)

Because the console is implemented by a separate process on the MC, you can't reasonably use the WLIB functions with the CLIB startup module. On the MC, we recommend using the PLIB library with the PLIB startup module (as described in more detail below).

The WLIB header

To get the constants and typedefs for the functions in the WLIB library, you should include:

```
#include <wlib.h>
```

as well as the header files for using CLIB and/or PLIB.

Connecting to the window server

To use the services of the window server, a process must first connect to it by calling `wConnect` (or a function that calls `wConnect` such as `wStartup`).

Not all processes are clients of the window server but an application process that is presenting a user interface is likely to be a client².

How you connect to the window server depends on whether you are using the CLIB or the PLIB C startup module and what machine you are running on.

Note that the default project files set up by the installation of the SDK use the CLIB startup module.

Using the CLIB startup module on the HC, S3, S3a or Workabout

The CLIB startup module automatically opens a channel to the console device `CON:`. This channel is used to implement the CLIB functions that access the screen display and the keyboard. For example, such functions as `printf`, `gets`, `cprintf` and `cgets`.

On an HC and all Series 3 machines, the console device is implemented such that opening it connects to the window server, making the process a client of the window server. When using the CLIB startup module, you take advantage of this console connection - as described in this section. If you mistakenly attempt to connect a second time by calling `wConnect` or `wStartup` in your program, the process will be panicked with panic number 100.

When the CLIB startup module opens `CON:`, it puts the channel in the static variable `winHandle`, which may be referenced as:

```
GLREF_D VOID *winHandle;
```

As well as connecting to the window server, opening a channel to the console creates and initialises a backed-up window (which does not have to be redrawn).

As described in the *Console* chapter of the *I/O Devices Reference* manual, you can obtain the ID of the console window using the `F_INQ` I/O function, as in the following program:

```
#include <plib.h>
#include <p_cons.h>
#include <wlib.h>
```

²One exception to this rule is on the MC where an application uses the services of the console process (with process name `SYS$CONS`) to access the screen and keyboard. In this case it is `SYS$CONS` that is the client of the window server - not the application process. On the MC, an OPL program (which is really a process of `SYS$OPLR`) uses `SYS$CONS` to draw to the screen.

```
GLDEF_C INT main(VOID)
{
    CONSOLE_INFO cinfo;
    WS_EV event;

    p_iow(winHandle, P_FINQ, &cinfo);
    gCreateGC0(cinfo.window_handle);
    gPrintText(10, 20, "Hello world!", 12);
    do
    {
        wGetEventWait(&event);
    } while (event.type != WM_KEY);
    return(0);
}
```

where `cinfo.window_handle` is the console window ID. After displaying the "Hello world!" message, the program waits for a key press event and then exits.

In CLIB programs you can prevent the automatic opening of a console channel by defining the function `p_xwind` in your code, as in the following example:

```
extern void *winHandle;

void p_xwind(void)
{
    winHandle=(void *)1;
}

int main(void)
{
    ...

    return(0);
}
```

You should ignore the warning, given during the linking of your program, that the symbol `_p_xwind` is duplicated.

If you use this technique, your CLIB program should not, of course, make any reference to `stdin`, `stdout` or `stderr`, unless you have redirected them. Setting `winHandle` to 1 (an illegal value for a handle) will guarantee that any such reference will fail with a panic.

Using the PLIB startup module on the HC, S3, S3a or Workabout

The PLIB startup model does not open a channel to the console and the simplest way to get going is to use the `wStartup` function that:

- connects to the window server
- creates and initialises a backed-up window to cover the whole screen
- creates a permanent graphics context on that window

After calling `wStartup`, you are in position to draw to the graphics context - as in the following example:

```
#include <plib.h>
#include <wlib.h>

GLDEF_C INT main(VOID)
{
    WS_EV event;

    wStartup();
    gPrintText(10, 20, "Hello world!", 12);
    do
    {
        wGetEventWait(&event);
    } while (event.type != WM_KEY);
    return(0);
}
```

Using the CLIB startup module on the MC

On the MC, the console device is implemented quite differently from the HC. When the `CON:` device is opened on an MC, a console display process is created by loading the `SYS$CONS.IMG` executable from the ROM. When the opener of the console device calls the channel's I/O functions, the console sends inter-process messages to the console display process which then sends inter-process messages to the window server. The console process keeps a character map of the console screen which it uses to redraw its window as necessary. The console process also manages a title bar that allows the window to be moved and/or resized and a menu bar that allows the program to be stopped.

Because the console device is implemented in this way, the console does not support the `P_FINQ` I/O function as described above for using the CLIB startup module on the HC. Because it is the console and not the application process that is the client of the windows server, you can't use the console's window. There is nothing to stop you from connecting to the window server using `wStartup` or `wConnect` but you will then be running an application with two independent window systems and two clients. The console channel is still in `winHandle` so you can close the console and terminate the display process but this is hardly satisfactory.

In conclusion, on the MC, you are better off embracing the EPOC system more completely and using the PLIB startup module. Despite what is said elsewhere, you can use *some* CLIB functions with the PLIB header - those that do not rely on any initialisation. You can, for example, use `strcpy` - but you can't use any I/O function such as `open`, or a memory allocation function such as `malloc`.

Using the PLIB startup module on the MC

As on the HC, the simplest way to get going is to use the `wStartup` function.

The MC is available with both version 2 and 3.5 of the window server. Version 2 does not support bitmap backed-up windows. Even with version 3.5 on the MC, backed-up windows are less attractive on a large screen where the bitmaps consume large amounts of memory and the processing overhead of maintaining the bitmap is likely to be more noticeable.

You can still call `wStartup` but, with version 2 of the window server, you get a window that is not backed-up and you have to deal with redraw events, as in the following example:

```
#include <plib.h>
#include <wlib.h>

GLREF_D UINT wMainWid;

GLDEF_C INT main(VOID)
{
    WS_EV event;

    wStartup();
    do
    {
        wGetEventWait(&event);
        if (event.type==WM_REDRAW)
        {
            wBeginRedrawWin(wMainWid);
            gPrintText(10,20,"Hello world!",12);
            wEndRedraw();
        }
    } while (event.type!=WM_KEY);
    return(0);
}
```

The global variable `wMainWid` contains the ID of the window that is created by `wStartup`.

On the MC with its larger screen, `wStartup` is probably too simplistic (for example, it creates a window the full size of the screen) and is best seen as a quick starting point for exploratory programming. In due course, you should look to using `wConnect` to implement a startup that is appropriate to your application.

Error handling

How errors are signalled

Should an error (such as out of system memory) occur in one of the window server functions, the window server will do one of the following:

- call `p_leave`, passing it the (negative) error number
- return the error number

By default, the window server calls `p_leave`. You can make it return the error number by calling `wDisableLeaves(TRUE)`. From version 3.5 onwards of the window server, you can also set the `W_CONNECT_DISABLE_LEAVES` flag when calling `wConnect`. This is equivalent to calling `wDisableLeaves(TRUE)` except that it also affects whether `wConnect` itself leaves or returns an error.

The enter and leave mechanism (which uses `p_enter`; and `.p_leave`) is commonly used to implement structured error recovery. See the *Error Handling* chapter of the *PLIB Reference* manual.

With the enter and leave mechanism, a call to `p_leave` should only occur within the protection of a `p_enter` harness. If you don't use `p_enter` and you don't call `wDisableLeaves` and `p_leave` is called, the process will be panicked with panic number 47.

Errors in blind operations

In the interests of performance, many of the functions performed by the window server have a "blind" interface in the sense that the client does not receive any acknowledgement that the operation has been performed.

Blind operations may be stored in a client-side buffer to be processed in batches when some condition causes the buffer to be flushed (as described later). The functions that perform blind operations don't return anything and are declared as `void`.

Many of the blind operations can't reasonably fail - such as drawing a line by calling `gDrawLine`. However, some functions can fail. For example, increasing or even decreasing the size of a bitmap backed-up window by calling `wSetWindow` can fail to allocate the additional memory (which is required transiently when decreasing the size of a backed-up window).

If an error occurs in a blind operation, the window server goes into a special state in which it discards all further blind operations until a function that can fail is called (such as `wCreateWindow`).

When such a non-blind function is called, it immediately fails - either by calling `p_leave` or by returning an error number, as described above.

When a failure does occur while drawing, you often don't care exactly where it failed and you can happily wait until a non-blind function is called (which might mean waiting until `wGetEventWait`, `wGetEvent` or `wGetEventSpecial` is next called).

However, if you need to establish that the processing has been successful thus far, you can call `wCheckPoint` which flushes the buffer and signals any failure (by calling `p_leave` or by returning an error number).

Cleaning up after an error

When handling an error while using the window server (say in response to a `p_leave`), you can use `wCleanup` to clean up any dangling window server resources where `wCleanup`:

- frees the temporary graphics context (if it exists)
- ends a redraw if one was in progress

If there is a current graphics context that is attached to a window, `wCleanup` also invalidates that window so that it is not left in a partly drawn state.

Panic numbers

See the *Error Handling* chapter of the *PLIB Reference* manual for a discussion of panics and panic numbers.

The window server panics a client that attempts an illegal operation, using the following panic numbers:

81	font does not exist
82	illegal window or bitmap ID
83	illegal window ID
84	null handle given to server
85	illegal graphics context ID
86	illegal GMODE value (V2 only)
87	illegal TEXTMODE value (V2 only)
88	illegal font ID (V2 only)
89	wBeginRedraw called while already in a redraw
90	mouse icon does not exist
91	illegal bitmap ID
92	window tree is already initialised
93	wEndRedraw called when there isn't a redraw to end
94	attempted to change the background of a backed-up window
95	wInitialiseWindowTree called when the <i>parent</i> window is not initialised
96	illegal parameters passed to wsAlertW, wsAlertA or wsAlertUpdate
97	illegal length in gPeekBit (V2 only)
98	illegal x+length value in gPeekBit (V2 only)
99	illegal ypos in gPeekBit (V2 only)
100	tried to connect a second time
101	tried to access a permanent graphics context while a temporary graphics context exists
102	illegal opcode in message
103	command buffer received by wserv is too long
104	generally bad message received
105	wFree was called with an ID that doesn't refer to a freeable object
106	illegal DYL ID
107	out of range count sent to wSetWinBitmap
108	a bitmap was freed while still in use by a wSetWinBitmap command
109	illegal window-bitmap ID
110	bad data or version in connect message
111	called wGetEvent while the previous call was still pending
112	function not available
113	illegal clock ID in wsSetClock
114	illegal sprite ID (V4 only)
115	client already has a sprite (V4 only)
120	corrupt control block (possibly not connected)
121	function number out of range

Note that because of client-side buffering, there may be a gap between calling the offending function and the call to `p_panic`. When tracking down the cause of a window server panic, you may need to insert temporarily calls to `wFlush` or `wCheckPoint` to precipitate the panic.

Series 3 compatibility mode.

To enable existing applications that were designed to run on the S3, to run successfully on the S3a and *Workabout* while keeping the same 'look' and 'feel', version 4 of the window server can run in what is called compatibility mode for that application. This mode is set or cleared by use of the `wCompatibilityMode` function.

As far as is practicable, the window server on the S3a attempts to emulate its behaviour on the S3. For example, in compatibility mode, all drawing to the screen is done in double pixel mode to overcome the fact that the S3a has a screen of 480 x 160 pixels compared to the S3 screen of 240 x 80 pixels.

When running in compatibility mode on the S3a or *Workabout*, the `version_id` member of the `CONNECT_INFO` sub-structure will have `WS_VERSION_4` set. This constitutes the only difference between the S3, S3a and *Workabout* in compatibility mode.

The *Workabout* has two different types of compatibility mode. In the first type, a 240 x 80 S3 display is centred on the screen, leaving an unused area above and below the display. In this type of compatibility mode, the only difference between the S3 and the *Workabout* is the `version_id` member of `CONNECT_INFO`, as described above.

In the second type of compatibility mode for the *Workabout*, the display covers the full 240 x 100 *Workabout* screen and uses a restyled status window. This type is intended for use by only those applications that can adjust the dimensions of their windows to match the available screen size. It is recommended that this type of compatibility mode should be used only if a centred 240x80 display is truly unacceptable.

References to compatibility mode will occur throughout this manual.

Clients and the window server

Client-side functions

In the interests of performance, not all the window server functions cause an inter-process message to be sent.

For example, the function `gTextwidth`, which calculates the printed width of a text string, is implemented entirely on the "client side" without requiring any context switch between client and window server.

Client-side buffer and flushing

Also in the interests of performance, functions that perform "blind" operations that have no return values (such as most drawing operations) are not sent directly to the window server but are queued in a client-side buffer.

In most cases, the client-side buffer is flushed automatically, when:

- the buffer is about to overflow
- an operation that requires a return value is requested
- an input event is requested by calling `wGetEventWait`, `wGetEvent` or `wGetEventSpecial`

Applications should not, however, make any assumptions as to whether a particular window server function call will or will not cause the buffer to be flushed.

The client-side buffer can be flushed explicitly (for example, to animate an image) by calling `wFlush`. In practice, it is rarely necessary to use `wFlush` and, although otherwise harmless, using `wFlush` unnecessarily will degrade performance.

As described earlier, you can also flush the client-side buffer by calling `wCheckPoint`.

The client-side buffer is allocated from the heap when `wConnect` is called and is approximately 300 bytes long. The address of the data structure that contains the client-side buffer is held in the reserved static variable `wClientData` which may be referenced from C by declaring:

```
GLREF_D VOID *wClientData;
```

Testing for a connection

If a process has connected to the window server, `wClientData` will contain a non-zero value. If a process has not connected to the window server, it will contain zero.

Foreground and background clients

Of all the clients of the window server one client is the *foreground* client and all the other clients are *background* clients³.

The foreground client is the client that receives keyboard input from the user.

Some special key presses (such as the TASK key as described below) are processed by the window server. A client can capture specific key presses (in which case it is sent the key whether it is foreground or not) by calling `wCaptureKey`.

The foreground client has its windows in front of the windows of any background client. On small screen versions of the window server, that is, on the HC, S3, S3a and *Workabout*, the windows of any background clients are not visible at all.

Events

For each client, the window server keeps a queue of events that inform the client of user input and other events. The different types of window server events include:

- key presses
- foreground/background changes
- redraw events (described later)
- mouse events (if the machine has a pointing device)

Mouse and key events are time stamped with a 16-bit time in system ticks which may be used to calculate the time between successive events that occur within a short time period (such as a double click).

Redraw and mouse events are directed at a particular window by a window handle event parameter. This handle is specified by the client when the window is created (and is commonly the address of a client data structure that contains the window ID).

The client can read the next event by calling `wGetEventWait`, which only returns when there is an event to deliver. If the client's event queue is empty, `wGetEventWait` will wait indefinitely for an event to occur.

Applications that need to respond to events other than just window server events (for example, serial input) would use either of the *asynchronous* functions `wGetEvent` or `wGetEventSpecial` which request an event without waiting. Asynchronous requests are described in the chapter *Asynchronous Requests and Semaphores* in the *PLIB Reference* manual.

Note that `wGetEventSpecial` is only available in version 4 of the window server and is a generalisation of `wGetEvent` in that it permits the caller to select which events are to be delivered. Calling `wGetEventSpecial(WE_EVENT_NORM)` is equivalent to calling `wGetEvent`.

Task switching;

The window server keeps all the clients in a front to back *task order*. Position zero in the task order is the front position and is held by the foreground client. Position 1 is held by the frontmost background task and so on.

When a program connects to the window server, it normally takes the foreground. (A program can connect in background by setting a parameter to `wConnect`.)

On an S3, an S3a or an HC that is running version 3.5 of the window server, when a process disconnects, the window server attempts to make the "owner" of the process foreground. Here, the owner is the process which last resumed it (by calling `p_resume`). If the owner has terminated or is not a client of the window server, the shell is made foreground.

A client may bring itself or any other client to foreground (or put itself or any other client to background) by calling `wClientPosition`.

Unless an application has taken steps to disable task switching, the user may switch tasks using the machine-dependent task-switching keys, as described next.

³On large screen versions of the window server when one or more clients have attached to a client, you can have multiple foreground clients (in the sense that they all last received a foreground event) but only one of them (the frontmost client) receives key events.

Task switching on the HC

On an HC with an alpha-numeric keyboard (as opposed to just numeric), the window server brings the foremost background client (at client position 1) to the foreground when the TASK key (SHIFT+LEFT ARROW) is pressed (the former foreground client is moved to the end of the task list).

On an HC, an application can disable task switching in one of two ways:

- locking itself into foreground by calling `wSystemModal(0)`
- capturing the task key by calling `wCaptureKey`

Task switching on the Workabout

On a *Workabout*, the window server brings the foremost background client (at client position 1) to the foreground when the TASK key (SHIFT+ESC) is pressed (the former foreground client is moved to the end of the task list).

On a *Workabout*, an application can disable task switching in one of two ways:

- locking itself into foreground by calling `wSystemModal(0)`
- capturing the task key by calling `wCaptureKey`

Task switching on the S3 and the S3a

Task switching is controlled by the 8 membrane keys (called application keys) above the main keyboard. Actually, the window server handles 16 application keys where a second set of 8 keys are accessed by pressing the CONTROL shift key.

These keys are handled co-operatively by the window server and the shell. (The shell has process name `SYS$SHELL` and is known by the user as the System task.)

When the system starts up, the shell calls `wAppKeyHandler` to declare itself as the handler of the application keys. The shell maintains two data structures pointed to by the reserved statics `DatApp1` and `DatApp2` which control the assignment of the application keys to particular applications.

When the system starts up, each application key is assigned to an in-built application. Using the shell's user interface, the application keys other than the 2 System application keys (14 in all) may be reassigned.

The application keys are handled as follows:

- On both the S3 and the S3a, if an application key (SHIFTed or otherwise) of an application different from that of the foreground is pressed, the window server makes the frontmost process of that application foreground⁴. If no process of that application exists, the application key handler (that is, the shell) is made foreground and sent a `WM_TASK_KEY` event.
- On the S3 only, if an application key of the same application as that of the foreground is pressed, the window server sends the foreground task a `WM_KEY` event with key code `W_KEY_MODE`. (Applications normally cycle through their display modes in response to this event.)
- On the S3 only, if a SHIFTed application key of the same application as that of the foreground is pressed, the window server brings the frontmost background client of that application to the foreground (the former foreground client is moved to the end of the task list).
- On the S3a under version 4 of the window server, pressing an application key of the same application as that of the foreground brings the frontmost background client of that application to the foreground (the former foreground client is moved to the end of the task list). Pressing the SHIFTed application key of the same application as that of the foreground does the reverse. In response to the DIAMOND key being pressed, the window server sends the foreground task a `WM_KEY` event with key code `W_KEY_MODE`. (Applications normally cycle through their display modes in response to this event.)
Note, however, that a foreground application running on the S3a in S3 compatibility mode will also receive this event when the DIAMOND key is pressed.
- On both the S3 and the S3a, if a PSION shifted application key is pressed, the window server makes the application key handler foreground and sends it a `WM_TASK_KEY` event.

⁴The application is identified by its name. For an S3 or S3a application process, the window server reads the associated application name from the reserved static `DatProcessNamePtr`.

The shell responds to a `WM_TASK_KEY` event (which indicates which key was pressed in a parameter) by positioning to the appropriate icon.

The `SYSTEM` application key is permanently assigned to the shell.

The `CONTROL+SYSTEM` application key is assigned to a notional `RunImg` application such that if this key is pressed, the window server makes the shell foreground and sends it a `WM_TASK_KEY` event. The shell then positions to the `RunImg` icon (a bubble containing the word `IMG`).

The shell calls `wSetTaskKey` to assign `SHIFT+SYSTEM` as a task key. The window server responds to a task key by moving the foreground client to the end of the task list thus bringing the client previously at position 1 to the foreground.

The shell also calls `wSetBackTaskKey` to assign `SHIFT+PSION+SYSTEM` to the "back-task" key which brings the task furthest from the front to the foreground.

On the `S3` and the `S3a`, an application can disable task switching in one of two ways:

- locking itself into foreground by calling `wSystemModal(0)`
- capturing all the application keys key by making 8 calls to `wCaptureKey` (capturing all shift states in each call)

Task switching on the MC

On the `MC`, the windows of a foreground or background client that has a lower task position will, if they overlap, obscure (partially or wholly) the windows of background client with a higher task position.

The user can change the task ordering by:

- pressing the `TASK` to move the foreground client to the end of the task list thus bringing the client previously at position 1 to the foreground
- pressing `SHIFT+TASK` to cycle through tasks in the reverse direction
- pressing `CTRL+TASK` and `CTRL+SHIFT+TASK` that cycle in either direction in such a way that iconised tasks are skipped
- using the digitiser to click on a background task's window

On the `MC`, it is also possible to attach a client to another (by calling `wAttachToClient` or `wAttachToForegroundClient`) such that the attached client and the client it is attached to behave as one task (with the attached client in front).

An application can disable task switching by locking itself into foreground by calling `wSystemModal(0)`.

Iconised clients

On large screen version of the window server such as the `MC`, a client can mark itself as iconised by calling `wClientIconised`.

A client that marks itself as iconised is excluded from a form of window server controlled task switching in which only non-iconised tasks are brought into the foreground. On an `MC`, if the user holds down the `CONTROL` key while pressing the `TASK` key, the window server selects only non-iconised tasks.

If an iconised client is brought to the foreground by a call to `wClientPosition` (normally by another process), the window server sends that client a `WM_DEICONISE` event (which would normally prompt the client to deiconise itself).

Note that it is the client's responsibility to make any changes to its appearance as a result of a change in its iconised state.

Client priorities

The window server can be instructed (by a parameter to `wConnect` or by a call to `wSetPriorityControl`) to adjust automatically the process priority of a client when it gains and loses the foreground such that the foreground client runs at a higher priority than any background clients.

A higher priority foreground client that is performing a computationally intensive task (for example, a spreadsheet program that is calculating) will totally and indefinitely block any lower priority background clients that are ready to run. This can be undesirable - particularly on the `MC` where the windows belonging to background clients may be visible.

To avoid robbing background tasks of all processing, computationally intensive processing should be bracketed with calls to `wStartCompute` and `wEndCompute`. In between the calls to `wStartCompute` and `wEndCompute`, the window server holds the client's priority at the background client level regardless of whether it has the foreground or not.

On machines using the small screen versions of the window server, where the windows belonging to background clients are never visible, the arguments for using `wStartCompute` and `wEndCompute` are less compelling but there may be circumstances when their use is still appropriate. For example, if a user switches from a task which is printing to continue a game of chess, should the foreground chess task halt the background task from printing?

System-modal clients

There is sometimes a requirement to disable task switching by locking a so called *system modal* task in the foreground. For example, to notify the user of a condition that must be acknowledged or rectified before proceeding.

A client may declare itself system modal by a parameter to `wConnect`, or may subsequently change its system modal state by calling `wSystemModal` or `wCancelSystemModal`.

The window server limits task switching to only those processes that have a lower client position than the frontmost system modal task (if there is one).

A client that is system modal would normally be in one of the following client positions:

- take the foreground in which case task switching is disabled
- be furthest in the background (that is, with the highest client position) in which case task switching excludes the system modal task

On larger screen versions of the window server (such as on an MC or a PC) where windows belonging to different clients are typically simultaneously visible on the screen, a system modal client would normally takes steps to make its windows invisible unless it has the foreground (since clicking on them will not bring them to foreground).

Client management

The window server is in a natural position to undertake most of the work necessary to manage clients, for example:

- handling foreground/background task switching
- automatically adjusting client process priorities
- notifying the user of clients which terminate abnormally (on machines other than the MC)
- supporting the link paste mechanism on the S3, S3a and *Workabout*

However, there are a few functions to support a special client (typically the shell, `SYS$SHLL`) taking on some client management. These functions are:

<code>wGetProcessList</code>	Gets the process IDs of the clients of the window server, in front to back order.
<code>wClientPosition</code>	Used to bring a client into the foreground.
<code>wSendCommand</code> <code>wGetCommand</code>	Used to transfer up to 127 bytes from one window server client to another.
<code>wAppKeyHandler</code>	Used by the shell on the S3 and S3a to handle membrane keys (application keys) in partnership with the window server.
<code>wSystem</code>	Modify the behaviour of the window server on a non-client-specific level. For example, to determine whether the window server handles the <code>p_notify</code> service. See also the section <i>System start-up</i> at the end of this chapter.
<code>wSetTaskKey</code> <code>wCancelTaskKey</code> <code>wSetBackTaskKey</code> <code>wCancelBackTaskKey</code>	Sets the window server to respond to additional keys that cycle through the tasks in the two directions. Provided mainly for the S3 and S3a (which do not have a system task key on the keyboard).

Windows

A *window* is a rectangle in *screen coordinates* that provides a coordinate system for clipped drawing.

Once a window has been created using `wCreateWindow`, its initial position and size may be changed (using `wSetWindow`).

Note that at the window server layer, a window is invisible unless it is drawn to. If a window has a boundary (or any other features) it is because the owning client drew it.

Most drawing is done to a *graphics context* to which a window has been assigned. Off-screen *bitmaps* can also be assigned to a graphics context. Both windows and bitmaps are sometimes called *drawables*.

Window trees

Windows are linked in a hierarchy or tree with the screen as the *root window*.

A window is created relative to its *parent window* and is called the *child window* of the parent window. A child window may be a parent of further child windows and so on to any depth.

The position of a child window is held relative to its parent. If a parent window is moved, all descendant windows move by the same amount. You can obtain the offset between any two windows (wherever they are in the hierarchy) by calling `wInquireWindowOffset`.

A child window

- is in front of the parent (and will obscure any drawing to the parent window)
- is clipped to the boundaries of its parent.

Note that a child window may be smaller or larger than its parent. A window is often *tiled* with multiple child windows which are smaller than the parent and where parts of the parent window may or may not be visible depending on whether there are any gaps between the boundaries of the child windows. A window may have a single child window that is slightly smaller than its parent and where the parent provides a frame around the child window - any drawing to the child window is clipped to the boundaries of the child window and will not corrupt the frame. A window may have as its child a larger window that is providing a clipped scrolling view of some information (such as a list) where the view is scrolled by simply moving the child window in its parent's coordinates.

A window is said to be the descendant of a window if it is its child or its grandchild and so on. All windows are descendants of the root window. Child windows of the root window are sometimes called *top-level windows*.

Child windows of the same parent are called *sibling windows*. Sibling windows have a front to back order which is apparent if they overlap.

Ownership of windows

Except for the root window, all windows are owned by a particular client. A client does not have access to windows belonging to other clients and can only create child windows of the root window (that is, top-level windows) or of its own windows.

On larger screen versions of the window server such as on an MC or a PC, windows belonging to different clients may (and typically are) simultaneously visible on the screen.

On small screen versions of the window server, only the foreground client's windows are visible at any time.

Background client drawing

Background clients can still draw to their windows whether they are visible or not and programs do not normally take any special measures to avoid drawing while in background. In fact, applications are typically oblivious of whether they are foreground or background - they just don't get delivered any key presses while in background.

In practice, background clients rarely draw to their windows on a *small screen* version of the window server because they have been robbed of keyboard input. Exceptions are a client that is switched to background while it is still processing and a client that is driven by events other than just key presses - for example a terminal emulation program or a clock program.

Background clients of a *large screen* version of the window server commonly draw to their windows while in background for two reasons. First, to take on the appearance of a background client (otherwise, unless its windows are partially obscured by an overlapping window, it is difficult to differentiate between background and foreground tasks). Second, to *redraw* its windows after being exposed by changes (of position, size, front to back ordering or visibility) to windows that previously obscured it.

Drawing region

When you draw to a window, the drawing is clipped to the visible part of the window - called the *drawing region*. When the window is partially obscured by overlapping windows, the drawing region is not a simple rectangle (it is actually represented by a variable-length array of rectangles).

For example, suppose an application is displaying a clock as part of its main display area. On the completion of a timer informing the application to update the appearance of the clock, the application can happily go through the motions of drawing the entirety of the clock, without worrying whether part of that display is being obscured by an overlapping dialog box or pulled-down menu. The window server ensures that only drawing to unobscured portions of the window is effective:



In the diagram, any drawing to the dotted region in the Clock window will fail to appear.

The window server will discard a drawing region (since it can be recalculated at any time) rather than maintain it unless the window is being drawn to or is assigned to a permanent graphics context.

Backed-up windows

A window can be created in such a way that all drawing to it is duplicated to off-screen bitmaps. The window server can then automatically redraw the window without bothering the client. Prior to version 4, the window needs to be created with a `W_WIN_BACK_BITMAP` background. In version 4 upwards of the window server the window can be created with one or both `W_WIN_BACK_BITMAP` and `W_WIN_BACK_GREY_BITMAP` backgrounds (see *creating and initialising a window* in the *Windows* chapter).

This makes life much easier for the programmer. (Otherwise, the programmer has to respond to *redraw events* and has to adopt a more sophisticated programming style, as described below.)

Prior to version 4, when a backed-up window is created, a single backup bitmap is also created with an appropriate size. This bitmap is initialised with zeros corresponding to a clear screen (white on an LCD).

In version 4 of the window server, when a backed-up window is created which is enabled for drawing both black and grey, two backed-up bitmaps are created with an appropriate size, one for the 'normal' plane and one for the grey plane (see the *Graphics* chapter for a fuller discussion of grey).

If the size of a backed-up window is subsequently increased, the backup bitmaps are also increased and the additional area (to the right and below) is filled with zeros.

If a backed-up window is scrolled using `wScrollRect` or `wScrollWin`, the area that is scrolled in from outside the window is filled with zeros.

The disadvantages of using bitmap backed-up windows are:

- it takes longer to draw the image in the first place (since all drawing is duplicated)
- additional storage is required to store the backup bitmap(s)

Although the original drawing is slower, the window server *redraws* a backed-up window with blinding speed.

Keeping backed-up bitmaps for a screen-sized window incurs a storage cost which varies according to the machine type:

- On the HC (160 by 80 pixels), a backed-up bitmap occupies a modest 1600 bytes.
- On the S3 (240 by 80 pixels), a backed-up bitmap occupies 2400 bytes.
- On the *Workabout* (240 by 100 pixels) using version 4 of the window server, a backed-up bitmap requires 3000 bytes. However, if the window is enabled to use both black and grey, two backed-up bitmaps are needed, thus doubling the space required to 6000 bytes.
- On the S3a (480 by 160 pixels) using version 4 of the window server, a backed-up bitmap requires 9600 bytes. However, if the window is enabled to use both black and grey, two backed-up bitmaps are needed, thus doubling the space required to 19200 bytes.
- On the MC400, a full screen (640 by 400 pixels) bitmap requires 32K bytes.

Slowing down screen drawing is clearly more of a problem on a larger screen and, in summary, using backed-up windows on an HC or an S3 with their smaller screens (and smaller windows) makes more sense than on an MC. Using backed-up windows on an S3a or *Workabout* enabled to draw both black and grey is questionable.

Note, however, that the size of the screen does not limit the size of the window. In most practical situations, the windows will be smaller on a smaller screen model, but it is sometimes useful to create windows that are larger (and possibly much larger) than the screen. For example, when presenting a scrolling view of a map that is much larger than the screen size. With such large windows it would not be desirable to use a backed-up window.

No-redraw windows

As well as backed-up windows, the programmer in search of an easy life should also consider windows created with the `W_WIN_NO_REDRAW` bit set (as a parameter to `wCreateWindow`).

For such windows, the window server does not invalidate the window or generate redraw events. What it does do when the window is partly or wholly invalidated (that is, when the backup bitmaps would have been used if they existed) depends on the background parameter to `wCreateWindow` as follows:

`W_WIN_BACK_CLR` prior to version 4, clears the pixels in the window (this is the default). This is useful, for example, when the window is tiled with child windows but does not itself contain any images.

in version 4, clears the pixels in the normal (black) plane of the window.

`W_WIN_BACK_GREY_CLR` available in version 4 only, clears the pixels in the grey plane of the window.

`W_WIN_BACK_SET` prior to version 4, sets the pixels in the window. This may be used, for example, to implement a black thick border to a child window that is slightly smaller and inset from its parent.

in version 4, sets the pixels in the normal (black) plane of the window.

`W_WIN_BACK_GREY_SET` available in version 4 only, sets the pixels in the grey plane of the window.

prior to version 4, either `W_WIN_BACK_CLR` or `W_WIN_BACK_SET` may be used when a window is entirely covered with its child windows. For example, when a larger child window provides, in conjunction with its parent, a scrolling view over a larger image.

In version 4, in addition to using either `W_WIN_BACK_CLR` or `W_WIN_BACK_SET`, one of the corresponding grey plane attributes `W_WIN_BACK_GREY_CLR` or `W_WIN_BACK_GREY_SET` may also be used, if appropriate.

Note that you should not rely on the fact that a window is always totally obscured by another window (such as a child window) to suppress redraw events since the window server can send unnecessary redraw events when there is insufficient free system memory to maintain update regions. For guaranteed suppression of redraw events you should specify the `W_WIN_NO_REDRAW` bit to `wCreateWindow` - even when a window is always totally obscured.

In version 4 of the window server, redrawing can also be suppressed on a per-plane basis by setting the `..._NO_REDRAW` modes. *This method is much preferred.*

The `W_WIN_NO_REDRAW` bit does not need to be set.

The `..._NO_REDRAW` background flags can be used as follows:

<code>W_WIN_BACK_CLR_NO_REDRAW</code>	suppresses re-drawing to the normal plane but the window server clears the pixels.
<code>W_WIN_BACK_SET_NO_REDRAW</code>	suppresses re-drawing to the normal plane but the window server sets the pixels.
<code>W_WIN_BACK_NONE_NO_REDRAW</code>	suppresses re-drawing to the normal plane; the window server does nothing to the pixels in this plane - it neither sets nor clears them.
<code>W_WIN_BACK_GREY_CLR_NO_REDRAW</code>	suppresses re-drawing to the grey plane but the window server clears the pixels.
<code>W_WIN_BACK_GREY_SET_NO_REDRAW</code>	suppresses re-drawing to the grey plane but the window server sets the pixels.
<code>W_WIN_BACK_GREY_NONE_NO_REDRAW</code>	suppresses re-drawing to the grey plane; the window server does nothing to the pixels in this plane - it neither sets nor clears them.

Bitmap sequences

A window may be given an animated image by attaching a sequence of up to twelve⁵ bitmaps to the window by calling `wSetWinBitmap`. The sequence is modified by calling `wChangeWinBitmap` and freed by calling `wFree` (destroying the window automatically frees the bitmap sequence).

Each entry in the sequence specifies

- a source bitmap
- the position of the bitmap in the window
- the transfer mode
- the time to wait (in tenths of a second) before copying the next bitmap in the sequence to the window

You should be careful about defining too short an interval between bitmaps (especially in combination with large bitmaps) as the computational effort to maintain the sequence may leave little processor bandwidth for the application to run. Note that the window server runs at a higher priority than its clients.

If the sequence contains a single bitmap, the bitmap is not animated.

Whether animated or not, a rectangle on the window that is currently covered by a bitmap in the sequence is automatically redrawn when that part of the window is invalidated.

Any number of bitmap sequences may be attached to a window. Where there is an overlap, bitmaps from sequences attached after another sequence appear *behind* that sequence.

Bitmap sequences are not designed to be used with backed-up windows and are typically used to produce:

- a window with a changing background bitmap for other drawing (however, if there is only one bitmap in the sequence, the background does not change)
- a window that is only drawn from the bitmaps in the one or more bitmap sequences and which, like backed-up windows, does not have to be redrawn (and the window should be created with the `W_WIN_NO_REDRAW` attribute).

Unless the window is created with the `W_WIN_NO_REDRAW` attribute, the window server invalidates the target rectangle in the window after it has copied a bitmap (it also invalidates any parts of the previous rectangle that is not covered by the new bitmap). The resulting redraw event is intended to prompt the client to draw on top of the bitmap. If you do not intend to draw on top of the bitmap, you should create the window with the `W_WIN_NO_REDRAW` attribute.

As well as the changes that occur as the bitmaps in the sequence are cycled through by the window server, you can make all manner of changes to the bitmaps themselves - for example, switching bitmaps and drawing to them.

⁵The limit derives from the requirement to fit the sequence in the client-side buffer.

In version 4 of the window server, bitmap sequences are, by default, drawn to the normal (black) plane only and the grey plane will display the appropriate grey background as specified in the call to `wCreateWindow`. However, there is nothing to prevent the client from drawing to the grey plane as a result of a redraw event.

A member of a bitmap sequence can be made to appear grey by setting `WS_WIN_BITMAP_GREY` for that member in the call to `wSetWinBitmap` in which case the normal plane will display the appropriate normal background.

If bitmap sequences are required which make use of both black and grey, then sprites (discussed later) may be used instead.

Using an attached bitmap to avoid redraws

If you set up a no-redraw window with an attached bitmap sequence consisting of a single bitmap that fills the window, you can draw to the bitmap and have that bitmap copied to the window by calling `wInvalidateWin` to invalidate the window. For flicker-free operation the window background should be set to `W_WIN_BACK_NONE`.

In some cases, this may be more efficient than using backed-up windows (where everything is drawn twice).

In version 4 of the window server, as mentioned before, the bitmap can be made to appear grey by setting `WS_WIN_BITMAP_GREY` in the call to `wSetWinBitmap`.

The following program uses the technique described above and works on all versions of the window server.

```
#include <plib.h>
#include <wlib.h>

#define NWS_HANDLE 0
#define MAIN_WIN 1

GLDEF_D WSERV_SPEC wspec;
GLDEF_D UINT gcid;
GLDEF_D UINT wid;
GLDEF_D P_POINT winsize={160,80};
GLDEF_D WS_WIN_BITMAP bitseq;

GLDEF_C VOID CreateBitmap(VOID)
{
    G_GC gc;

    bitseq.bitmap=gCreateBit(0,&winsize);
    bitseq.pos.x=0;
    bitseq.pos.y=0;
    bitseq.rect.tl.x=0;
    bitseq.rect.tl.y=0;
    bitseq.rect.br=winsize;
    bitseq.mode=G_TRMODE_REPL;
    gc.style=G_STY_BOLD|G_STY_DOUBLE;
    gcid=gCreateGC(bitseq.bitmap,G_GC_MASK_STYLE,&gc);
    gClrRect(&bitseq.rect,G_TRMODE_CLR);
}

GLDEF_C VOID CreateWindow(VOID)
{
    W_WINDATA windata;

    windata.flags=W_WIN_NO_REDRAW;
    windata.extent.tl.x=0;
    windata.extent.tl.y=0;
    windata.extent.width=winsize.x;
    windata.extent.height=winsize.y;
    windata.background=W_WIN_BACK_NONE;
```

```

wid=wCreateWindow(0,W_WIN_NO_REDRAW
    |W_WIN_EXTENT
    |W_WIN_BACKGROUND,&windata,MAIN_WIN);
wSetWinBitmap(wid,1,&bitseq);
wInitialiseWindowTree(wid);
}

GLDEF_C INT main(VOID)
{
    WS_EV event;
    P_RECT box;
    TEXT bb[32];

    wConnect(&wspec,NWS_HANDLE,W_CONNECT_PRIORITY);
    CreateBitmap();
    CreateWindow();
    box=bitseq.rect;
    p_insrec(&box,1,1);
    gDrawBox(&box);
    wInvalidateWin(wid);
    p_insrec(&box,1,1);
    for (;;)
    {
        wGetEventWait(&event);
        if (event.type==WM_KEY)
        {
            p_atos(&bb[0],"Key code: %d",event.p.key.keycode);
            gPrintBoxText(&box,50,G_TEXT_ALIGN_CENTRE,0,&bb[0],p_strlen(&bb[0]));
            wInvalidateWin(wid);
            if (event.p.key.keycode==W_KEY_RETURN)
                break;
        }
    }
    return(0);
}

```

Sprites

In version 4 of the window server a window may be given an animated image by attaching a sprite to the window.

A sprite is created at a specified position within the window and the animation produced by creating a sequence of up to 13 bitmap sets attached to the sprite by calling `wCreatesprite`. The bitmap sets and the position of the sprite in the window can be changed by calling `wSetSprite`; the sprite itself can be freed by calling `wFree`.

Each bitmap set specifies:

- up to three source bitmaps for the normal plane, i.e. one for each of the three possible transfer modes (set, clear and invert)
- up to three source bitmaps for the grey plane, i.e. one for each of the three possible transfer modes (set, clear and invert)
- the position of the bitmaps relative to the sprite
- the time to wait (in tenths of a second) before displaying the next bitmap set

Unlike the pre-version 4 animated sequences, a sprite is not displayed as part of the window background. The window server takes care of saving and restoring the contents of the underlying display even if this changes during the sprite's existence. This can give the impression that the animation 'floats' above the underlying display.

When the sprite is created, setting the flag `W_SPRITE_CLIP_CHILDREN` allows child windows of the window to which the sprite is attached, to clip the sprite. If the flag is not set, the sprite can only be clipped by the edges of the window to which it is attached or by other non-related windows.

If the sequence contains a single bitmap set, the resulting display is not animated.

Only one sprite may be created for each client of the window server.

Redrawing

Windows that are not backed up by bitmaps or an attached bitmap or are not created as no-redraw windows must be *redrawn* as well as drawn.

If you intend to avoid redraws by using only backed-up windows, no-redraw windows and windows that are drawn from bitmap sequences, none of this section applies.

Redraw events

A window should be redrawn by a client when it receives a *redraw event* from the window server (after calling `wGetEventWait`, `wGetEvent` or `wGetEventSpecial`).

A redraw event indicates

- the window to be redrawn
- a rectangle within the window that needs to be redrawn

The rectangle is often ignored (especially for simple windows) and the whole window is drawn instead.

The window server keeps an *update region* for each window to record that part of a window that is *invalid*. When the update region is not empty and the event queue is otherwise empty, the window server will complete a client's call to `wGetEventWait`, `wGetEvent` or `wGetEventSpecial` with a redraw event.

Note that user input events and foreground/background change events are effectively delivered at a higher priority than redraw events. This is sometimes desirable and sometimes not.

Note also that, unlike other events, the sending of a redraw event by itself does not clear or otherwise reduce the update region (this is described below). The window server will continue to send redraw events indefinitely⁶ until the update region is reduced by the client *validating* a part or the whole of the region - normally by the client calling `wBeginRedraw` (or a variant thereof), as described below.

Update region

The update region is similar to the drawing region in that it consists of a list of rectangles that are used to clip graphics output. Whereas the drawing region is used for *drawing*, the update region is used for *redrawing*. The update region is also used to generate redraw events as described above.

The drawing region describes that part of the window that is visible. The update region describes that part of the window that is both visible and invalid.

The window server automatically adds to a window's update region when:

- the window is first created
- the window's size is increased
- a previously obscured part of the window is exposed by changes (of position, size, front to back ordering or visibility) to a window that previously obscured it
- the contents of a window is scrolled using `wScrollRect` or `wScrollWin`

A backed-up window always has an empty update region because the window server copies the data from the backup bitmap rather than add to the update region.

Invalidating rather than drawing

The client can itself add to the update region by *invalidating* a part of the window or all of the window by calling `wInvalidateRect` or `wInvalidateWin` respectively.

It is often simpler for an application (at the expense of performance) to invalidate a part of a window or the whole of the window rather than draw to it.⁷ Invalidating causes the window server subsequently to send redraw events to the client.

⁶A client that just ignores redraw events will loop indefinitely and "hog" the processor.

⁷If a window is not visible or is substantially obscured, invalidating can actually be more efficient than drawing. However, this occurs rarely in practice.

The update region of a window is automatically reduced by the window server as a window becomes obscured by other windows. However, in practice, the update region is normally reduced (partially or wholly) *before* the client draws or redraws to it.

Validating before drawing or redrawing

You always automatically validate before redrawing (by calling a variant of `wBeginRedraw`) and you would normally also validate before drawing (by calling `wValidateWin` or `wValidateRect`).

As well as reducing the update region, validating can also prepare the background, depending on the value of the background window attribute, as follows:

<code>W_WIN_BACK_CLR</code>	prior to version 4 of the window server, clear the pixels in the window (this is the default) in version 4, clear the pixels in the normal (black) plane (this is the default)
<code>W_WIN_BACK_SET</code>	prior to version 4 of the window server, set the pixels in the window in version 4, set the pixels in the normal (black) plane
<code>W_WIN_BACK_NONE</code>	do nothing; in version 4, this is specific to the normal (black) plane

In version 4 of the window server the above attributes can be OR'd with one of the following:

<code>W_WIN_BACK_GREY_CLR</code>	clear the pixels in the grey plane
<code>W_WIN_BACK_GREY_SET</code>	set the pixels in the grey plane
<code>W_WIN_BACK_GREY_NONE</code>	do nothing; specific to the grey plane

The `W_WIN_BACK_CLR` background is the easiest to deal with and is often used - particularly for simple windows or for windows with no grey. For flicker-free drawing or redrawing, use `W_WIN_BACK_NONE` (and `W_WIN_BACK_GREY_NONE` if using grey) and then program such that every pixel is covered when the drawing code executes (for example, using `gPrintBoxText` rather than `gPrintText`).

Drawing

You normally draw to a window when the data it is displaying has changed (say as a result of user input).

When drawing a particular rectangle of a (non-backed-up) window, a client should call `wValidateRect` before commencing the drawing. When drawing the whole window (which is more common), the client first validates the whole window by calling `wValidateWin`.

You must validate before drawing if you are relying on the `W_WIN_BACK_CLR`, `W_WIN_BACK_SET`, `W_WIN_BACK_GREY_CLR` or `W_WIN_BACK_GREY_SET` window attribute to prepare the background.

If you are drawing to a window with a `W_WIN_BACK_NONE` or `W_WIN_BACK_GREY_NONE` background, you need not validate before drawing to the window. If you don't validate, you will not preempt any redraw event when the same area happens to be invalid at the time and the same image may subsequently be redrawn unnecessarily (although, frankly, this is unlikely to be noticed by any user).

When drawing to a window, a client may use a permanent or a temporary graphics context. (Note that all graphics output is directed at a current graphics context.)

When using a temporary graphics context, the calls to graphics output functions would be enclosed by calls to `gCreateTempGC` and `gFreeTempGC`. If there is no requirement to change the default initial values of the graphics context, you would use `gCreateTempGC0` in place of `gCreateTempGC`.

When using a permanent graphics context (which was previously created and assigned to the window by a call to `gCreateGC` or `gCreateGC0`), you would typically use `gSetGC` or `gSetGC0` to make the appropriate permanent graphics context current before drawing.

Redrawing

While drawing is clipped to the window's drawing region, redrawing is clipped to the window's update region (for appearance and efficiency reasons).

The client informs the window server that it is redrawing rather than drawing by enclosing the graphics output function calls between calls to `wBeginRedraw` and `wEndRedraw`. As well as informing the window server that the client is about to redraw rather than draw, the call to `wBeginRedraw` also validates that rectangle of the window (specified as a parameter to `wBeginRedraw`).

As with drawing, a client that is redrawing to a window may use a permanent or a temporary graphics context.

There are no fewer than six variants of `wBeginRedraw` which vary according to whether a temporary graphics context is created (and, if so, whether it is to be altered from its default settings) and whether a part or the whole of the window is being redrawn. The full set is as follows:

<code>wBeginRedraw</code>	to redraw a part of the window using an independently created temporary or permanent graphics context
<code>wBeginRedrawWin</code>	to redraw the whole of the window using an independently created temporary or permanent graphics context
<code>wBeginRedrawGC</code>	to redraw a part of the window using a temporary graphics context that is created and initialised with specified values
<code>wBeginRedrawGC0</code>	to redraw a part of the window using a temporary graphics context that is created with default initial values
<code>wBeginRedrawWinGC</code>	to redraw the whole of the window using a temporary graphics context that is created and initialised with specified values
<code>wBeginRedrawWinGC0</code>	to redraw the whole of the window using a temporary graphics context that is created with default initial values

When a begin redraw function is used to simultaneously create a temporary graphics context, the call to `wEndRedraw` automatically frees it.

Note that if you mistakenly enclose the graphics output function calls between calls to `wBeginRedraw` and `wEndRedraw` when *drawing* (rather than redrawing in response to a redraw event), the drawing will probably not appear since it will be clipped to the update region (which is most likely to be null).

Going deaf

A client that owns one or more windows with invalid areas and which is not requesting events from the window server is said to have "gone deaf".

An application goes deaf because it is performing a task that takes an extended time to complete. Examples of such extended tasks are:

- loading or saving a large file (or some other processing of a large file)
- a stalled write to the parallel port (say, because the printer is out of paper)
- computing *PI* to a thousand decimal places
- a bug that has caused the process to "hang" in an indefinite loop

Applications that process redraw events have a responsibility to process them within a reasonable time. If this is not done, the screen may contain images drawn by some other client (which is very confusing to the user).

With a window server in a preemptive multi-tasking operating system, deaf clients need not block the user from switching to another task (as happens in non-preemptive multi-tasking window environments such as Microsoft Windows and others, where a deaf application typically leads to an unwelcome mouse icon such as an hour glass, a watch or, arguably more positively, a bee). With the window server handling the task switch key or keys, an aberrant application task that has hung does not require a system reset - the user can task to a suitable system application and terminate the task.

To make it easier to diagnose when a client has gone deaf, the window server has mechanisms to detect deaf clients. On the HC, S3, S3a and *Workabout*, task switching to a deaf client will cause the "hung up" status window to be presented. On the MC, the invalid areas are eventually covered with a grey pattern. From the user's point of view, exposing a deaf client is preferable to leaving the debris of some other client's windows which may lead the user to blame the wrong application.

The window server does not detect a client which is reading redraw events but discarding them.

Applications can avoid going deaf within potentially blocking functions such as a write to a parallel port by performing such operations asynchronously - as described in chapter *Asynchronous Requests and Semaphores* in the *PLIB Reference* manual. The same chapter also describes how to use `p_ioyield` to poll at intervals for the receipt of a redraw message (after using the asynchronous `wGetEvent` or `wGetEventSpecial`) while performing an extended process. Rather than polling, it is worth considering using `p_execc` to create a transient sub-process to perform an extended task and to use `p_logona` to be signalled when the process completes (as described in the chapter *Processes and Inter-Process Messaging* in the *PLIB Reference* manual).

Applications that are structured to present percentage done indicators or a cancel option (or preferably both) while performing an extended task are well structured to handle redraws and can easily avoid going deaf. In any case, applications that are deaf (and dumb) to the user for extended periods are generally not welcomed by them.

Redraw priority

Although redraw events always have a lower priority than user input events and background/foreground events, there is a two-level redraw priority that operates between windows.

By default, windows are created at the lower priority and certain windows may be selected to receive their redraw events before the crowd by specifying the `W_WIN_PRIORITY` bit to `wCreateWindow` or `wSetWindow`. This is very much fine tuning, though.

More about windows

Creating and initialising a window system

A window is created by calling `wCreateWindow` where you specify such things as the parent window, the position and size of the window (in the parent's coordinate system), whether the window is backed-up by bitmaps and so on.

When sibling windows are created, they are created in front of any existing siblings (this is only significant when sibling windows overlap - they often don't). You can set and sense the front-to-back position of a window in its sibling list using `wWindowPosition` and `wGetWindowPosition` respectively.

Each successful call to `wCreateWindow` returns a window ID. The ID is used subsequently to refer to that window.

After a successful return from `wCreateWindow`, the window is just a dormant data structure in the window server's data segment with no visibility on the screen. You can't draw to the window and you won't get any redraw events (or any mouse events if there is a pointing device) until the window is initialised by calling `wInitialiseWindowTree`.

As its name suggests, `wInitialiseWindowTree` initialises not just the window but all its descendants as well. In practice, a tree of windows is often created as a logical whole and it is desirable to activate the whole tree at once by calling `wInitialiseWindowTree` (passing it the window ID of the parent) after having successfully set the tree up from the parent window down by successive calls to `wCreateWindow`.

The above is not meant to suggest that all new window systems take the form of a single tree with a single parent (although this is the more common case). It is quite possible for the window system to be created to consist of a number of (say sibling) windows or window trees. In such a case, you would still create the whole system with successive calls to `wCreateWindow` and then make multiple calls to `wInitialiseWindowTree` to initialise the system.

Note that `wCreateWindow` can fail through lack of system memory but `wInitialiseWindowTree` cannot. When creating a window system you must be careful to destroy any partially created system should a call to `wCreateWindow` fail.

Destroying a window system

Just as windows are created a tree at a time, windows are, in general, destroyed a tree at a time by calling `wCloseWindowTree`.

A client that builds window systems would keep at least the window IDs of the windows in client-side data structures. In the client's data space, the data structures that contain the window IDs are unlikely to be related in the same tree structure as the windows are in the window server. Where this is the case, it may not be convenient for the client to recover the window side relationships and destroy windows a tree at a time (effectively bottom up) when destroying a window system. If a client uses its relationships (which are probably tree-like but a different tree) to destroy a window system, some windows would be destroyed more than once (because `wCloseWindowTree` has to destroy any descendants as well as the specified window).

In recognition of this problem in destroying window systems, the window server does not consider it an error if a window is destroyed more than once.

This does (quite reasonably) assume that windows systems are destroyed without any intervening window creations.

Event sources other than the window server

You need only read this section if you are implementing a program that responds to event sources other than just the window server (and therefore the program is using one of the asynchronous functions `wGetEvent` or `wGetEventSpecial` rather than `wGetEventWait`).

This section assumes familiarity with asynchronous requests - as described in the chapter *Asynchronous Requests and Semaphores* in the *PLIB Reference* manual.

When responding to window server events that are requested asynchronously using `wGetEvent` or `wGetEventSpecial`, you should completely process a window server event (after returning from `p_iowait` and having established that `wGetEvent` or `wGetEventSpecial` has completed) before calling either `wGetEvent` or `wGetEventSpecial` again to request the next event.

Bearing in mind the multi-tasking nature of the system and the fact that the window server runs at a higher priority than its clients, it is quite possible for the request made by a call to `wGetEvent` or `wGetEventSpecial` to complete while responding to an event from a source other than the window server (for example, the expiry of a timer or the receipt of some data from the serial port).

In particular, if one or more windows are directly destroyed in response to a non-window server event there is the possibility that the next window server event (a redraw event say) will relate to a window that has already been destroyed.

To guard against this possibility, you should not destroy a window or a window system directly in response to a non-window server event but use `wCancelGetEvent` to instruct the window server to send the caller a `WM_CANCELLED` event and then respond to the `WM_CANCELLED` event as you would otherwise have responded to the non-window server event.

After a call to `wCancelGetEvent`, the window server delivers the `WM_CANCELLED` event at the highest priority - any other events waiting in the window server client event queue are overtaken. The only window server event `WM_CANCELLED` can't overtake is one that has already been delivered.

Visibility of windows

A window is invisible when:

- it has been created (using `wCreateWindow`) but not yet initialised (using `wInitialiseWindowTree`)
- it has been made invisible by a call to `wMakeInvisible`;
- the window was created with the `W_WIN_FOREGROUND_ONLY` attribute set and belongs to a client that is currently a background client (this case only applies to large screen versions of the window server)

Note that `wInitialiseTree`, `wMakeInvisible` and `W_WIN_FOREGROUND_ONLY` all propagate their effect to descendant windows.

After being initialised, a window is marked as visible. Once a window tree has been made invisible using `wMakeInvisible`, it may be made visible again by calling `wMakeVisible`.

A call to `wMakeVisible` or `wInitialiseWindowTree` does not necessarily make all descendants visible since `wMakeInvisible` may have been applied to a descendant.

The window server treats windows that are invisible as follows:

- if the window is backed-up, any drawing to it is drawn only to the backup bitmap(s)
- invalidating a window using `wInvalidateRect` or `wInvalidateWin` has no effect
- windows behave as if they do not exist with respect to mouse input (this only applies when there is a pointing device)

Scrolling

The contents of a window may be scrolled by a horizontal or vertical direction or a combination of the two using `wScrollWin` or `wScrollRect`.

Both these functions are better understood in terms of rectangle copying rather than scrolling where:

<code>wScrollRect</code>	copies a source rectangle in a window to a rectangle of the same size in the same window but displaced.
<code>wScrollWin</code>	works just like <code>wScrollRect</code> except that the source rectangle is defined by the boundaries of the window offset, in the opposite direction, by the amount of the scroll. This is what is normally used to scroll the contents of a window.

Both functions copy only those parts of the source rectangle that are both visible and valid. This means that the copy excludes the following from the source rectangle:

- those parts that are in the update region
- those parts that are obscured or clipped by other windows
- those parts that are beyond the boundaries of the window

Although these parts are not copied, their existence causes the corresponding region of the target rectangle to be invalidated.

If the window is a backed-up window, the copy is also applied to the backup bitmap(s). For a backed-up window:

- the update region is always empty
- those parts that are obscured or clipped can be recovered from the backup bitmap(s)
- those parts that are beyond the boundaries of the window are cleared.

Since these functions access the same window server operations that are applied when a window is moved using `wSetWindow`, the above comments about not copying invalid regions applies to window moving too. If you wish to simultaneously move and change the appearance of a window you should use `wInvalidateWin` to invalidate those parts that are to change before using `wSetWindow` to move the window.

Continuous scrolling

The function `wScrollWin` is often used to scroll the contents of a window in response to user input (such as down arrow key presses, for example). Calling this function necessarily introduces an invalid area at the upwind border (or borders) of the direction of scroll.

If the window is a backed-up window, these areas are automatically cleared by the window server by the `wScrollWin` operation and would subsequently be drawn by the client as part of the scroll processing.

If the window is not backed-up by a bitmap, the window server invalidates those areas brought in from outside the window and there is the temptation to leave it to the redraw event handling to draw these areas.

However, this is not good practice because redraw events are delivered only when there is no user input and in the quite typical case where the user holds down the key that is causing the scroll, the redraws are not processed until the user releases the key. This means that the screen rapidly fills with a copy of the image that was at the upwind border of the scroll and the window is only redrawn with something sensible when the key is released.

When using non-backed-up windows you should program as for backed-up windows and calculate the area that needs to be drawn after the scroll and then validate and draw it.

Text cursor

A client can nominate at most one window at a time to contain a text cursor (which is optionally flashing) by calling `wTextCursor`. The text cursor can subsequently be moved from one window to another by calling `wTextCursor` again. To remove the text cursor from the window that contains it, you call `wEraseTextCursor`.

The cursor is specified as a rectangle which is, in principle, XORed with whatever is underneath it. Applications typically define a text cursor as a vertical line in between characters, a horizontal line underneath characters or a block cursor that fits over characters.

In version 4 of the window server, the cursor can be made to appear grey on those machines that support grey such as the S3a and *Workabout*.

The window server handles the flashing of the cursor and ensures that it does not interfere with any drawing or redrawing in its vicinity.

On large screen versions of the window server where the windows of more than one client are on the screen at a time, the window server automatically ensures that only the foreground text cursor is visible.

Bitmaps

Drawing to the screen from a bitmap

A bitmap is a piece of off-screen memory that is organised in the same way as the screen bitmap. A bitmap can rapidly be copied to a window for display using one of:

<code>gCopyBit</code>	to copy a rectangle from a bitmap to a given position in the current graphics context
<code>gDrawBit</code>	to copy a rectangle from an open bitmap file to a given position in the current graphics context. This is available in version 4 of the window server.
<code>gFillPattern</code>	to fill a rectangle in the current graphics context with repeated copies of a bitmap
<code>wSetWinBitmap</code>	to copy bitmaps to a window at specified time intervals from a sequence of bitmaps (as described earlier)

All the above functions can copy a bitmap in one of four transfer modes:

<code>G_TRMODE_REPL</code>	where bits in the source pattern replace corresponding bits in the destination.
<code>G_TRMODE_SET</code>	where 1s in the source pattern set corresponding bits in the destination (0s in the source do not change corresponding bits in the destination). This would normally be used to copy a bitmap on to a previously cleared destination.
<code>G_TRMODE_CLR</code>	where 1s in the source pattern clear corresponding bits in the destination (0s in the source pattern do not change corresponding bits in the destination). This would normally be used to copy a bitmap on to a previously set bitmap.
<code>G_TRMODE_INV</code>	where 1s in the source pattern toggle corresponding bits in the destination (0s in the source pattern do not change corresponding bits in the destination). This is suitable for copying over an existing pattern and may be reversed by a second identical application.

Creation and storage of bitmaps

A bitmap is created uninitialised by calling `gCreateBit` or (more likely) it is loaded from a file that contains one or more bitmaps using `gOpenBit` or `gGetBit`. A bitmap is freed using `wFree`.

The function `gSetOpenAddress` may be used immediately before `gOpenBit` or `gGetBit` to load the bitmap from anywhere within the file (typically used to load a bitmap which has been embedded into the program file).

In version 4 of the window server, `gInitBit` is used to open a multiple bitmap file ready for calls to `gGetBit` or `gDrawBit`.

When a bitmap is successfully created, `gCreateBit`, `gOpenBit` and `gGetBit` return a bitmap ID (which is subsequently used to reference the bitmap).

The window server keeps a built-in ROM-based grey bitmap (with a chequerboard pattern). This may be accessed with the bitmap ID `WS_BITMAP_GREY`. The bitmap has size `WS_BITMAP_GREY_SIZE_X` by `WS_BITMAP_GREY_SIZE_Y`.

Note that version 4 of the window server supports grey for those machines such as the Series 3a that can display true grey.

When a bitmap is created, it may be stored in the window server's data space or in a named memory segment (named memory segments are described in the *Memory Allocation* chapter of the *PLIB Reference* manual).

The window server automatically places bitmaps that are larger than 2K in named memory segments. You can request that a bitmap be stored in a named segment rather than the window server's data segment (regardless of the size).

A bitmap that is in a named memory segment can be accessed directly by the client using `p_sgcopyfr` and `p_sgcopyto` (described in the *Memory Allocation* chapter of the *PLIB Reference* manual) or otherwise.

Drawing to bitmaps

Like windows, bitmaps can be drawn to using any of the window server graphics output functions.

However, unlike windows there is no drawing region or update region (so there is no such thing as validating before drawing). Drawing is clipped only to the limits of the bitmap.

Bitmaps that are loaded from a file are typically read-only (which also makes them shareable). If you are going to draw to a loaded bitmap, you should specify the `WS_BIT_WRITE` attribute when you call either `gOpenBit` or `gGetBit` to load the bitmap.

Graphics output is directed at a current graphics context (which may be assigned to a window or a bitmap). As with drawing to a window, a program may use a permanent or a temporary graphics context.

When using a temporary graphics context, the calls to graphics output functions would be enclosed by calls to `gCreateTempGC` and `gFreeTempGC`. If there is no requirement to change the default initial values of the graphics context, you would use `gCreateTempGC0` in place of `gCreateTempGC`.

When using a permanent graphics context (which was previously created and assigned to the bitmap by a call to `gCreateGC` or `gCreateGC0`), you would typically use `gSetGC` or `gSetGC0` to make the appropriate permanent graphics context current before drawing.

Bitmap files

Bitmap files (which normally have the file name extension `.pic`) may be created in one of the following ways:

- by saving the contents of a bitmap, a screen or a backed-up window using `gSaveBit` (which saves the whole bitmap) or `gSaveRect` (which saves a rectangle of the bitmap). In version 4 of the window server, if the screen or a backed up window uses grey, saving either of them will create a double bitmap.
- by saving the contents of a bitmap, a screen or a backed-up window using `gSaveMultiBit` (which saves the whole bitmap) or `gSaveMultiRect` (which saves a rectangle of the bitmap). If the screen or a backed up window uses grey, saving either of them will create a double bitmap. Available in version 4 only.
- by converting a PCX file using the `wspcx` program (which runs on a PC). Many PC-based graphics applications are able to produce PCX files.
- by saving the whole screen to a file by pressing `SHIFT-CTRL-PSION-S`. This is not possible on machines without a `CTRL` key (such as the HC).

Using wspcx

The `wspcx.exe` program (which is placed in the `\sibosdk\sys` directory by the installation) may be used to convert PCX files to window server bitmap files and vice versa and also to link a number of `.pic` files into one `.pic` file.

Where the `.pcx` file contains more than two colours, the following 'rules' apply:

- *white* is converted to *white*
- *black* is converted to *black*
- *all other colours* are converted to *grey*

On conversion:

- If a `.pcx` file is marked as being black and white only, then the `.pic` file will contain a single bitmap.
- If a `.pcx` file is marked as being in colour, then the `.pic` file will contain a double bitmap, where the first bitmap represents the normal plane and the second represents the grey plane.
- If a `.pcx` file is marked as being in colour but only contains an image using the black and white "colours", then the `.pic` file will still contain a double bitmap.

To convert a *.pcx* file to a window server *.pic* bitmap file, use:

```
wspcx -p <input> [-i] [-o<output>] [-s] [-x<width>] [-y<height>]
```

where the *-p* indicates PCX to PIC conversion and *<input>* is the name of file to be converted (which is assumed to have a *.pcx* extension unless otherwise specified). The remaining optional parameters are:

<i>-i</i>	Invert the bitmap while converting.
<i>-o<output></i>	Specifies the output file name and directory (otherwise it is the same as the input file name with a <i>.pic</i> extension).
<i>-x<width></i> <i>-y<height></i>	Clip or expand the bitmap to the specified size (in pixels). If expanded, the bitmap is padded out with blank space.
<i>-s</i>	Suppresses output messages.

For example:

```
wspcx -p -i sausage.pcx
```

produces the inverted *sausage.pic*.

To convert from a *.pic* file to a *.pcx* file, you use:

```
wspcx -w <input> [-i] [-o<output>] [-s]
```

To just invert the bits in a *.pic* file without any other conversion, you use:

```
wspcx -i <input> [-o<output>] [-s]
```

To link a number of *.pic* files into one output *.pic* file, you use:

```
wspcx -l <input> [-o<output>] [-s]
```

where *<input>* is a text file (with extension *.plk*) that lists the *.pic* files to be linked to produce a file with the same name as the *.plk* file but with the *.pic* extension. A C header file (with extension *.ph*) is also generated that contains *#defines* for the index number and dimensions of each component bitmap.

Capturing the screen to a bitmap file

Pressing SHIFT-CTRL-PSION-S on an MC, S3, S3a or a *Workabout* saves the current screen to a file called *screen.pic* in the current path of the window server. Any existing file of the same name is replaced.

In practice, the current path of the window server on a SIBO machine is always *LOC::M:* (it is defined when the window server process is started - well before you have any chance of influencing it).

However, if an environment variable with the name *\$WS_SD* exists, the window server uses its value to open the file to be created. For example, running the following program:

```
#include <p_std.h>

GLDEF_C INT main(VOID)
{
    p_setenv("$WS_SD", "B:\\SCREEN.PIC");
    return(0);
}
```

subsequently causes the screen dump to be written to the root directory of the local *B:* drive.

If the save fails for any reason (such as disk full), the file is not produced and no notification of the failure is given.

You can use this behaviour to disable the SHIFT-CTRL-PSION-S screen dump key by setting up *\$WS_SD* to contain an illegal file specification. For example, just inserting the following line of code:

```
p_setenv("$WS_SD", "");
```

disables the screen dump key.

Screen capture program for the HC

The following program illustrates how you can construct your own screen capture program on an HC, or an S3. The program will work on an S3a or *Workabout* provided that the screen does not contain grey. To capture grey, the program code needs to be changed in order to capture the grey plane as well as the normal plane (see `gPeekBit` in the *Graphics Output* chapter and any reference manual on PCX file formats).

The program has a "quick and dirty" user interface constructed from the simple console functions `p_printf`, `p_getch` and `p_getl` (described in the *PLIB Reference* manual). The first call to `p_printf` automatically connects to the window server and must precede the call to `wCaptureKey`.

To save a screen (by default to *rem::screen.pic*), you task to the application and press `PSION+S`.

```
/*
SCAPT.C - Capture the screen to a file
*/

#include <plib.h>
#include <wlib.h>

GLDEF_C VOID main(VOID)
{
    INT ret;
    TEXT name[64];

    p_scpy(&name[0], "rem::screen.pic");
    p_printf("\f"); /* connect to window server */
    wCaptureKey(W_SPECIAL_KEY|'s', 0, 0);
    for (;;)
    {
        p_printf("\fCapture file is\r\n%s", &name[0]);
        p_printf("\nE to Exit\r\nN to set file Name\r\nPsion+S to capture");
        switch (p_getch())
        {
            case 'e':
            case 'E':
                p_exit(0);
            case 'n':
            case 'N':
                p_getl("Name:", &name[0], 64);
                break;
            case W_SPECIAL_KEY|'s':
                ret=gSaveBit(&name[0], 0);
                if (ret)
                    p_notifyerr(ret, "Screen save failed", 0, 0, 0);
                break;
        }
    }
}
```

Capturing the screen directly to a PCX file

It isn't that difficult to generate a PCX file directly from the screen or any other bitmap. The module *pcxsave.c* (supplied in *lsdkdoc\demo*) contains the code which supports the function `pcxScreenSave` that saves the entire screen in PCX format of a given name.

The source of *pcxsave.c* is as follows:

```
/*
Save the screen to PCX file
*/

#include <plib.h>
#include <wlib.h>

#define BUFLen 256

GLREF_D WSERV_SPEC *wserv_channel;
```

```
LOCAL_D VOID *fcb;
LOCAL_D UBYTE *pbuf;
LOCAL_D UBYTE *pobuf;
LOCAL_D UBYTE obuf[BUFLen];

LOCAL_C VOID FlushBuffer(VOID)
{
    f_write(fcb,&obuf[0],pobuf-&obuf[0]);
    pobuf=&obuf[0];
}

LOCAL_C VOID putb(INT b)
{
    *pobuf++=b;
    if (pobuf==&obuf[BUFLen])
        FlushBuffer();
}

LOCAL_C INT rev(INT dat)
{
    INT i;
    INT rdat;

    rdat=0;
    for (i=0;i<8;i++)
        rdat|=((dat>>i)&1)<<(7-i);
    return(rdat^0xff);
}

LOCAL_C VOID WritePCXLine(UBYTE *buf,UINT len)
{
    UBYTE *p;
    UINT end;
    UINT count;
    INT byte;

    p=buf;
    byte=*p++;
    count=1;
    do
    {
        end=(p==(&buf[0]+len));
        if (byte==*p && count<0x3f && !end)
        {
            count++;
            p++;
        }
        else
        {
            byte=rev(byte);
            if (count>1 || (byte&0xC0)==0xC0)
                putb(count+0xC0);
            putb(byte);
            byte=*p++;
            count=1;
        }
    } while (!end);
}
```

```

LOCAL_C VOID WriteHeader(TEXT *name,UINT width,UINT height,UINT bytewid)
{
    struct
    {
        UBYTE manuf;
        UBYTE hard;
        UBYTE encod;
        UBYTE bitpx;
        P_RECT rect;
        WORD hres;
        WORD vres;
        UBYTE clrma[48];
        UBYTE vmode;
        UBYTE nplanes;
        WORD bpln;
        UBYTE padding[60];
    } header;

    f_open(&fcb,name,P_FREPLACE|P_FSTREAM|P_FUPDATE);
    p_bfil(&header,sizeof(header),0);
    header.manuf=10;
    header.hard=3;
    header.encod=TRUE;
    header.bitpx=1;
    header.rect.br.x=width-1;
    header.rect.br.y=height-1;
    header.hres=640;
    header.vres=480;
    header.nplanes=1;
    header.bpln=bytewid;
    f_write(fcb,&header,sizeof(header));
}

#pragma save, ENTER_CALL

LOCAL_C INT WritePCXFile(TEXT *name)
{
    UINT len;
    P_POINT size;
    P_POINT line;

    size=wserv_channel->conn.info.pixels;
    len=((size.x+15)>>3)&~1;
    WriteHeader(name,size.x,size.y,len);
    pbuf=f_alloc(len);
    line.x=0;
    for (line.y=0;line.y<size.y;line.y++)
    {
        gPeekBit(0,&line,size.x,pbuf);
        WritePCXLine(pbuf,len);
    }
    FlushBuffer();
    return(0);
}

#pragma restore

GLDEF_C INT pcxScreenSave(TEXT *name)
{
    INT ret;

    fcb=NULL;
    pbuf=NULL;
    pobuf=&obuf[0];
    ret=p_enter2((VOID *)WritePCXFile,name);
    p_free(pbuf);
    p_close(fcb);
    return(ret);
}

```

The function `WritePCXFile` writes the PCX header and then loops extracting lines from the screen using `gPeekBit` and converting each line to PCX format. Bytes are written to the output file as they are converted (by `writePCXLine`). The above code can easily be converted to extract from arbitrary areas of the screen or from any bitmap.

Note the use of `p_enter` to implement structured error handling.

Some of the screen shots used in this manual were captured using *scapt.c* (also supplied in `\sibosdk\demo`) which uses the function `pcxScreenSave` in *pcxsave.c*. The source of *scapt.c* is:

```
/*
SCAPT.C - Capture the screen to a file
*/

#include <plib.h>
#include <wlib.h>

GLREF_D TEXT *DatCommandPtr;

GLREF_C INT pcxScreenSave(TEXT *name);

LOCAL_D WSERV_SPEC wSpec;

GLDEF_C INT main(VOID)
{
    INT ret;
    TEXT *pc;
    TEXT name[P_FNAME_SIZE];

    pc=p_skipch(DatCommandPtr)+1;
    if (*pc)
        pc=p_skipwh(pc+1);
    ret=p_fparse(pc, "REM::SCREEN.PCX",&name[0], NULL);
    if (!ret)
    {
        wConnect(&wSpec, 0, W_CONNECT_AT_BACK);
        ret=pcxScreenSave(&name[0]);
        p_sound(1, 512);
    }
    return(ret);
}
```

To build *scapt.img* from *scapt.pr*, *scapt.c* and *pcxsave.c*, just enter:

```
TSC/M SCAPT
```

You might consider using TSCX rather than TSC. See the *Installation* chapter of the *General Programming Manual* for more information.

The program *scapt.img* is designed to be run on the target from *MCLINK* on the PC. In preparation, copy *scapt.img* to the root directory of the default drive on the target (an S3 say). To capture the screen to say *fred.pcx* in the current directory of your PC, start *MCLINK* and enter:

```
RUN SCAPT FRED
```

The target machine beeps faintly (from the call to `p_sound`) when the screen has been saved. If you omit the *FRED*, you get *screen.pcx* (from the related file name in the call to `p_fparse`).

The physical structure of bitmap files and bitmaps

Bitmap files start with a `PIC_HEAD` struct, defined in *wlib.h* as:

```
typedef struct
{
    P_FSIG sig;
    UWORD count;
    WS_PIC_HEADER wspec;
} PIC_HEAD;
```

The first member of this struct is a P_FSIG header:

```
typedef struct
{
    TEXT app_id[3]; /* application ID */
    UBYTE chk_sum; /* application ID checksum */
    UBYTE file_vn; /* file version number */
    UBYTE app_vn; /* application version number */
} P_FSIG;
```

where the P_FSIG struct is defined in *p_file.h*. For a bitmap file, the appropriate values for the P_FSIG header are:

```
P_FSIG sig = {"PIC", 'P'+'I'+'C', 0x30, 0x30};
```

The P_FSIG header is followed by a WORD count of the number of bitmaps in the file. This is then followed by an array of that many WS_PIC_HEADER structs. A WS_PIC_HEADER struct is defined in *wlib.h* as follows:

```
typedef struct
{
    UWORD checksum;
    P_POINT size;
    UWORD byte_size;
    ULONG offset;
} WS_PIC_HEADER;
```

The members of WS_PIC_HEADER are as follows:

checksum	is calculated by applying the <i>p_crc</i> function (described in the <i>PLIB Reference manual</i>) to the bitmap that is referenced by the WS_PIC_HEADER struct (excluding all headers).
size	the pixel dimensions of the bitmap (<i>size.x</i> by <i>size.y</i>)
byte_size	the byte size of the bitmap.
offset	the relative offset from the end of this header to the start of the bitmap.

The bitmap consists of *size.y* scan lines from top to bottom. Each scan line consists of an array of $((\text{size.x}+15)/16)$ words describing the pixels in the scan line from left to right. The leftmost pixel in a scan line corresponds to the least significant bit of the first word.

A named memory segment which contains a bitmap (created, for example, using *gopenBit* or *gGetBit*) contains just the bitmap, without the WS_PIC_HEADER header.

One example of the use of the physical bitmap structures described above is to animate the screen from a previously generated sequence of equally sized bitmaps from a bitmap file. After creating the bitmap memory segment using *gopenBit*, *gCreateBit* or *gGetBit*, the steps in the animation sequence are:

- use *p_read* to read the bitmap from the file into a buffer
- use *p_sgcopyto* to copy the data to the bitmap segment
- use *gCopyBit* to draw the bitmap to the screen

Since the bitmaps are stored sequentially there is no need to position the file between each *p_read* - you only have to position each time you return to the first bitmap in the sequence.

Embedded bitmap files

A bitmap file may be built into a program file by including its name in an add-file list. This process is more fully described in the *Building an Application* chapter of the *General Programming manual*.

Text fonts

A text font is a bitmap that contains up to 256 bit-images called character graphics. The character graphics in the font are indexed by a character code in the range 0 to 255.

Fonts are primarily used to implement the SIBO character set in different typefaces and sizes.

A font may also be used to implement any collection of bit-images that have the same height (as an alternative to using independent bitmaps).

A text font may not contain character graphics for the whole 256 code range and within the code range supported there may also be "holes" for which there is no character graphic.⁸

Although all the character graphics in a font are of the same height, their widths may in general vary. When all the characters with codes greater than 31 have the same width, the font is said to be *monospaced* (otherwise it is said to be *proportional*).

The SIBO character set is compatible with the IBM code page 850 character set for character codes in the range 32 to 255. In some proportional fonts, the code page 850 block graphics characters (for example, the box drawing characters) are absent. The characters with codes less than 32 are not compatible with any standard and vary from font to font.

Fast fonts

Fast fonts are stored in an expanded form that uses more memory but can be drawn faster.

All characters in a fast font must be less than or equal to 8 pixels wide.

The window server automatically recognises the difference between normal and fast fonts. Window servers before version 3.5 do not recognise fast fonts and will refuse to load them.

ROM-based fonts

An application can access the ROM-based fonts by font IDs that are known at compile time. The ROM-based font IDs start at `WS_FONT_BASE` and you can use `WS_FONT_BASE+1` etc for as many fonts as are built into the ROM.

The default font of a newly created graphics context, sometimes called the system font, is in most cases the first font in the ROM - with ID `WS_FONT_BASE`.

Whenever it is expecting a font ID, the window server converts the constant `WS_FONT_SYSTEM` to the system font (`WS_FONT_SYSTEM` is outside the range of possible font IDs). You can also obtain the system font ID directly from the `system_font` member of the `W_SERVER_INFO` struct (as described under `wConnect` in the next chapter).

On the HC and the S3, the system font is determined by the `$WS_SF` environment variable which should contain a WORD binary value of 0 for `WS_FONT_BASE` and 1 for `WS_FONT_BASE+1` and so on. If you change the value of `$WS_SF`, you must reset the machine by pressing the recessed reset button to effect the change.

In version 4 of the window server which runs on the S3a, the system fonts are determined by the `$WS_FNTS` environment variable. This contains a series of words each of which contains the fonts used by the window server in various situations (listed in the changes section earlier). The full list is repeated below and is given in the correct order.

- System font
- Notifier/Alert font
- Status Window font
- Symbols font used for the status window diamond symbol
- Medium 2 digital clock font
- Medium 2 date font
- Notifier/alert button font
- Small status window clock font

⁸If a client passes a character code for which there is no graphic, the graphic with the highest code is selected.

On the MC, the system font is determined in the same way except that two environment variables are used - \$WS_SF2 and \$WS_SF4. If the screen has fewer than 300 lines (as on the MC200), \$WS_SF2 is used. Otherwise (as on the MC400), \$WS_SF4 is used.

The following program illustrates how the environment variable may be changed.

```
#include <p_std.h>

GLDEF_C INT main(VOID)
{
    WORD flags;

    flags=1; /* choose WS_FONT_BASE+1 */
    return(p_setenvron("$WS_SF", 6, &flags, 2));
}
```

Changing the system font may upset existing applications.

In version 4 of the window server used on the Series 3a, fonts can be collected into what are called 'font groups'. A more detailed discussion of this concept can be found in the description of gConfigureFonts in the *Graphics Output* chapter of this manual.

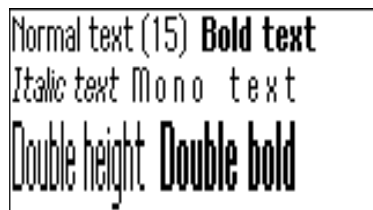
Briefly, font groups are a collection of fonts with a single identity. Essentially, each font within the group will have been specially designed with a style or a combination of styles in mind. Where a font group is to be used to print text, the window server will select the best font from within this group according to criteria based on the style or combination of styles selected (i.e. bold, italics etc). Having selected a font from within the group, it may, if necessary, algorithmically apply further styles.

The font groupings for the Series 3a and Workabout machines are summarised in the header file *fonts.h*, which also supplies a range of defined constants that can be used to identify the various ROM-based fonts.

HC fonts

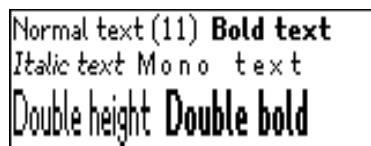
On a standard HC, there are six ROM-based fonts:

WS_FONT_BASE - large proportional (the system font). Also the system font on the MC400.



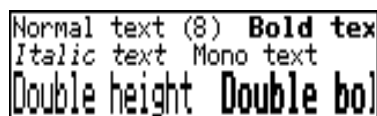
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
1	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
2	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
8	Ç	ü	é	â	ä	à	ç	ê	ë	è	ï	î	ì	Ä	Å	
9	É	æ	Æ	ô	ö	ò	û	ü	ÿ	Ö	Ü	ø	£	¤	☐	f
A	á	í	ó	ú	ñ	Ñ	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
B	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
C	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
D	ø	ð	ê	ë	è	☐	í	î	ï	☐	☐	☐	☐	☐	☐	☐
E	ó	β	ô	ò	õ	☐	p	þ	ú	û	ü	ý	☐	☐	☐	☐
F	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐

WS_FONT_BASE+1 - small proportional. Also the system font on the MC200.

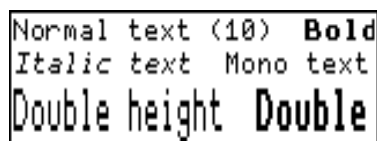


	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	☐	=	>	↑	↩	=	Ⓢ	🔄		✋	🖱️	🔍	☎️	🔔	▲	▼
1	▶	◀	🚰	✓	☐	☑	◆	♠	↑	↓	→	+	⇕	➡	↶	
2		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	¥	☐	ƒ
A	á	í	ó	ú	ñ	Ñ	☐	☐	¿	☐	☐	☐	☐	¡	«	»
B	☐	☐	☐	☐	☐	Á	Â	Ã	©	☐	☐	☐	☐	¢	☐	☐
C	☐	☐	☐	☐	☐	☐	ä	ã	☐	☐	☐	☐	☐	☐	☐	☐
D	ð	Ð	Ê	Ë	È	☐	Í	Î	☐	☐	☐	☐	☐	☐	Ì	☐
E	Ó	β	Ô	Ò	Õ	☐	Þ	þ	☐	Ú	Ù	Û	Ý	Ý	☐	☐
F	☐	☐	☐	☐	☐	☐	☐	°	☐	☐	☐	☐	☐	☐	☐	☐

WS_FONT_BASE+2 - fast monospaced (the console font). Only used on the HC.

[illegible]

WS_FONT_BASE+3 - wide monospaced. The monospaced font on the MC.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
1	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
2		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	[	]	^	
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	
6	`	a	b	c	d	e	f	g	h	i	j	k	{	}	~	
7	ç	ü	é	à	ä	å	â	ç	ê	ë	è	ï	í	î	ï	
8	ÿ	z	æ	ô	ö	õ	ñ	ú	ý	ö	ü	ø	í	î	ï	
9	É	í	ó	ú	ñ	ñ	ñ	ú	ý	ö	ü	ø	í	î	ï	
A	á	í	ó	ú	ñ	ñ	ñ	ú	ý	ö	ü	ø	í	î	ï	
B	á	í	ó	ú	ñ	ñ	ñ	ú	ý	ö	ü	ø	í	î	ï	
C	á	í	ó	ú	ñ	ñ	ñ	ú	ý	ö	ü	ø	í	î	ï	
D	á	í	ó	ú	ñ	ñ	ñ	ú	ý	ö	ü	ø	í	î	ï	
E	á	í	ó	ú	ñ	ñ	ñ	ú	ý	ö	ü	ø	í	î	ï	
F	á	í	ó	ú	ñ	ñ	ñ	ú	ý	ö	ü	ø	í	î	ï	

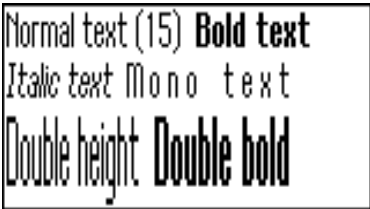
The figure on the first line of each sample display gives the height of the font in pixels.

On the S3, there is no monospace font as such. The proportional fonts are designed such that a reasonable monospace font is obtained by using the monospace style attribute `G_STY_MONO`.

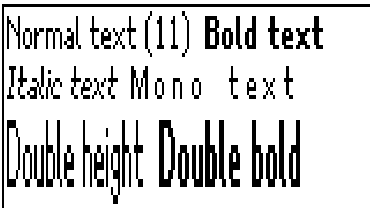
See the last two fonts in the list of HC fonts above for the full set of characters.

MC fonts

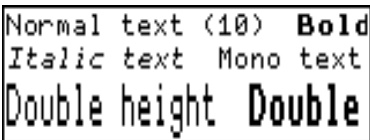
On a standard MC, there are three ROM-based fonts:



WS_FONT_BASE - proportional (MC400 system font)



WS_FONT_BASE+1 - small proportional (MC200 system font)



WS_FONT_BASE+2 - monospaced

The figure on the first line of each sample display gives the height of the font in pixels.

See the first, second and fourth fonts in the list of HC fonts above for the full set of characters.

S3a and Workabout fonts

On the S3a and *Workabout*, thirteen ROM-based fonts are defined; the first three are Series 3 fonts, examples of which are shown earlier. Access to these fonts is summarised in *fonts.h*.

The following figures show sample displays of the fonts with font heights given in pixels. Accompanying each of these displays is the corresponding set of character codes.

Note that the sample displays are 200 pixels wide.

WS_FONT_BASE+2 - numeric font



	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

WS_FONT_BASE+3 - monospaced 8 x 8

Normal text (8) **Bold text**
Italic text Mono text
 Double height Double bold

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	↓	↑	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	↓	↑	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2	↓	↑	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
3	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	q	r	s	t	u	v	w	x	y	z	{		}	~	
7	ç	ü	é	â	ä	å	ä	ü	ç	ë	ö	ï	ï	ï	ï	ï
8	É	æ	Æ	ô	ö	ö	ü	ü	ÿ	ö	ü	ø	ø	ø	ø	ø
9	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
A	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
B	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
C	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
D	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
E	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
F	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á

WS_FONT_BASE+4 - Roman 8 point

Normal text (8) **Bold text**
Italic text Mono text
 Double height Double bold

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	↓	↑	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	↓	↑	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2	↓	↑	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
3	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	q	r	s	t	u	v	w	x	y	z	{		}	~	
7	ç	ü	é	â	ä	å	ä	ü	ç	ë	ö	ï	ï	ï	ï	ï
8	É	æ	Æ	ô	ö	ö	ü	ü	ÿ	ö	ü	ø	ø	ø	ø	ø
9	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
A	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
B	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
C	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
D	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
E	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
F	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á

WS_FONT_BASE+5 - Roman 11 point

Normal text (11) **Bold text**
Italic text Mono text
 Double height Double bold

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	↓	↑	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	↓	↑	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2	↓	↑	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
3	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	q	r	s	t	u	v	w	x	y	z	{		}	~	
7	ç	ü	é	â	ä	å	ä	ü	ç	ë	ö	ï	ï	ï	ï	ï
8	É	æ	Æ	ô	ö	ö	ü	ü	ÿ	ö	ü	ø	ø	ø	ø	ø
9	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
A	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
B	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
C	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
D	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
E	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á
F	Á	á	í	ó	ñ	ñ	á	á	á	á	á	á	á	á	á	á

WS_FONT_BASE+6 - Roman 13 point

Normal text (13) **Bold text**
Italic text Mono text
 Double height Double bold

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	←	...	≡			⌘	-	.	+	↓						
1																
2	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	Δ
8	Ç	ü	é	â	ä	à	â	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	Ø	×	ƒ
A	Á	í	ó	ú	ñ	Ñ	ª	º	¿	®	¬	½	¼	¡	«	»
B	■	■	■		†	Á	Â	À	©	¶		¶	¶	¶	¥	γ
C	ℒ	⊥	⊥	⊥	+	ã	Ã	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	α
D	ð	Ð	Ê	Ë	È	Ì	Í	Î	Ï	ℒ	ℒ	■	■	■	■	■
E	Ó	ß	ô	ò	õ	ö	μ	þ	Þ	Ú	Û	Ü	Ý	Ý	˘	˙
F	-	±	=	%	¶	§	÷	.	°	˘	.	1	3	2	.	.

WS_FONT_BASE+7 - Roman 16 point

Normal text (16) **Bold text**
Italic text Mono text
 Double height Double bold

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	←	...	≡			⌘	-	.	+	↓						
1																
2	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	Δ
8	Ç	ü	é	â	ä	à	â	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ø	£	Ø	×	ƒ
A	Á	í	ó	ú	ñ	Ñ	ª	º	¿	®	¬	½	¼	¡	«	»
B	■	■	■		†	Á	Â	À	©	¶		¶	¶	¶	¥	γ
C	ℒ	⊥	⊥	⊥	+	ã	Ã	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	α
D	ð	Ð	Ê	Ë	È	Ì	Í	Î	Ï	ℒ	ℒ	■	■	■	■	■
E	Ó	ß	ô	ò	õ	ö	μ	þ	Þ	Ú	Û	Ü	Ý	Ý	˘	˙
F	-	±	=	¾	¶	§	÷	.	°	˘	.	1	3	2	.	.

WS_FONT_BASE+8 - Swiss 8 point

Normal text (8) **Bold text**
Italic text Mono text
 Double height **Double bold**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	↓	...	μ	♦	⌘	⌘	-	.	+	↓	↓	↓	↓	↓	...	...
1	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
2	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	Δ
8	Ç	ü	é	â	ä	à	â	ç	ê	ë	è	ï	î	ï	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ü	ö	Ü	ø	£	¢	¥	×	÷
A	Á	á	í	ó	ú	ñ	Ñ	º	º	¿	¼	½	¾	¼	½	¾
B	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘
C	L	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘
D	ä	ð	ê	ë	è	ï	ö	ü	ü	ü	ü	ü	ü	ü	ü	ü
E	ó	β	ô	ò	ö	ü	ü	ü	ü	ü	ü	ü	ü	ü	ü	ü
F	-	±	=	¼	½	¾	¾	¾	¾	¾	¾	¾	¾	¾	¾	¾

WS_FONT_BASE+9 - Swiss 11 point

Normal text (11) **Bold text**
Italic text Mono text
 Double height **Double bold**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	↓	...	μ	♦	⌘	⌘	-	.	+	↓	↓	↓	↓	↓	...	...
1	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
2	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	Δ
8	Ç	ü	é	â	ä	à	â	ç	ê	ë	è	ï	î	ï	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ü	ö	Ü	ø	£	¢	¥	×	÷
A	Á	á	í	ó	ú	ñ	Ñ	º	º	¿	¼	½	¾	¼	½	¾
B	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘
C	L	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘
D	ä	ð	ê	ë	è	ï	ö	ü	ü	ü	ü	ü	ü	ü	ü	ü
E	ó	β	ô	ò	ö	ü	ü	ü	ü	ü	ü	ü	ü	ü	ü	ü
F	-	±	=	¼	½	¾	¾	¾	¾	¾	¾	¾	¾	¾	¾	¾

WS_FONT_BASE+10 - Swiss 13 point

Normal text (13) **Bold text**
Italic text Mono text
 Double height **Double bold**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	↓	...	μ	♦	⌘	⌘	-	.	+	↓	↓	↓	↓	↓	...	...
1	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
2	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	Δ
8	Ç	ü	é	â	ä	à	â	ç	ê	ë	è	ï	î	ï	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ü	ö	Ü	ø	£	¢	¥	×	÷
A	Á	á	í	ó	ú	ñ	Ñ	º	º	¿	¼	½	¾	¼	½	¾
B	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘
C	L	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘	⌘
D	ä	ð	ê	ë	è	ï	ö	ü	ü	ü	ü	ü	ü	ü	ü	ü
E	ó	β	ô	ò	ö	ü	ü	ü	ü	ü	ü	ü	ü	ü	ü	ü
F	-	±	=	¼	½	¾	¾	¾	¾	¾	¾	¾	¾	¾	¾	¾

WS_FONT_BASE+11 - Swiss 16 point

Normal text (16) **Bold text**
Italic text Mono text
 Double height Double bold

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	↓	...	μ	♦	⌘	-	,	→	↓	↕	✓	+	-			
1	↑	+										▶	↑	+		
2		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	Δ
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	í	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ü	Ö	Ü	ø	£	Ø	×	ƒ	
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	®	-	½	¼	¡	«	»
B					¡	Á	Â	À	©	¶		¶	¶	¶	¥	γ
C	L	⊥	T	⊥	-	+	ä	Ã	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	α
D	ð	Ð	Ê	Ë	È	Í	Î	Ï	ℒ	ℒ	■	■	■	■	■	■
E	Ó	Ô	Õ	Ö	Ø	µ	þ	Þ	Ú	Û	Ü	Ý	ÿ			
F	-	±	=	¾	¶	§	÷	.	.	.	.	.	.	.	.	.

WS_FONT_BASE+12 - monospaced 6 x 6

Normal text (6) **Bold text**
Italic text Mono text
 Double height Double bold

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	↓	...	μ	♦	⌘	-	,	→	↓	↕	✓	+	-			
1	↑	+										▶	↑	+		
2		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	Δ
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	í	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ü	Ö	Ü	ø	£	Ø	×	ƒ	
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	®	-	½	¼	¡	«	»
B					¡	Á	Â	À	©	¶		¶	¶	¶	¥	γ
C	L	⊥	T	⊥	-	+	ä	Ã	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	α
D	ð	Ð	Ê	Ë	È	Í	Î	Ï	ℒ	ℒ	■	■	■	■	■	■
E	Ó	Ô	Õ	Ö	Ø	µ	þ	Þ	Ú	Û	Ü	Ý	ÿ			
F	-	±	=	¾	¶	§	÷	.	.	.	.	.	.	.	.	.

Font files

An application can load a font from a font file by calling `gopenFont` (which returns a font ID if it is successful).

The function `gSetOpenAddress` may be used immediately before `gopenFont` to load the font from anywhere within the file (typically used to load a font which has been embedded into the program file).

If another process of the same or a different application loads the same font, the font is shared and not reloaded.

Font files normally have the extension `.FON`.

In version 4 of the window server, an application can load an individual font from a multiple font file by calling `gopenFontIndex` and specifying the relative position of the font. Like `gopenFont`, it too returns a font ID if successful.

Multiple font files can be created by version 2.00 upwards of the font compiler and are given extensions of `.FON`, the same as for single font files.

Using wsfcamp

The *wsfcamp.exe* program (which is placed in the `\sibosdk\sys` directory during installation of the core software) compiles a font source file (which normally has the extension *.fsc*) to a font file (*.fon*). A font file should always be generated from a font source file by the use of *wsfcamp.exe*: developers should not attempt to generate font files directly.

To compile *mono.fsc* to produce *mono.fon* you use:

```
wsfcamp mono
```

The font source file contains a header section followed by the patterns which make up the font, as in the following extract from the beginning of a font source file:

```
*name System mono
*special 1
*descent 2
*height 10
*maxwid 7
*flag ascii
*flag cp850
*char 28
```

```
100000000000
100001100000
100001100000
101111111100
100111111000
110011110111
000001100000
000000000000
111111111111
100000000000
```

```
0000000
0000000
0001000
0001100
1111110
1111110
0001100
0001000
0000000
0000000
```

```
0000000
0000011
0000011
0010011
0110011
1111111
1111111
0110000
0010000
0000000
```

where the keywords are as follows:

*name	The font name (maximum length 16 characters)
*special 1	An optional statement to produce the 'special' fast font. It must come before the *height statement. The parameter is ignored currently but should be 1 to allow for future upgrades.
*descent <n>	The descent for the font.
*height <n>	The height of the font.
*maxwid <n>	Not the real maximum width, but the width of widest normal character. Used as the width for G_STY_MONO spacing and is returned in the G_FONT_INFO struct.
*flag ascii	Sets the flags member in the G_FONT_INFO struct.

*flag cp850
*flag bold
*flag italic
*flag serif

*char <n> Skip all characters up to the specified code. Character codes follow sequentially until the next *char <n> statement. The character number must always be greater than or equal to what the character would have been.

In version 2.00 upwards of *wsfcomp.exe*, multiple source files can be converted into a multiple font file by specifying the -L flag and creating a *.flk* file containing a list of *.fsc* files to be compiled. The output file is a *.fon* file of the same name as the *.flk* file. A *.fh* header file is also created containing a #define for the index value of each font built and also a #define for the number of fonts built.

The #define for each index is built as follows:

```
#define <FLK file name>_FI_<FSC file name> index_value
```

The #define for the count is built as follows:

```
#define <FLK file name>_COUNT count
```

This version of the font compiler also supports the conversion of PCL bitmap fonts (as used by the HP laser jet printer) to either *.fsc* source files or directly to *.fon* files.

To use a PCL file as source, use the -C flag. By default a *.fon* file is created but by using the -F flag as well, a *.fsc* file is created instead. Note that the default extension for the PCL source file is *.sfp*.

Also in version 2.00 of *wsfcomp*, output file names can be overridden by using the -O<filename> option.

Font information

Given a font ID (either of a ROM-based font or of a loaded font), you can call *gFontInfo* to obtain such information about the font as:

- the name of the font and whether it corresponds to a character set
- the lowest and highest code for which there is a graphic
- the height of the font and how much of that height is above and below the base line (called the ascent and descent respectively)
- whether the font is monospaced, the width of the character 0 (ASCII code 0x30) and the width of the widest character graphic

Note that in addition to the font ID, *gFontInfo* takes as parameters the style (eg bold, double height) with which that font is to be printed because the width and height of a font can be changed by the style.

The following functions are required when laying out text (for example, to word wrap):

<i>gTextWidth</i>	Calculates the printed width of a text string.
<i>gTextCount</i>	Calculates the number of characters that can be displayed in a given pixel width.
<i>wGetWidthTable</i>	Gets the font width table.

These functions also take the font ID and the style as parameters.

Text drawing

A text string may be printed to a drawable (as specified by the current graphics context) using one of the following functions:

<i>gPrintText</i>	to draw text from a pixel position
<i>gPrintClipText</i>	to draw text clipped to a given width
<i>gPrintBoxText</i>	to draw text in a box with left, right or centred alignment (commonly used to draw flicker-free)
<i>gXPrintText</i>	to draw text with embellishment
<i>gShadowText</i>	to draw text with a shadow effect (version 4)

All the above functions are subject to the values of the text drawing fields in the current graphics context. These specify:

- the font ID
- the font style (normal, bold, underlined, inverse, double height, monospaced or italic)
- the text transfer mode (this is not used by `gPrintBoxText`, `gxPrintText` or `gShadowText`)

The font style is either `G_STY_NORMAL` (zero) for normal style or a combination of the following:

- `G_STY_BOLD`
- `G_STY_UNDERLINE`
- `G_STY_INVERSE`
- `G_STY_DOUBLE`
- `G_STY_MONO`
- `G_STY_ITALIC`

Prior to version 4, the above styles are applied to a font algorithmically in the following way:

<code>G_STY_BOLD</code>	Generated by "bolding" character graphics - effectively drawing it twice with a one pixel right shift on the second draw. This does not produce as good a result as designing a bold font but it saves a having to store another font.
<code>G_STY_UNDERLINE</code>	Where each character (including space) is drawn with a horizontal line beneath its graphic. See also <code>gxPrintText</code> for a different style of underlining.
<code>G_STY_INVERSE</code>	Where the bits in the character graphics are inverted before drawing.
<code>G_STY_DOUBLE</code>	To generate double height characters where each row of pixels in the graphic is doubled up before drawing. As with bolding, you will get a much better result by designing a taller font.
<code>G_STY_MONO</code>	To add space around the characters of a proportional font to turn it into a monospaced font. For this to be effective, the proportional font should be designed with monospacing in mind (as is the built in font on the Series 3).
<code>G_STY_ITALIC</code>	To generate a crude (but surprisingly effective) italic font by shifting the top half of each character across by one pixel.

Fast text printing of a fast font only works with normal style or with just `G_STY_MONO`. The use of any other style will cause text drawing to fall back to the slower algorithm.

In version 4 upwards, use can be made of font groups as described at the beginning of the section on *ROM-based fonts*; applying a style to a font group will cause the window server to select the most appropriate font from within the group and *might* not need to use the above algorithms.

Therefore, if a font is used with a style or combination of styles, there is no easy way of knowing whether a font within a font group has been used or whether a style has been algorithmically applied to a single font or even whether styles have been applied to a font within a font group.

The text transfer mode (which affects `gPrintText` and `gPrintClipText`) is one of:

<code>G_TRMODE_SET</code>	where 1s in the font set bits in the destination and 0s do not change bits in the destination (used to print on to a previously cleared area).
<code>G_TRMODE_REPL</code>	where 1s and 0s in the font overwrite the destination (used to print over unprepared areas).
<code>G_TRMODE_CLR</code>	where 1s in the font clear bits in the destination and 0s do not change bits in the destination (used to print on to a previously set area).
<code>G_TRMODE_INV</code>	where 1s in the font toggle corresponding bits in the destination and 0s in the source pattern do not change bits in the destination (used to print over an existing image and may be reversed by a second application).

The physical structure of font files

The information in this section is intended to allow applications to read data from a font file. A font file should always be created by use of *wsfcomp.exe*.

As with bitmap files, font files start with a P_FSIG header:

```
typedef struct
{
    TEXT app_id[3]; /* application ID */
    UBYTE chk_sum; /* application ID checksum */
    UBYTE file_vn; /* file version number */
    UBYTE app_vn; /* application version number */
} P_FSIG;
```

where P_FSIG is defined in *p_file.h*.

For a normal font file, the appropriate values for the P_FSIG header are:

```
P_FSIG sig = {"FON", 'F'+ 'O'+ 'N', 0x30, 0x30};
```

For a fast font file, the appropriate values are:

```
P_FSIG sig = {"FN1", 'F'+ 'N'+ '1', 0x10, 0x10};
```

The window server automatically recognises the difference between normal and fast fonts. However, window servers before version 3.5 do not recognise fast fonts and will refuse to load them.

The P_FSIG header is followed by the data which makes up the WS_FONT_FILE_HEADER (not supplied in any *\sibosdk\include* header file - define your own if you need it):

```
typedef struct
{
    P_FSIG sig;
    UWORD checksum;
    UWORD size;
    G_FONT_INFO info;
    UWORD private[10];
} WS_FONT_FILE_HEADER;
```

The checksum is calculated by applying *p_crc* to the data that follows the WS_FONT_FILE_HEADER header (the character width array and the font bitmap).

The size is the byte size remaining in the file after the size field.

The G_FONT_INFO struct is as returned by *gFontInfo*:

```
typedef struct
{
    UWORD low_ch; /* lowest code in font */
    UWORD high_ch; /* highest code in font */
    UWORD height; /* height of font */
    UWORD descent; /* height of bottom part of character */
    UWORD ascent; /* height of top part of character */
    UWORD numeric_width; /* Width of numeric character */
    UWORD max_width; /* The width of the widest character in the font */
    UWORD flags;
    TEXT name[FONT_NAME_LEN];
} G_FONT_INFO;
```

The G_FONT_INFO struct is followed by 20 bytes of private data, written into the file by *wsfcomp.exe*.

In a normal font, the WS_FONT_FILE_HEADER is followed by:

- an array of $(high_ch - low_ch + 2)$ WORD offsets into the font bitmap
- the font bitmap, stored like the body of a bitmap in a bitmap file

The font bitmap contains the characters in the font as one long string in code order. The window server literally bit copies the characters from this font bitmap to the drawable (the screen or a bitmap).

Each word in the array of offsets contains the doubled pixel offset within the font bitmap with the least significant bit of the word set to 1 if the character is missing from the font and left as 0 otherwise. The offset corresponding to a missing character must be that of the next existing character. An extra offset is stored at the end of the array containing what would have been the offset of the next character so that the width of a character may be calculated by subtracting its offset from the offset of the next character.

Fast fonts are stored in an expanded form that uses more memory but can be drawn faster. In a fast font, the `WS_FONT_FILE_HEADER` is followed by:

- an array of 256 `BYTE` character pixel widths (each set to between zero and 8, inclusive)
- the font bitmap for all 256 characters (which is $256 \times \text{height}$ bytes long), stored like the body of a bitmap in a bitmap file

All characters in a fast font must be less than or equal to 8 pixels wide. The font bitmap contains all 256 characters, stored left aligned in their 8 pixel slot. Unused bits are, by convention, set to zero.

Embedded font files

A font file may be built into a program file by including its name in an add-file list. This process is more fully described in the *Building an Application* chapter of the *General Programming* manual.

Graphics output

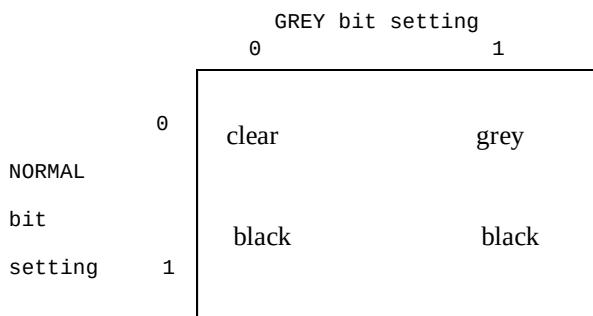
Pixel coordinates

The graphics output functions (and other functions such as window creation) operate on a drawable - either a window or a bitmap. In either case, you are actually drawing a two-dimensional array of pixels in which each pixel is either black or white.

Version 4 of the window server can support grey as well as black. In this case you draw to a *pair* of two-dimensional arrays, referred to as the normal (or black) plane and the grey plane.

Arrays of pixels are represented by data structures called *bitmaps* (whether on or off the screen) where the bits in the bitmap represent pixels. When a bitmap is imaged on an LCD screen, a clear bit shows as white and a set bit as either black or grey.

With a mixture of black and grey, the situation is more complicated; two arrays of pixels are needed to display an image which means that two bitmaps are needed. When the pair of bitmaps are imaged on an LCD screen, what is seen depends on the corresponding bit values of *both* planes. This is best illustrated by the following diagram:



Note that a bit set in the normal plane and in the corresponding position in the grey plane is seen as black.

Internally, a bit in a bitmap is addressed by the index of the word that contains it and the position of the bit within that word. Externally, a bit is addressed (somewhat more usefully) by its pixel coordinates.

When thinking of pixel coordinates, it is best to think in terms of discrete cells arranged in rows and columns rather than in terms of continuous Cartesian coordinate systems based on real numbers⁹. Pixel coordinates are based on a pair of integers (x,y) that identify a particular pixel by the column (x) and the row (y) in which it lies.

⁹Although such a view may well be appropriate at higher levels.

In a bitmap or window, the top-left pixel has coordinates (0,0). The x coordinate is based on the column number (which increases to the right) and the y coordinate is based on the row number (which increases downwards). Pixel coordinates are signed and can address pixels that are outside the bitmap.

It is often useful to address pixels that are beyond the boundaries of the bitmap. When any drawing operation is performed, it is clipped to the extent of the bitmap. When drawing to a window, the drawing operation may be clipped to much else besides the window's extent.

There is no graphics function that draws a single pixel since this is too inefficient to be of use¹⁰. In practice, pixel coordinates are used to specify:

lines	by the coordinates of the ends of the lines
rectangles	by the coordinates of the top left internal pixel and the coordinates of the bottom right external pixel

When drawing a horizontal line the line includes the pixel with the lower x coordinate and excludes the pixel with the higher x coordinate.

Similarly, when drawing a vertical line the line includes the pixel with the lower y coordinate and excludes the pixel with the higher y coordinate.

When drawing a line in which both coordinates change, the window server turns the coordinates of the end pixels into a rectangle with a top-left internal pixel and a bottom-right external pixel. The line drawing algorithm then fills in those pixels that are intersected by a mathematical line between the corners of the mathematical rectangle on the boundary of those pixels that are in the rectangle.

Surprisingly, when the rectangle has unit height or width, you get a horizontal or a vertical line respectively as with a zero height or width.

There are some useful functions that operate on rectangle data structures in the *Integer Conversions and Rectangle Functions* chapter of the *PLIB Reference* manual.

Rectangles can also be specified by the coordinates of the top left (internal) pixel and the width and height of the rectangle in pixels.

Graphics structs

The structs described below (defined in *p_graf.h*¹¹) are used extensively by the WLIB functions.

The `P_POINT` struct is used to represent a pixel coordinate or a displacement:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;
```

The `P_RECT` struct specifies a rectangular block of pixels in terms of the top-left internal pixel and the bottom-right external pixel:

```
typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;
```

The `P_EXTENT` struct also specifies a rectangular block of pixels - but in terms of the top-left internal pixel width and height of the block in pixels:

```
typedef struct
{
    P_POINT tl; /* top left pixel */
    WORD width; /* width of extent */
    WORD height; /* height of extent */
} P_EXTENT;
```

An empty rectangle is a rectangle that has one or both of its sides zero or negative.

¹⁰If you really must set individual pixels, you can use `gClrRect` on a rectangle with unit sides.

¹¹The header *p_graf.h* is also included by including *plib.h*.

Graphics output

The following graphics output functions operate on the drawable that is referenced by the current graphics context:

<code>gDrawLine</code> <code>gDrawPolyLine</code>	to draw horizontal, vertical or angled lines
<code>gDrawBox</code>	to draw a box
<code>gBorderRect</code> <code>gBorder</code>	to draw a border inside a rectangle in one of a number of styles
<code>gBorder2Rect</code> <code>gBorder2</code>	to draw a border inside a rectangle with a shadowed effect. Available in version 4 only.
<code>gDrawObject</code>	to draw a scaleable graphics object inside a rectangle. Available in version 4 only.
<code>gClrRect</code>	to set, clear or invert a rectangle of pixels
<code>gInvObloid</code>	to invert all the pixels in a rectangle except for the four corner pixels
<code>gCopyBit</code>	to copy a rectangle from a bitmap (or a backed-up window) to a given position
<code>gDrawBit</code>	to copy a rectangle from an open bitmap file. Available in version 4 only.
<code>gCopyRect</code>	to copy a rectangle to a displaced position (only really suitable for bitmaps - <code>wScrollRect</code> should be used for windows)
<code>gFillPattern</code>	to fill a rectangle with repeated copies of a bitmap (or a backed-up window)
<code>gPrintText</code>	to draw text from a pixel position
<code>gPrintClipText</code>	to draw text that is clipped to a given width
<code>gXPrintText</code>	to draw text with embellishment
<code>gPrintBoxText</code>	to draw text in a box with left, right or centred alignment
<code>gShadowText</code>	to draw text with shadow effects. Available in version 4 only.
<code>wDrawButton</code>	to draw a S3 style button (essentially a rectangle containing a text string), depressed or not. Available in version 3.5 of the window server, but not available on any version of the MC
<code>wDrawButton2</code>	to draw a button in both the S3 and S3a styles. Available in version 4 only.

Graphics output other than the current graphics context

There is a set of graphics output functions are not directed at the current graphics context. These functions are:

<code>wScrollRect</code> <code>wScrollWin</code>	to scroll a window or a bitmap by copying a rectangle to a displaced position.
<code>wTextCursor</code> <code>wDrawTextCursor</code> <code>wEraseTextCursor</code>	to control the drawing of a flashing rectangular text cursor in a window.
<code>wInfoMsg</code> <code>wInfoMsgCorner</code> <code>wSetBusyMsg</code> <code>wCancelBusyMsg</code>	to display temporary text messages at the corners of the screen. Requires version 3.5 of the window server, but is not available on any version of the MC.
<code>wsCreateClock</code> <code>wsSetClock</code>	to draw and update a S3 style clock (analog or digital) in a window. Requires version 3.5 of the window server, but is not available on any version of the MC.
<code>wsCreateClock2</code>	to draw a S3a style clock (analog or digital) in a window. Requires version 4 of the window server.

wsEnable	to control the display of a task status window. Intended primarily for the S3 but
wsUpdate	available also on the HC with version 3.5 of the window server. Not available
wsDisable	on the MC.
wsEnableTemp	
wsDisableTemp	
wsStatusWindow	to control the state of a status window and the display of modes within that
wsSetList	window. Intended for the S3a running version 4 of the window server.
wsSelectList	
wsAlertW	to access p_notify-like services but with enhanced formatting and an
wsAlertA	asynchronous version (so that an application can continue processing while
wsAlertUpdate	waiting for the user to respond). Requires version 3.5 of the window server, but
wsAlertCancel	is not available on any version of the MC.

You can also draw to a window indirectly by attaching one or more bitmaps to it (using `wSetWinBitmap`) and then draw to the bitmaps.

Graphics contexts

A graphics context is a window server resource at which graphics output is directed.

A graphics context contains the following:

- a drawable (a window or a bitmap) that is the ultimate recipient of the graphics output
- whether to set, clear or invert pixels when drawing lines using `gDrawLine`, `gDrawPolyLine`, `gDrawBox`, `gBorderRect`, `gBorder`, `gBorder2Rect`, `gBorder2`, `wDrawButton` or `wDrawButton2`
- the font, style and transfer mode to use when drawing text using `gPrintText`, `gPrintClipText`, `gXPrintText`, `gPrintBoxText`, `wDrawButton` or `wDrawButton2`

The drawable is set once and for all when the graphics context is created. The rest of the content can be set up when the graphics context is created and it can also be altered subsequently using `gSetGC`.

Current graphics context

Rather than make each function take the ID of a graphics context as a parameter, the graphics output functions are directed to a *current graphics context*.

Using a current graphics context makes sense because you typically need to call several text drawing or line drawing functions (all to the same graphics context) to draw something.

When you create a graphics context, it is automatically made current. If you have created more than one permanent graphics context, you can set which one of them is current by calling:

<code>gSetGC</code>	to make the specified graphics context current and to change its value
<code>gSetGC0</code>	just to make the specified graphics context current

Permanent and temporary graphics contexts

You can draw to a *permanent* or a *temporary* graphics context.

When your program draws to more than one drawable, it is normally simpler to use a temporary graphics context that is created and then freed each time you draw to one of the drawables.

A client can only create one temporary graphics context at any one time. When a temporary graphics context is created, it is automatically made current. If a permanent graphics context was current before a temporary graphics context was created, the permanent graphics context is made current when the temporary graphics context is freed.

Creating a temporary graphics context is an efficient "blind" operation, which is stored in the client-side buffer.

Creating a permanent graphics context

To create a permanent graphics context for drawing to any drawable, you use:

<code>gCreateGC</code>	to create a permanent graphics context with specified initial values
<code>gCreateGC0</code>	to create a permanent graphics context with default initial values

A permanent graphics context is freed by calling `wFree`.

Creating a temporary graphics context

There are six functions that create a temporary graphics context: two for drawing to any drawable and four for redrawing to a window.

The two that are used for drawing to any drawable are:

<code>gCreateTempGC</code>	to create a temporary graphics context with specified initial values
<code>gCreateTempGC0</code>	to create a temporary graphics context with default initial values

A temporary graphics context that is created with one of the above functions is freed with `gFreeTempGC`.

The four that are used for redrawing to a window are:

<code>wBeginRedrawGC</code>	to redraw a part of the window using a temporary graphics context that is created and initialised with specified values
<code>wBeginRedrawGC0</code>	to redraw a part of the window using a temporary graphics context that is created with default initial values
<code>wBeginRedrawWinGC</code>	to redraw the whole of the window using a temporary graphics context that is created and initialised with specified values
<code>wBeginRedrawWinGC0</code>	to redraw the whole of the window using a temporary graphics context that is created with default initial values

A temporary graphics context that is created with one of the above functions is freed with `wEndRedraw`.

Keyboard input

Key press events

A client is informed of a key press by `wGetEventWait`, `wGetEvent` or `wGetEventSpecial` completing with a `WM_KEY` event type.

The `WM_KEY` event parameters are:

- a key code
- a code modifier that indicates what shift keys were pressed at the time
- a repeat count

The key code may correspond to a code page 850 character code or, if the key does not correspond to a character in code page 850, a special key code (for example, `W_KEY_LEFT` for the left arrow key).

For single key presses, the repeat count is 1. If a key is held down, the repeat count will normally be greater than 1. A repeat count gets to be greater than 1 when the client is unable to process keys at the rate at which the system generates repeated keys.

Counter intuitively, application programmers are best advised just to ignore the repeat count.

When a user holds down a key, there is presumably some feedback on the screen of the processing of that key. The user will release the key when the screen image is such that enough key presses have been generated (think of, for example, deleting text with the `BACKSPACE` key or scrolling with a down arrow key). When the user releases the key, he or she will expect the processing to cease as soon as the key is released.

This behaviour is exactly what you get by ignoring the repeat count. If you process the key count, this only benefits users that can anticipate the number of repeats generated by holding the key down for a given period (and that kind of ability is arguably only required when playing a computer game).

Capturing keys

Normally, the window server sends all key presses to the foreground client.

All key presses, that is, except those that the window server processes itself (which includes keys to task switch and the key that pauses the foreground client).

In much the same way as it is useful for the window server to process keys, it is sometimes useful for a special client to receive keys whether it has the foreground or not. Such a client can capture specific key presses by calling `wCaptureKey` (and such captures can be cancelled by calling `wCancelCaptureKey`).

Pausing the client

If a client has called `wEnablePauseKey`, the user can pause the client's drawing when it has the foreground by pressing the pause key.

On the HC, the pause key is `PSION+LEFT-ARROW` and on the S3, S3a and MC, it is `CTRL+S`. The user resumes the client by pressing any key. The key press that resumes the client is not delivered to the client.

The pause key may be disabled by calling `wDisablePauseKey`. The pause key is disabled by default.

Setting task keys

From version 3.5 onwards of the window server you can call:

`wSetTaskKey` to set and cancel a key which cycles the foreground task
`wCancelTaskKey`

`wSetBackTaskKey` to set and cancel a key which cycles the foreground task in the opposite
`wCancelBackTaskKey` direction

These task keys are in addition to the normal system task keys on the HC and MC. This feature is provided for the S3 and S3a (which do not have a system task key on the keyboard).

Note that the system screen calls `wSetTaskKey` to assign `SHIFT+SYSTEM` as a task key.

Mouse input

This section only applies when the window server is running on a machine with a pointing device (a digitiser or a mouse).

Mouse icons

The window server automatically draws a corresponding mouse icon when the pointing device is over a window.

Each window is assigned a mouse icon when it is created using `wCreateWindow` and the mouse icon can subsequently be changed by calling `wSetWindow`. By default, a window is created with the same mouse icon as its parent. Unless otherwise specified, windows tend to have the standard built-in mouse icon with ID `W_WIN_MI_STANDARD` - the mouse icon used by the root window.

A window can effectively have no mouse icon by giving it a built-in invisible icon with ID `W_WIN_MI_NULL`.

The window server automatically removes the mouse icon while a window it obscures is being drawn to.

The mouse icon is drawn such that its "hot spot" (which varies according to the icon) is over the pixel that is being pointed at.

Mouse icons are handled much like bitmaps and fonts:

- There are ROM-based mouse icons with IDs that are known at compile time.
- You can load a mouse icon from a file by calling `gopenMouseIcon` (which returns an ID if successful). Such mouse icons may be freed when no longer required by calling `wFree`.

Mouse events

A client is informed of a mouse event (movement, press or release) by `wGetEventWait` or `wGetEvent` completing with a `WM_MOUSE` event type.

The `WM_MOUSE` event parameters are:

- the window at which the mouse event is directed and the position of the mouse relative to that window
- the type of the mouse event (movement, press or release)
- the state of the mouse button and of the shift keys

A window will not receive any mouse events at all if the `W_WIN_NO_MOUSE` attribute is set (in which case the other mouse related attributes are ignored).

By default, mouse movement events are not generated. If the `W_WIN_MOUSE_DRAG` attribute is set, mouse movement events are reported when the mouse button is down. If a window has the `W_WIN_MOUSE_MOVE` attribute set, mouse movements are reported when the mouse button is up.

Capturing the mouse

Normally, a mouse event is directed to the window that was pointed to when the event occurred. If the mouse has been captured by a window, a mouse event that occurs in the root window or in any of the capturing client's windows is directed at the capturing window or to the relevant child window of the capturing window. There is a mouse event parameter that indicates whether the position was inside or outside the window.

Capturing is often used to limit the mouse interaction to a window system to the exclusion of others as, for example, in the implementation of a pull-down or a pop-up menu.

A client can capture the mouse to a particular window by calling `wCaptureMouse` and release the mouse by calling `wReleaseMouse`.

Grabbing the mouse

The mouse can also be "grabbed" while the mouse button is pressed by setting the `W_WIN_MOUSE_GRAB` attribute on a window. The grab is automatically released when the user releases the mouse button and a `WM_MOUSE` event of type `WM_MOUSE_RELEASE` is sent to the grabbing window.

Grabbing the mouse is not quite the same thing as capturing it. When the mouse is grabbed, the grabbing window always gets the `WM_MOUSE_RELEASE` event - even if it occurs over another client's window. Also, grab is terminated with the mouse up, but capture persists until it is relinquished.

Input-only windows

A window that is used for mouse input but that is never drawn to should have the `W_WIN_INPUT_ONLY` attribute set.

Input-only windows are never visible on the screen (and they do not obscure other windows) although they may (and typically would) have a different mouse icon from their surroundings.

Inactive windows

A window tree is made inactive by setting the `W_WIN_INACTIVE` attribute of the window at the head of the tree.

If a mouse click occurs anywhere in a window with this flag set or in any of its descendants, a `WM_ACTIVE` event is sent to the window (rather than a `WM_MOUSE` event). A client would normally respond to a `WM_ACTIVE` event by making the window tree active again by clearing the `W_WIN_INACTIVE` attribute.

The rubber band

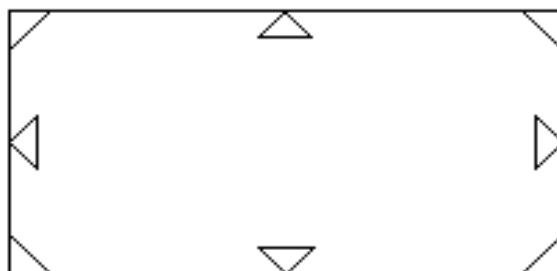
The rubber band is an input mode, controlled by the window server, for allowing the user to do one of the following:

- specify a new position for a rectangle
- specify a new position and size of a rectangle

Rubber band mode is typically used to allow the user to move and/or resize a window.

Once a client has entered rubber band mode, it must be completed before proceeding with any other input (although the user might abandon it). While in rubber band mode, the window server handles all user interaction with the rubber band. When the user completes the rubber band, the window server generates a `WM_RUBBER` event.

A rubber band that is used just to move (but not resize) appears as an XORed box. The rubber band for resizing as well as moving appears as in the following diagram:



Each of the 8 triangles may be dragged with the mouse to stretch or shrink the rubber band in the direction of the triangle. Releasing the mouse will leave the rubber band in its stretched state. The mouse icon disappears while the rubber band is being dragged.

To move the rubber band, it should be dragged from a point inside the box but outside any of the triangles.

The arrow keys can be used to move the rubber band in the corresponding direction. The shifted arrow keys can be used to resize the rubber band moving the bottom and right edges. If the CTRL shift key is also pressed, the affect of the arrow keys (whether shifted or not) is amplified.

Pressing RETURN selects the current position and size of the rubber band. The window server generates a WM_RUBBER event giving the position and size of the rubber band and also indicating whether the rubber band moved and/or changed its size.

Pressing ESC cancels the rubber band. The window server generates a WM_RUBBER event that indicates that the rubber band was cancelled.

Rubber band mode is entered by the client calling `wRubberBand`. The parameters to `wRubberBand` include those that indicate:

- the initial size and position of the rubber band
- the window at which the WM_RUBBER event is directed
- the window in which the rubber band is drawn (and therefore clipped to)
- whether the rubber band is completed on a mouse up event
- whether the rubber band can be resized as well as moved
- optional limits on the size and movement of the rubber band
- an optional grid snap on the allowable positions and sizes of the rubber band

If a mouse down event occurs in a window that is set with the `W_WIN_RUBBER_BAND_CAPTURE` attribute, it and subsequent mouse and keyboard events are captured to the window server's rubber band processing until the rubber band mode is terminated. This first mouse click generates a `WM_RUBBER_BAND_INIT` event to which the client must respond with a call to `wRubberBand`.

If when calling `wRubberBand` in response to a `WM_RUBBER_BAND_INIT` event you specify that the rubber band should be completed on a mouse up event, you should also set the window attribute `W_WIN_RUBBER_BAND_COMPLETE_ON_RELEASE` (this handles nasty race conditions in which the mouse up occurs before the call to `wRubberBand`).

System start-up

When the system starts up (for example, when you press the recessed reset button), the following happens:

- the operating system "hand crafts" `SYS$NULL` (null process), `SYS$MANG` (supervisor) and `SYS$FSRV` (file server)
- the file server creates the `SYS$WSRV` (window server) process
- the window server creates the `SYS$SHLL` (shell) process
- in some cases, the window server creates the `SYS$NTFY` (notifier) process

The window server normally loads the shell and the notifier from the ROM. However, on the HC, S3, S3a and MC, the window server first looks for `SYS$SHLL.IMG` and `SYS$NTFY.IMG` in the root directories of the local devices in alphabetical order (for example, `LOC::A\`, `LOC::B:\` and `LOC::M` on an HC).

In some cases (such as on the S3, S3a and optionally on the HC), version 3.5 and upwards of the window server handles the notifier itself and does not create a notifier process.

Using the window server's notifier on the HC

By default on the HC, the window server runs `ROM::SYS$NTFY.IMG`. If the HC is running version 3.5 of the window server, you can switch to the window server's notifier with the following benefits:

- less memory is consumed by the system
- it looks better

The HC can be switched to the window server's notifier with the following program:

```
#include <plib.h>
#include <wlib.h>

LOCAL_D WSERV_SPEC wSpec;

GLDEF_C INT main(VOID)
{
    INT NotifierPid;

    if ((NotifierPid=p_pidfind("SYS$NTFY.*"))>0)
    {
        wConnect(&wSpec,0,W_CONNECT_AT_BACK);
        wSystem(WSERV_FLAG_NO_NOTIFIER_REBOOT,WSERV_FLAG_NO_NOTIFIER_REBOOT);
        p_terminate(NotifierPid,0);
        wSystem(WSERV_FLAG_HOOK_NOTIFIER,WSERV_FLAG_HOOK_NOTIFIER);
    }
    return(0);
}
```

If SYS\$NTFY exists, the program connects to the window server and:

- stops the window server from re-booting the notifier
- kills the notifier process
- causes the window server to hook the notifier

On the HC, the font used by the notifier is determined by the Internal Font environment variable, \$WS_IF, which should contain a WORD binary value of 0 for WS_FONT_BASE, 1 for WS_FONT_BASE+1, and so on.

If you change the value of \$WS_IF, you must reset the machine by pressing the recessed reset button to effect the change. The height of the font should not exceed 12 pixels.

The "factory" setting of \$WS_IF is 4 (which selects the S3 font).

In version 4 of the window server, the fonts used by the notifier are determined by the \$WS_FNTS environment variable. This contains a number of words containing the font ids used by the window server as described at the beginning of this chapter.

In particular, the second word contains the ID of the notifier font while the seventh word contains the ID of the notifier button font.

The \$WS_FL environment variable on the HC

On an HC with version 3.5 of the window server, the initial value of the internal parameter that is set by wSystem is loaded from the \$WS_FL environment variable when the window server starts.

After setting \$WS_FL to the desired WORD value, you must reset the HC by pressing the recessed reset button to make the new value effective.

Recall that environment variables survive a soft reset but are cleared (and loaded from a ROM initialisation file) on a hard reset (where the ON key is pressed at the same time as the reset button).

The wSystem flags parameter is made up by ORing a number of bit fields of the form WSERV_FLAG_XXX where some of the values of xxx are¹²:

NO_NOTIFIER_REBOOT	If set, the window server does not boot or re-boot the notifier.
HOOK_NOTIFIER	If set, the window server attempts to hook the notifier.
NO_PANIC_NOTIFY	If clear and the window server has successfully hooked the notifier, the window server notifies the user of a process that terminates abnormally with a panic or with a negative reason number. This flag is ignored unless the window server has hooked the notifier. You would set this flag to prevent the window server from reporting abnormal terminations when HOOK_NOTIFIER is set.

¹²See the description of wSystem for the full set and also for the application of these flags to the S3, the S3a and the MC.

LOW_BATTERY_WARNINGS	If set, the window server notifies the user of a low lithium or main battery. Note that the window server only checks for low battery when the machine is turned on and that the window server is only informed of the machine being switched on after <code>p_setonevent(TRUE)</code> has been called. The on-event state is FALSE after any reset.
HUNG_UP	If set, the window server presents a "hung up" status window if the foreground task is not using backed-up windows and fails to respond to redraw events.

After a hard reset on an HC with version 3.5 of the window server, the `$WS_FL` environment variable does not exist (which is equivalent to it being zero).

The following example program sets the `$WS_FL` environment variable:

```
#include <plib.h>
#include <wlib.h>

GLDEF_C INT main(VOID)
{
    WORD flags;

    flags=WSERV_FLAG_NO_NOTIFIER_REBOOT|WSERV_FLAG_HOOK_NOTIFIER
        |WSERV_FLAG_LOW_BATTERY_WARNINGS|WSERV_FLAG_HUNG_UP_SW;
    return(p_setenv("WS_FL",6,&flags,2));
}
```

After running this program and resetting the HC, the window server will:

- provide the notifier service
- report low battery voltages
- present a hung-up status window if an application hangs
- report a process that terminates with a panic or with a negative reason number

S3a

In version 4, some HC features from the later releases of version 3.5 have been added to the S3a variant of the window server. This includes:

- the use of the environment variable `$WS_FL` for the initial value of the window server system flags (as referenced by `wSystem`).

Replacing the shell on the HC

By default, the window server runs `ROM: :SYS$SHLL.IMG`. This program provides a classical command-driven user interface to perform such commands as `COPY` and `DIR` and to run programs (for example, `LINK`) in response to the program's file name being entered.

When developing a turnkey system, it is advisable to run an alternate custom shell that provides a restricted end-user environment. With version 3.5 of the window server, this is particularly straightforward as the window server can be persuaded to take over many of the responsibilities - particularly the link paste services - that previously had to be provided by the shell. You simply call your main application `SYS$SHELL.IMG` and place it in the root directory of any drive. The window server will run the shell on system start-up and it will also re-run the shell should it terminate.

Since any restarting EPOC system will pick up a `SYS$SHLL.IMG` (such as the debugger, for example), it is a good idea *not* to call such a program `SYS$SHLL` but to rename it when you copy it to its intended working destination.

To revert to the ROM shell when using an SSD-based `SYS$SHLL`, first remove the SSD and then either terminate the existing shell process or reset the machine. If you have placed a `SYS$SHLL.IMG` in `M:\`, you can revert to the ROM shell by hard resetting the machine (which clears the contents of `M:\` and resets the environment variables) or you can place an alternate `SYS$SHLL.IMG` in an SSD drive (since the drives are scanned in alphabetic order).

The following simple shell/application program terminates any `SYS$NTFY` process and sets up the window server to provide the notifier service and other services (as described above) and then presents a user interface that reports on key presses.

```
/*
HCSHELL.C - Sample shell for the HC
*/

#include <plib.h>
#include <wlib.h>

GLREF_D UINT wMainGc;
GLREF_D WSERV_SPEC wSpec;

LOCAL_D INT FontHeight;
LOCAL_D INT FontAscent;

LOCAL_C VOID SetFontHeight(VOID)
{
    G_FONT_INFO info;

    gFontInfo(WS_FONT_SYSTEM,0,&info);
    FontHeight=info.height;
    FontAscent=info.ascent;
}

LOCAL_C VOID CDECL PrintLine(INT line,INT align,TEXT *fmt,...)
{
    INT len;
    P_RECT box;
    TEXT b[80];

    box.tl.x=4;
    box.br.x=wSpec.conn.info.pixels.x-4;
    box.tl.y=FontHeight*line+4;
    box.br.y=box.tl.y+FontHeight;
    len=p_atob(&b[0],fmt,&fmt+1);
    gPrintBoxText(&box,FontAscent,align,0,&b[0],len);
}

LOCAL_C VOID HandleKeyPress(WMSG_KEY *pk)
{
    PrintLine(2,G_TEXT_ALIGN_CENTRE,"code:%02x mod:%02x count:%02x",
        pk->keycode,pk->modifiers,pk->count);
}

LOCAL_C VOID MainEventLoop(VOID)
{
    WS_EV event;

    SetFontHeight();
    gBorder(W_BORD_SHADOW_D|W_BORD_SHADOW_ON);
    PrintLine(0,G_TEXT_ALIGN_LEFT,"Free memory: %dKbytes",p_sgfree())>>6);
    for (;;)
    {
        wGetEventWait(&event);
        if (event.type==WM_KEY)
            HandleKeyPress(&event.p.key);
    }
}

GLDEF_C INT main(VOID)
{
    INT NotifierPid;
```

```
p_setonevent(TRUE); /* required on the HC */
wStartup();
wSystem(WSERV_FLAG_NO_NOTIFIER_REBOOT,0xffff);
if ((NotifierPid=p_pidfind("SYS$NTFY.*"))>0)
    p_terminate(NotifierPid,0);

wSystem(WSERV_FLAG_HOOK_NOTIFIER|WSERV_FLAG_LOW_BATTERY_WARNINGS|WSERV_FLAG_HUNG_UP_SW
,

WSERV_FLAG_HOOK_NOTIFIER|WSERV_FLAG_LOW_BATTERY_WARNINGS|WSERV_FLAG_HUNG_UP_SW);
MainEventLoop();
return(0);
}
```

You may prefer to use the following shell (the source may be found in `\sibosdk\demo\lkshell.c`) when using the remote debugger on the HC:

```
/*
LKSHELL.C - Just starts up the link
*/

#include <plib.h>
#include <wlib.h>

LOCAL_D WSERV_SPEC wSpec;
LOCAL_D UINT wMainGc;
LOCAL_D UINT wMainWid;
LOCAL_D INT FontHeight;
LOCAL_D INT FontAscent;

LOCAL_C VOID SetFontHeight(VOID)
{
    G_FONT_INFO info;

    gFontInfo(WS_FONT_SYSTEM,0,&info);
    FontHeight=info.height;
    FontAscent=info.ascent;
}

LOCAL_C VOID CDECL PrintLine(INT line,INT align,TEXT *fmt,...)
{
    INT len;
    P_RECT box;
    TEXT b[80];

    box.tl.x=4;
    box.br.x=wSpec.conn.info.pixels.x-4;
    box.tl.y=FontHeight*line+4;
    box.br.y=box.tl.y+FontHeight;
    len=p_atob(&b[0],fmt,&fmt+1);
    gPrintBoxText(&box,FontAscent,align,0,&b[0],len);
}

LOCAL_C VOID MainEventLoop(VOID)
{
    WS_EV event;

    SetFontHeight();
    for (;;)
    {
        wGetEventWait(&event);
        if (event.type==WM_REDRAW)
        {
            wValidateWin(wMainWid);
            gBorder(W_BORD_SHADOW_D|W_BORD_SHADOW_ON);
            PrintLine(0,G_TEXT_ALIGN_LEFT,"Free memory: %dkbytes",p_sgfree())>>6);
        }
        else if (event.type==WM_KEY && event.p.key.keycode==W_KEY_RETURN)
            wInvalidateWin(wMainWid);
    }
}
```

```
GLDEF_C VOID main(VOID)
{
    INT NotifierPid;
    WORD stat;

    p_setonevent(TRUE);
    wConnect(&wSpec,0,W_CONNECT_PRIORITY);
    wSystem(WSERV_FLAG_NO_NOTIFIER_REBOOT,WSERV_FLAG_NO_NOTIFIER_REBOOT);
    if ((NotifierPid=p_pidfind("SYS$NTFY.*"))>0)
    {
        p_logona(NotifierPid,&stat);
        p_terminate(NotifierPid,0);
        p_waitstat(&stat);
    }

    wSystem(WSERV_FLAG_HOOK_NOTIFIER|WSERV_FLAG_LOW_BATTERY_WARNINGS|WSERV_FLAG_HUNG_UP_SW
    ,

    WSERV_FLAG_HOOK_NOTIFIER|WSERV_FLAG_LOW_BATTERY_WARNINGS|WSERV_FLAG_HUNG_UP_SW);
    if (p_pidfind("SYS$NCP.*")<0)
        p_presume(p_execc("ROM::LINK",NULL,0));
    wMainWid=wCreateWindow(0,0,0,1);
    wsCreateClock(wMainWid,WS_CLOCK_WITH_DATE|WS_CLOCK_WITH_SECONDS,104,66,0);
    wInitialiseWindowTree(wMainWid);
    wMainGc=gCreateGC0(wMainWid);
    MainEventLoop();
}
```

To save memory, the program processes redraw messages rather than keep a backup bitmap. Pressing ENTER causes the "Free Memory" figure to be re-evaluated.

There is further information on creating a replacement shell in the *HC Programming Guide*.

PC EPOC

On the PC version, the window server looks for the shell and notifier in the following directories:

- the current directory
- the directory that contains SYS\$WSRV.IMG
- the ROM

CHAPTER 2

GENERAL WINDOW SERVER FUNCTIONS

The connection to the window server

Before you use any window server services you must connect to the window server.

How you connect to the window server depends on whether you are using the CLIB or the PLIB C startup module and what machine you are running on. See the section *Connecting to the window server* in the first chapter for a full discussion.

The CLIB C startup module opens the console device `CON:` before calling `main`. On the HC, S3 and S3a, the console device open function connects to the window server. If you try to connect a second time, your program will be panicked with panic number 100.

Provided the C startup module has not already connected the process, you may call either `wStartup` or `wConnect` to connect. The convenience function `wStartup` calls `wConnect` and then carries on to perform additional setting up that will satisfy the requirements of many applications. For a more sophisticated use of the window server, you would use `wConnect` directly.

wStartup

Connect and initialise a window

```
VOID wStartup(VOID);
```

Perform the following actions:

- Connect to the window server using `wConnect`.
- Create a window to cover the whole screen and store its ID in `wMainWid`.
- Initialise `wMainWid` so that it is visible.
- Create a permanent graphics context on the window and store its ID in `wMainGc`.

On version 3 of the window server, the window is created with a backed-up bitmap (so that no redraws are required).

The code for `wStartup` is effectively¹:

```
#include <plib.h>
#include <wlib.h>

#define NWS_HANDLE 0
#define MAIN_WIN 1

GLDEF_D WSERV_SPEC wSpec;
GLDEF_D UINT wMainGc;
GLDEF_D UINT wMainWid;
```

¹The actual code in `WLIB` is written in 8086 assembler.

```
GLDEF_C VOID wStartup(VOID)
{
    UINT field_set;
    W_WINDATA windata;

    wConnect(&wSpec, NWS_HANDLE, W_CONNECT_PRIORITY);
    field_set=0;
    if ((wSpec.conn.info.version_id&WS_VERSION_MASK)!=WS_VERSION_2)
    {
        field_set=W_WIN_BACKGROUND;
        windata.background=W_WIN_BACK_BITMAP;
    }
    wMainWid=wCreateWindow(0, field_set, &windata, MAIN_WIN);
    wInitialiseWindowTree(wMainWid);
    wMainGc=gCreateGC0(wMainWid);
}
```

The created graphics context is the current graphics context and, after calling `wStartup`, you are in a position to draw to the window. If necessary, you can reference `wSpec`, `wMainGc` and `wMainWid` by including the following declarations:

```
GLREF_D WSERV_SPEC wSpec;
GLREF_D UINT wMainGc;
GLREF_D UINT wMainWid;
```

The following example is suitable for an HC, an S3 or an S3a:

```
#include <plib.h>
#include <wlib.h>

GLREF_D UINT wMainGc;
GLREF_D WSERV_SPEC wSpec;

GLDEF_C INT main(VOID)
{
    WS_EV event;
    G_GC gc;
    P_RECT box;
    TEXT bb[32];

    wStartup();
    box.tl.x=box.tl.y=0;
    box.br=wSpec.conn.info.pixels;
    p_insrec(&box, 8, 8);
    gBorder(W_BORD_SHADOW_D|W_BORD_SHADOW_ON);
    gc.style=G_STY_BOLD|G_STY_DOUBLE;
    gSetGC(wMainGc, G_GC_MASK_STYLE, &gc);
    for (;;)
    {
        wGetEventWait(&event);
        if (event.type==WM_KEY)
        {
            p_atos(&bb[0], "Key code: %d", event.p.key.keycode);
            gPrintBoxText(&box, box.br.y>>1, G_TEXT_ALIGN_CENTRE, 0,
                        &bb[0], p_strlen(&bb[0]));

            if (event.p.key.keycode==W_KEY_RETURN)
                break;
        }
    }
    return(0);
}
```

wConnect**Connect to the window server**

```
VOID wConnect(.i.WSERV_SPEC *pwserv_spec, VOID *pnws_handle, UINT flags);
```

Connect to the window server.

The parameter `flags` should contain a combination of the following bit masks:

<code>W_CONNECT_AT_BACK</code>	Connect as a background application (the default is foreground).
<code>W_CONNECT_USER_FLAG</code>	The value of this flag is returned by a <code>wClientInfo</code> call.
<code>W_CONNECT_SYSTEM_MODAL</code>	Makes the client system modal.
<code>W_CONNECT_PRIORITY</code>	Enables the window server's process priority handling for the client. In version 4 of the window server, process priority handling can be both enabled and disabled by a suitable call to <code>wSetPriorityControl</code> .
<code>W_CONNECT_DISABLE_LEAVES</code>	If set, the window server will return negative error numbers rather than call <code>p_leave</code> . Equivalent to calling <code>wDisableLeaves(TRUE)</code> except that it also affects whether <code>wConnect</code> itself leaves or returns an error.

The parameter `pnws_handle` is a handle that the window server will use in events sent to the client that are not directed at a window (for example, key events).

The parameter `pwserv_spec` is the address of a `WSERV_SPEC` struct that must be maintained for the duration of the connection (it holds information used by `WLIB` functions). The `WSERV_SPEC` struct is typically implemented as a static variable or as allocated memory.

The `WSERV_SPEC` struct is defined as:

```
typedef struct
{
    UWORD handle_check; /* used internally */
    CONNECT_INFO conn;
    ... /* used internally */
} WSERV_SPEC;

typedef struct
{
    UWORD client_handle; /* used internally */
    W_SERVER_INFO info;
} CONNECT_INFO;
```

where `wConnect` writes information useful to the client in the `CONNECT_INFO` sub-struct `conn` (the only part of `WSERV_SPEC` that should be accessed by the client).

The `W_SERVER_INFO` struct is defined as:

```
typedef struct
{
    P_POINT pixels; /* display size */
    UWORD width_1000_pixels_mm; /* width of 1000 pixels in mm */
    UWORD height_1000_pixels_mm;
    UBYTE set_is_dark; /* TRUE if set pixels are dark */
    UBYTE version_id; /* machine type and window server version */
    UWORD system_font_handle /* ID of default font */
    .... /* extra space for future expansion */
} W_SERVER_INFO;
```

where:

<code>pixels</code>	the size of the screen in pixels (<code>pixels.x</code> wide by <code>pixels.y</code> high).
<code>width_1000_pixels_mm</code>	the width and height (in millimetres) of 1000 screen pixels for applications that wish to draw objects of a certain physical size or to correct for the pixel aspect ratio. Note that, for the Series 3 (but not for other machines, including the Series 3a) these two values are not reliable.
<code>height_1000_pixels_mm</code>	
<code>set_is_dark</code>	TRUE if a set bit appears dark on the display (as on an LCD display) and FALSE otherwise (as for a CRT display). An application can invert drawings according to this flag so that they appear the same on both types of display. (Not reliable on a PC version of the window server.)

version_id	the machine type and window server version number. version_id WS_TYPE_MASK is one of WS_TYPE_MC, WS_TYPE_HC, WS_TYPE_S3, WS_TYPE_S3A or WS_TYPE_S3C depending on whether the machine is an MC, HC, S3, S3a or <i>Workabout</i> . The value of version_id WS_VERSION_MASK is WS_VERSION_2, WS_VERSION_3 or WS_VERSION_4 depending on whether a connection was made to version 2, version 3 or version 4 of the window server. In this context, version 3.5 is grouped with version 3.
system_font_id	the ID of the default font that you get when you create a graphics context. (Alternatively, you can use WS_FONT_SYSTEM to specify the system font.)

Note that any application running on the S3a in S3 compatibility mode will find that the version_id is set to WS_TYPE_S3A|WS_VERSION_4. In a similar situation, an application running on the *Workabout* will have version_id set to WS_TYPE_S3C|WS_VERSION_4. Therefore, the window server is not providing a completely identical interface to such applications.

Whether wConnect was called directly or indirectly, the address of the WSERV_SPEC variable which was passed is recorded in the reserved static wserv_channel. This can be used in general purpose code to obtain the above information. For example:

```
GLREF_D WSERV_SPEC *wserv_channel;

LOCAL_D P_POINT ScreenSize;

ScreenSize=wserv_channel->conn.info.pixels;
```

The following example program (which requires the PLIB C startup module) illustrates the use of wConnect to connect to the window server.

```
#include <plib.h>
#include <wlib.h>

#define WBORDER 8

GLDEF_D WSERV_SPEC wspec;
GLDEF_D UINT wid;
GLDEF_D WMSG_KEY key;

GLDEF_C VOID CreateWindow(VOID)
{
    UINT border;
    W_WINDATA windata;

    windata.flags=W_WIN_NO_REDRAW;
    windata.background=W_WIN_BACK_SET;

    border=wCreateWindow(0,W_WIN_NO_REDRAW|W_WIN_BACKGROUND,&windata,2);
    windata.extent.tl.x=WBORDER;
    windata.extent.tl.y=WBORDER;
    windata.extent.width=wspec.conn.info.pixels.x-(2*WBORDER);
    windata.extent.height=wspec.conn.info.pixels.y-(2*WBORDER);
    windata.background=W_WIN_BACK_NONE;
    wid=wCreateWindow(border,W_WIN_EXTENT|W_WIN_BACKGROUND,&windata,1);
    wInitialiseWindowTree(border);
}

GLDEF_C INT main(VOID)
{
    WS_EV event;
    G_GC gc;
    P_RECT box;
    TEXT bb[32];

    wConnect(&wspec,0,W_CONNECT_PRIORITY);
    CreateWindow();
    box.tl.x=box.tl.y=0;
    box.br.x=wspec.conn.info.pixels.x-(2*WBORDER);
    box.br.y=wspec.conn.info.pixels.y-(2*WBORDER);
```

```

    for (;;)
    {
        wGetEventWait(&event);
        if (event.type==WM_REDRAW)
        {
            gc.style=G_STY_BOLD|G_STY_DOUBLE;
            wBeginRedrawWinGC(wid,G_GC_MASK_STYLE,&gc);
            p_atos(&bb[0],"Key code: %d",key.keycode);

            gPrintBoxText(&box,box.br.y>>1,G_TEXT_ALIGN_CENTRE,0,&bb[0],p_slen(&bb[0]));
            wEndRedraw();
        }
        if (event.type==WM_KEY)
        {
            key=event.p.key;
            if (key.keycode==W_KEY_RETURN)
                break;
            wInvalidateWin(wid);
        }
    }
    return(0);
}

```

The function `createwindow` sets up a two-window parent-child system where the parent implements a thick border and all drawing is done to the child window. The main function contains an event loop that handles redraw events and key events to display the code of the last key pressed. Note the use of `wInvalidateWin` to update the screen on receipt of a key event by forcing a redraw.

The true screen and pixel dimensions of the various LCD screens are as follows:

Machine type	Screen (pixels)	Pixel pitch (mm)	Pixel size (mm)	Screen size (cm)	Screen size (in)
HC	160x80	0.34x0.43	0.31x0.40	5.44x3.44	2.14x1.35
S3	240x80	0.385x0.43	0.355x0.40	9.24x3.44	3.64x1.35
<i>Workabout</i>	240x100	0.26x0.30	0.23x0.27	6.24x3.00	2.45x1.18
S3a	480x160	0.259x0.259	0.20x0.20	12.477x4.156	4.915x1.637
MC200	640x200	0.33x0.33	0.30x0.30	21.12x6.60	8.31x2.60
MC400	640x400	0.33x0.33	0.30x0.30	21.12x13.20	8.31x5.20

In the above table, the horizontal measure is shown before the vertical measure. The pixel pitch measures the horizontal and vertical distance between the same points on adjacent pixels. The difference between the pixel size and the pixel pitch gives the gap between pixels.

wDisconnect

Disconnect from the window server

```
VOID wDisconnect(VOID);
```

Disconnect from the window server and free resources within the client process and within the window server.

A client is automatically disconnected if it terminates.

wFlush

Flush buffered commands

```
VOID wFlush(VOID);
```

Flush any contents of the client-side buffer. This will ensure that the window server has received and executed all previous functions.

Note that `wFlush` does not report any errors that occur in the processing of the client-side buffer. The function `wCheckPoint` (described below) flushes the buffer and reports errors.

The client-side buffer is automatically flushed when:

- the buffer is about to overflow
- the client calls a function that returns a value that requires the window server process to run (for example `wCreateWindow` returns the ID of the window it creates). The functions `gTextwidth`, `gTextCount`, `gFontInfo` and `wCheckBitmapid` do not flush the client-side buffer because they are implemented by code that runs in the client's process.
- the client calls `wGetEventWait`, `wGetEvent` or `wGetEventSpecial`

Most applications don't need to call `wFlush` and calling `wFlush` unnecessarily will degrade performance.

Programs that perform animation or that respond to an event source other than the window server (such as a serial I/O device) may need to use `wFlush`. For example, the following code, intended to produce some animation:

```
gClrRect(prect, G_TRMODE_INV); /* invert a rectangle */
p_sleep(5L); /* pause for half a second */
gClrRect(prect, G_TRMODE_INV); /* invert it back again */
```

does not have the intended effect. The code should be as follows:

```
gClrRect(prect, G_TRMODE_INV);
wFlush();
p_sleep(5L);
gClrRect(prect, G_TRMODE_INV);
```

When debugging a program, it can be useful to insert calls to `wFlush` (which are removed subsequently) to force the screen to be updated.

Series 3 compatibility modes

Both the Series 3a and the *Workabout* can be set to operate in Series 3 compatibility mode. The motivation for this is to be able to emulate Series 3 graphics, so that unmodified Series 3 applications can run on either machine. Naturally, grey is not available to any application running in Series 3 compatibility mode.

On the Series 3a, the compatibility mode is implemented by allowing all Window Server graphics commands to draw with double size pixels. Because the Series 3a's screen has 480 x 160 pixels, compared to the Series 3's 240 x 80 pixels, doubling up the pixels on the Series 3a screen gives the 'look' and 'feel' of the Series 3 screen for Series 3 applications running on the Series 3a.

On the *Workabout*, with its 240 x 100 screen, full Series 3 compatibility is implemented by restricting drawing to a 240 x 80 region, centred on the screen, leaving ten rows of pixels unused at both the top and bottom of the screen.

The *Workabout* has a second compatibility mode that, while not allowing the use of grey, allows an application to draw to the full 240 x 100 extent of the screen. This mode can only be used with a Series 3 application that is written in such a way that it can adjust the sizes of its windows according to the screen dimensions of the machine on which it is running. It is recommended that this mode be used only if the appearance of a Series 3 application running in full Series 3 compatibility mode on the *Workabout* is truly unacceptable.

Drawing with double size pixels is a feature that is available in version 4 of the window server. As well as being used for compatibility mode on the Series 3a, it can be set for individual windows; see the *Windows* chapter for further information.

The following two functions relate to compatibility mode.

wCompatibilityMode **Set or cancel compatibility mode**

```
VOID wCompatibilityMode(UINT flags, .i.WSERV_SPEC *pwspec);
```

On the S3a and *Workabout*, full S3 compatibility mode is turned on by setting `flags` to `W_CTBY_S3`.

On the *Workabout*, the S3 compatibility mode that allows use of the full 240 x 100 extent of the screen is turned on by setting `flags` to `W_CTBY_S3_SCR`.

On both machines, Series 3 compatibility is turned off by setting `flags` to zero.

The `pwspec` parameter *must* point to the same `WSERV_SPEC` structure that was passed to the `wConnect` function (see earlier in this chapter). `wCompatibilityMode` modifies information such as the screen dimensions, held in this structure.

wInquireCompatibility**Inquire state of compatibility flags**

```
UINT wInquireCompatibility(VOID);
```

The function returns the current state of the compatibility flags of the calling client. The flags are the same as set by the function `wCompatibilityMode`.

Error handling**wCleanUp****Return to defined state**

```
VOID wCleanUP(VOID);
```

Put the window server back into a defined state by:

- freeing the temporary Graphics Context if it exists
- ending the redraw if one was in progress

If there is a current graphics context that is attached to a window, `wCleanUp` also invalidates that window - so that it is not left in a partly drawn state.

The function `wCleanUp` is typically called in response to a `p_leave`.

wCheckPoint**Check for an error**

```
INT wCheckPoint(VOID);
```

Flush the client-side buffer (as for `wFlush`) and return zero if there was no error.

If there is an uncleared error or if an error occurred in the processing of the buffer, call `p_leave(err)` or return `err`, depending on whether `wDisableLeaves` has been called, where `err` is the negative error number.

wDisableLeaves**Disable/enable leaves**

```
UINT wDisableLeaves(UINT flag);
```

If `flag` is `TRUE`, the window server functions will (for the calling client) return an error code rather than call `p_leave` when they encounter an error. If `flag` is `FALSE`, enable `p_leaves`.

Returns the previous value of `flag` (not before version 3.5).

By default, the window server functions that can fail call `p_leave` when an error occurs.

Not available in version 2 of the window server.

Priority changing

The window server will not change a client's process priority unless the client has priority control enabled.

Priority control is enabled if the client sets the `w_CONNECT_PRIORITY` flag when it connects to the window server or, if running version 4 of the window server, the client calls `wSetPriorityControl(TRUE)`.

Priorities are set as follows:

- when a client loses the foreground or calls `wStartCompute`, its priority is set to `E_PRIORITY_BACK`
- when a client gains the foreground and is not in compute mode, its priority is set to the higher priority `E_PRIORITY_FORE`

A client with priority control enabled should not change its own priority.

wSetPriorityControl **Set process priority handling on or off**

```
INT wSetPriorityControl(UINT state);
```

Introduced in version 4 of the window server, this function enables and disables process priority handling for a client. Setting state to TRUE enables it, while setting state to FALSE disables it.

The function always returns 0.

wStartCompute **Enter compute mode**

```
VOID wStartCompute(VOID);
```

Mark the client as being in compute mode, setting the caller's process priority to E_PRIORITY_BACK regardless of whether it has the foreground or not.

Should be called before performing a computationally intensive task.

Has no effect unless the client has priority control enabled.

wEndCompute **Leave compute mode**

```
VOID wEndCompute(VOID);
```

The caller is marked as not being in compute mode.

Its priority will be set to E_PRIORITY_FORE whenever it is foreground.

Has no effect unless the client has priority control enabled.

General client functions

wClientInfo **Get information about a client**

```
INT wClientInfo(UINT pid);
```

Return a word mask giving information about the window server client with process ID pid.

If pid is a client of the window server, the function returns a bit mask that contains the following bit fields:

W_CONNECT_CONNECTED	this is set
W_CONNECT_USER_FLAG	if set, the W_CONNECT_USER_FLAG was specified at connect time
W_CONNECT_SYSTEM_MODAL	if set, the W_CONNECT_SYSTEM_MODAL flag was specified at connect time or the client is in a system modal state as a result of calling wSystemModal
W_CONNECT_PRIORITY	if set, priority control is enabled.

If pid is not a client of the window server, the function calls p_leave(E_FILE_NXIST) or returns E_FILE_NXIST, depending on whether wDisableLeaves has been called.

On version 2 of the window server, the function returns zero if pid is not a client of the window server.

wClientPosition **Position client in task order**

```
VOID wClientPosition(UINT pos, UINT pid);
```

Position client pid to position pos in the task order, zero being at the front and any value greater than the number of connected tasks being at the back.

The constant WS_LAST_CLIENT_POSITION is provided to position a client at the back.

Passing a pid of zero is equivalent to passing the pid of the caller.

On a large screen version of the window server such as the MC, if the client pid is marked as iconised and it is positioned to the front by a call to wClientPosition, it will be sent a WM_DEICONISE event.

wClientIconised**Mark client as iconised**

```
VOID wClientIconised(UINT state);
```

Mark the caller as iconised if state is TRUE, otherwise mark it as deiconised.

This function only applies to large screen version of the window server such as the MC (and is not available on hand-held machines such as the HC, S3, S3a and *Workabout*).

On an MC, if the user holds down the CONTROL key while pressing the TASK key, iconised task are skipped and only non-iconised tasks are selected.

If a client is marked as iconised, the window server generates a WM_DEICONISE event to client pid if wClientPosition(pid,0) is called (normally by another client) to make client pid the foreground client. The WM_DEICONISE event would normally prompt the client to deiconise itself.

On the MC, the window server recognises the shell (with process name *sys\$shell*) and sends it a WM_DEICONISE event when the PSION+TASK key is pressed while the shell is iconised.

wSystemModal**Make client system modal**

```
VOID wSystemModal(UINT pos);
```

Make the caller system modal and place it at position pos in the task list.

The window server limits task switching to only those processes that have a lower client position than the frontmost system modal task. If there are no clients with a lower client position, the system modal task is locked into the foreground.

When wSystemModal is used, pos is commonly zero - to lock the client to the foreground.

Not available in version 2 of the window server.

It is important to note that calling this function does not prevent a task from being made foreground. For example, a user pressing ENTER on a task in the system screen will cause that task to be made foreground.

To handle an attempt to bring an application into foreground, it must test for a WM_FOREGROUND event. In response to this event, the application can call wClientPosition to return itself to background.

wCancelSystemModal**Cancel system modal state**

```
VOID wCancelSystemModal(UINT pos);
```

Cancel the caller's system modal state and place it at position pos in the task list.

Not available in version 2 of the window server.

wEnablePauseKey**Enable pause key**

```
VOID wEnablePauseKey(VOID)
```

Allow the user to pause the calling client's graphics output when it has the foreground.

Useful, for example, to stop information scrolling off the top of a display.

On the HC, the pause key is PSION+LEFT-ARROW and on the S3, S3a, *Workabout* and MC, it is CTRL+S. The user resumes the client by pressing any key. The key press that resumes the client is not delivered to the client.

The pause key may be disabled by calling wDisablePauseKey.

The pause key is disabled by default, except in Console applications, where it is enabled by default.

When the user presses the pause key, the client will be stalled within a window server function and can not therefore process any events that occur in the meantime. Applications that redraw their windows or that respond to events other than the window server (such as the receipt of data from the serial port) should not enable the pause key.

Not available in version 2 of the window server.

wDisablePauseKey**Disable pause key**

```
VOID wDisablePauseKey(VOID);
```

Disable pause key processing for the calling client.

The pause key is enabled by calling `wEnablePauseKey`.

Not available in version 2 of the window server.

wGetProcessList**Get client list**

```
VOID wGetProcessList(UWORD *pbuf);
```

Write the process IDs of the clients of the window server as a zero terminated list in front to back order to `pbuf`.

There should be at least `WS_MAX_CLIENTS+1` words of memory at `pbuf`.

Not available in versions prior to version 3.5 of the window server.

Screen-based output

When using the window server, graphics output can be directed at:

- a graphics context (as described in the *Graphics Output* chapter)
- a particular window (as described in the *Windows* chapter)
- the screen as a whole (as described next)

On the HC, the font used for output that is not graphics context directed is determined by the `$WS_IF` ("Internal Font") environment variable.

This should contain a `WORD` binary value of 0 for `WS_FONT_BASE` and 1 for `WS_FONT_BASE+1` and so on. If you change the value of `$WS_IF`, you must reset the machine by pressing the recessed reset button to effect the change.

The "factory" setting of `$WS_IF` is 4 (which selects the S3 font).

In version 4 of the window server, the environment variable `$WS_FNTS` is used to contain the indices of the fonts to be used by the window server for notifies, clocks and so on.

In order, they are:

- System font
- Notifier/Alert font
- Status Window font
- Symbols font used for the status window diamond symbol
- Medium 2 digital clock font
- Medium 2 date font
- Notifier/alert button font
- Small status window clock font

wInfoMsgCorner**Present an information message**

```
INT wInfoMsgCorner(TEXT *pmsg,UINT corner);
```

Displays the zero terminated string pmsg for 2 to 2.5 seconds or until cancelled.

The message is displayed in one of the four corners of the screen depending on corner, which should be one of:

W_CORNER_TOP_LEFT	to display pmsg in the top left corner
W_CORNER_TOP_RIGHT	to display pmsg in the top right corner
W_CORNER_BOTTOM_LEFT	to display pmsg in the bottom left corner
W_CORNER_BOTTOM_RIGHT	to display pmsg in the bottom right corner

The length of pmsg (excluding its zero terminator) should be less than or equal to W_INFO_MSG_MAX_LEN (64) bytes. A longer pmsg is truncated.

The message display is cancelled when:

- the calling client loses the foreground (the message is never displayed if the caller is a background client)
- wInfoMsgCorner or wInfoMsg is called again
- the message is explicitly cancelled by calling wInfoMsg(NULL) or wInfoMsg("")

The function behaves as for wCheckPoint in that it flushes the client-side buffer and reports any uncleared error - either by calling p_leave or by returning the error number.

wInfoMsg**Present an information message**

```
INT wInfoMsg(TEXT *pmsg);
```

Displays the zero terminated string pmsg in the bottom right hand corner of the screen for 2 to 2.5 seconds or until cancelled.

Behaves as for wInfoMsgCorner(&pmsg,W_CORNER_BOTTOM_RIGHT).

wSetBusyMsg**Present a flashing busy message**

```
INT wSetBusyMsg(TEXT *pmsg,UINT corner_delay);
```

Displays the zero terminated string pmsg as a flashing "busy" message in the specified corner of the screen.

The message continues to display whenever the caller has the foreground.

The message is cancelled by calling wCancelBusyMsg, wSetBusyMsg(NULL) or wSetBusyMsg("").

The parameter corner_delay specifies both the corner of the screen in which the message will appear and a delay to stop the message from appearing instantly. The delay is used to stop the message from appearing at all when there is the possibility that the task can be completed in a short time.

The time delay should be ored in with the corner mask, which should be one of:

W_CORNER_TOP_LEFT	to display pmsg in the top left corner
W_CORNER_TOP_RIGHT	to display pmsg in the top right corner
W_CORNER_BOTTOM_LEFT	to display pmsg in the bottom left corner
W_CORNER_BOTTOM_RIGHT	to display pmsg in the bottom right corner

The delay is specified in half seconds. For example:

```
wSetBusyMsg("Saving",W_CORNER_TOP_LEFT|6);
```

will display the message in the top left corner after 3 seconds if it has not been cancelled before the time is up.

The delay can range from 0 to 63 half seconds, inclusive.

The length of `pmsg` (excluding its zero terminator) should be less than or equal to `W_BUSY_MSG_MAX_LEN` (20) bytes. A longer `pmsg` is truncated.

The function behaves as for `wCheckPoint` in that it flushes the client-side buffer and reports any uncleared error - either by calling `p_leave` or by returning the error number.

While the window server is loading a large bitmap or font (`gOpenBit` or `gOpenFont`) or saving a large bitmap (`gSaveBit`), it doesn't maintain the busy message. In these cases, the busy message will not flash and it might not even appear.

wCancelBusyMsg **Cancel a flashing busy message**

```
INT wCancelBusyMsg(VOID);
```

Cancel a busy message.

Entirely equivalent to `wSetBusyMsg(NULL)`.

Alerts

The functions that support alerts are available on S3, S3a and *Workabout* machines - and on HC machines that are running version 3.5 or later of the window server.

Alerts present a `p_notify`-like display where the user is presented with a message and prompted to respond by pressing a button. Unless you are already familiar with the notifier services, you may find it useful to read the *Notifier Services* section of the *Error Handling* chapter in the *PLIB Reference* manual.

Alerts extend the specification of `p_notify` as follows:

- the maximum number of message lines is increased from 2 to 3. In version 4 of the window server, the maximum number is increased to 4 (provided the screen is large enough to display four lines of text in an alert)
- message lines may be centred or placed at a specified horizontal position
- rather than specify the address of a text string, it is possible to specify built in text strings by number (the same text strings that are obtained using `p_gettext`)
- an asynchronous function is also provided so that the calling program can perform other tasks while waiting for the user to respond

The alert functions are:

`wsAlertw` presents the user with a message and waits for a response. This function is similar in effect to `p_notify`.

`wsAlerta` is the asynchronous form of `wsAlertw`. Using this, the program can perform other tasks while waiting for the user to respond. There is no asynchronous form of `p_notify`.

`wsAlertUpdate` is used to update a pending alert (which was launched using `wsAlerta`).

For example, the program:

```
#include <plib.h>
#include <wlib.h>

LOCAL_D WSERV_SPEC wSpec;

GLDEF_C INT main(VOID)
{
    wConnect(&wSpec, 0, W_CONNECT_PRIORITY);
    wsAlertw(WA_ALERT_CLIENT, "Hello World", NULL, NULL);
    return(0);
}
```

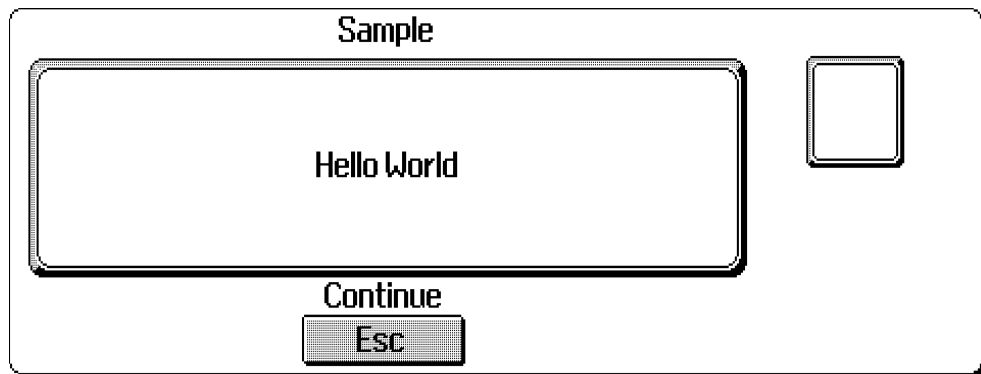
when compiled and linked to produce a program with the name *sample.img*, presents the following display on the HC.



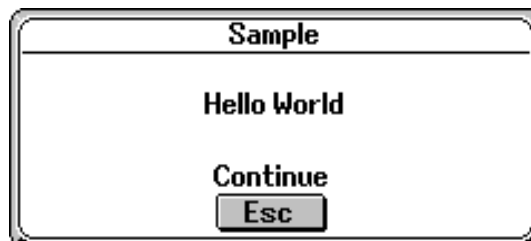
From top to bottom, the display consists of 3 parts:

- Title** Displays the program name. If the reserved static `DatStatusNamePtr` contains other than `NULL`, it is assumed to point to a zero terminated string which is taken as the program name (of up to 8 characters in length and stopping when a '.' is reached). If `DatStatusNamePtr` is `NULL` (which it will be if not explicitly set), the process name is used. The process name is normally the name of the executable and, for a single source file program, the name of the executable is normally the name of the source file so you can deduce that the above example had the file name *sample.c*. On the S3, the title area also contains the date and time.
- Message** The message area contains up to 3 lines of text.
- Buttons** The button area contains one, two or three buttons.

On the S3a, running version 4 of the window server, the display is slightly different. Using the above code results in the following:



On the *Workabout*, the appearance is as follows:



If there is a single button, it is activated by ESC. With two buttons, the left button is activated by ESC and the right button is activated by ENTER. With three buttons, the buttons are activated by, from left to right, ESC, SPACE and ENTER.

Version 3.5 of the window server added the ability to provide the notifier services (accessed via `p_notify` and `p_notifyerr`). In EPOC terminology, the window server is said to "hook the notifier".

On the S3, S3a and *Workabout*, the window server *always* hooks the notifier.

On an HC, version 3.5 the window server does not by default hook the notifier (for backward compatibility with version 3 of the window server). However, an HC may be configured such that the window server does hook the notifier - as described in the section *System start-up* in the first chapter.

If you remove the first parameter from `wsAlertw` or the first two parameters from `wsAlertA`, the remaining parameters correspond to the 5 parameters to `p_notify`.

If the window server has hooked the notifier, the program:

```
#include <plib.h>

GLDEF_C INT main(VOID)
{
    p_notify("Hello World", NULL, NULL, NULL, NULL);
    return(0);
}
```

produces the same result as the above example using `wsAlertw` - visually at least (and assuming that the program is still called *sample.img*).

However, there are differences between `wsAlertw` and `p_notify`:

- The alert presented by `p_notify` is system modal - the user can't task away from it. In contrast, `wsAlertw` is not system modal. For example, if a program reports a "No system memory" error using `wsAlertw`, it can reasonably include a "Retry" option because it is possible to task to another process and release memory (say by exiting a task) before returning to the alert and selecting the Retry button.
- The caller of `p_notify` need not be a client of the window server.
- When `p_notify` is called, the alert is presented regardless of whether the calling client is foreground or not (although calling `p_notify` does not displace the foreground client). If a background client calls `wsAlertw`, the alert is not drawn until that task is made foreground. In some circumstances, it may be desirable to call `wClientPosition(0,0)` and then `wFlush` to make the caller foreground before calling `wsAlertw`.
- As well as having an extra leading parameter, `wsAlertw` has a stack-based calling convention which is prototyped in such a way that unnecessary trailing NULLs may be omitted whereas `p_notify` uses a register calling convention that requires all 5 parameters to be present.

The similarities between `p_notify` and `wsAlertw` are:

- In terms of setting up the display, all the features of `wsAlertw` are also available via `p_notify` and vice versa. (Unfortunately, this means that the additional parameters associated with the increased functionality have been squeezed into the existing `p_notify` compatible parameters in a somewhat inelegant way.)
- They are both designed not to fail when there is no free system memory. Both are ideal for reliably reporting errors - including a "No System Memory" error.

The extra parameters are provided by passing data structures that are differentiated from a zero terminated string by a leading zero. It follows that zero length strings should not be used as parameters to any of the alert-based functions (they should be converted to NULLs).

Calling an alert-based function does *not* flush the client-side buffer.

On the HC, the font used to present alerts is determined by the `$WS_IF` environment variable - as described earlier.

wsAlertW **Present and wait for an alert**

```
INT wsAlertw(INT mode, TEXT *pT1, TEXT *pT2, TEXT *p01, TEXT *p02, TEXT *p03);
```

Present a `p_notify`-like display where the user is presented with a message and prompted to respond by pressing a button. As with `p_notify`, the function waits for the user to respond and returns the index (in the range 0 to 2) of the button pressed.

When called from a regular application, the `mode` parameter should be `WS_ALERT_CLIENT`. Other values can only be used by a special "alarm server" client. (The alarm server is a system component on the S3 and S3a.)

Except for the additional `mode` parameter and except for the behavioural differences noted above, this function provides the same services as `p_notify` - as described in the *PLIB Reference* manual.

The following describes only the extensions to the functionality normally provided by `p_notify`.

Note that `wsAlertw` is actually prototyped as:

```
INT CDECL wsAlertw(INT, TEXT *, TEXT *, TEXT *, ...);
```

so that you can leave out trailing NULLs when it is appropriate to do so. The compiler will complain if you leave out trailing NULLs in a call to `p_notify` (because `p_notify` uses a register calling convention which does not permit a variable number of parameters).

Access to built in text

You can access operating system text (such as an error message) by passing a 3 byte array in place of a text string to any of the 5 text parameters. The contents of the array should contain:

byte 0	zero
byte 1	0xfe (0376 in octal)
byte 2	the signed index of the operating system text as passed to <code>p_gettext</code> .

For example, the following program, again compiled and linked as *sample.img*:

```
#include <plib.h>
#include <wlib.h>

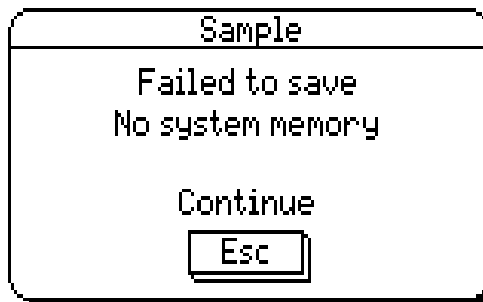
LOCAL_D WSERV_SPEC wSpec;

LOCAL_C VOID AlertErr(INT err, TEXT *msg)
{
    TEXT bb[3];

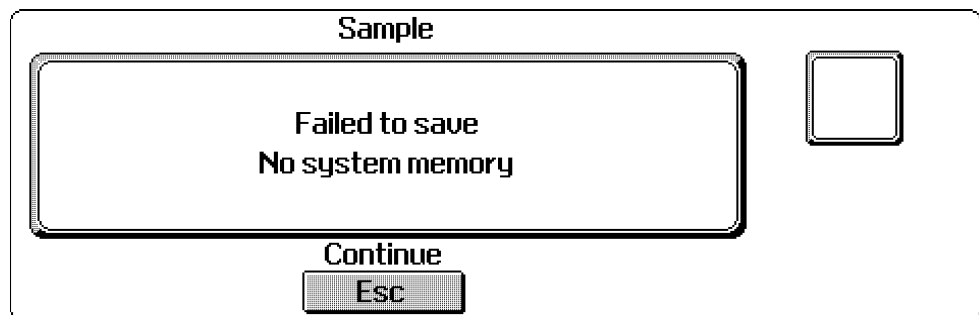
    bb[0]=0;
    bb[1]=0xfe;
    bb[2]=err;
    wsAlertw(WS_ALERT_CLIENT, msg, &bb[0], NULL);
}

GLDEF_C INT main(VOID)
{
    wConnect(&wSpec, 0, W_CONNECT_PRIORITY);
    AlertErr(E_GEN_NOMEMORY, "Failed to save");
    return(0);
}
```

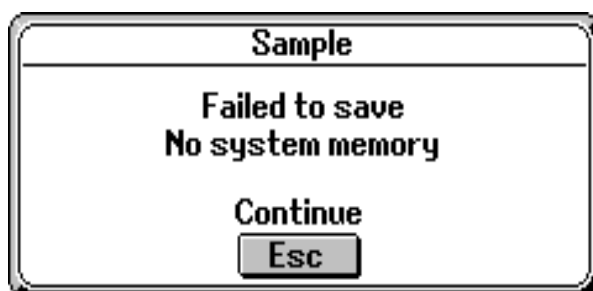
when run on the HC, displays:



when run on the S3a under version 4 of the window server, displays:



and when run on the *Workabout* displays:



Formatted text with 3 message lines

If the first two bytes at `pt1` are zero, `pt2` is ignored and `wsAlertw` assumes that the two zero bytes are immediately followed by:

- an array of three `DESC` structs
- immediately followed by a character buffer of maximum length `W_ALERT_TEXT_MAX_LEN` (80) that contains the text for the three lines

The struct `DESC` is defined in *wlib.h* as:

```
typedef struct
{
    UBYTE hposition;
    UBYTE length;
    UWORD offset;
} DESC;
```

where

`hposition` is either `0xff` for centred text or any other value to specify the pixel position from the left of the alert

`length` is the length of the text for the line, to be taken from the buffer

`offset` is the offset of the start of the text relative to `pt1`

Such a data structure would normally be built up by a function as in, for example:

```
LOCAL_C VOID CDECL Alert3(TEXT *m1, TEXT *m2, TEXT *m3)
{
    TEXT *pt;
    TEXT **pps;
    DESC *pd, *pdend;

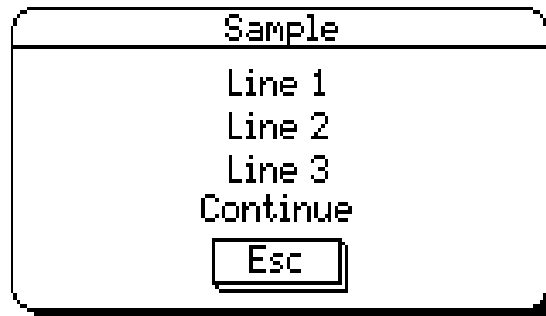
    struct {
        WORD zero;
        DESC line[3];
        TEXT buf[W_ALERT_TEXT_MAX_LEN];
    } al;

    al.zero=0;
    pt=&al.buf[0];
    pps=&m1;
    for (pd=&al.line[0], pdend=pd+3; pd<pdend; pd++)
    {
        pd->hposition=0xff;
        pd->length=p_slen(*pps);
        pd->offset=pt-(TEXT *)&al;
        pt=(TEXT *)p_bcpy(pt, *pps++, pd->length);
    }
    wsAlertw(WS_ALERT_CLIENT, (TEXT *)&al, NULL, NULL);
}
```

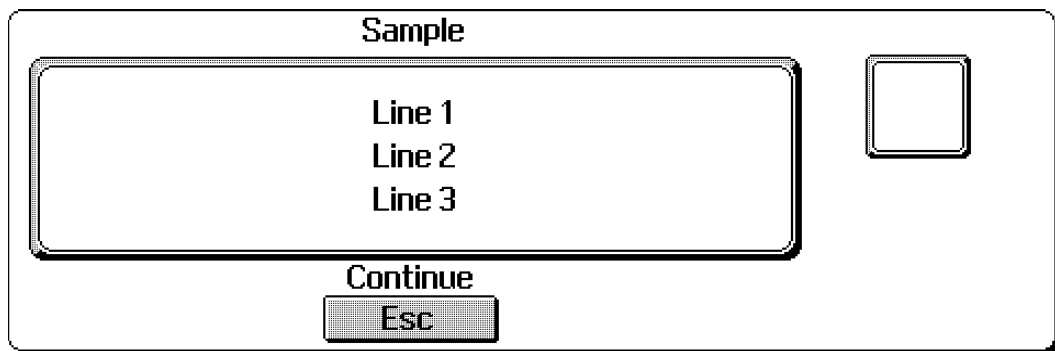
where the following line:

```
Alert3("Line 1","Line 2","Line 3");
```

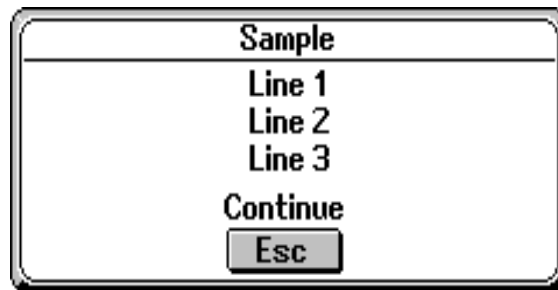
when executed on an HC, displays:



when executed on the S3a under version 4 of the window server, displays:



and when executed on the Workabout, displays:



Text with 4 message lines

This is possible in version 4 of the window server and is achieved by ORing the `WS_ALERT_B` attribute into the mode parameter. The interpretation of the parameters `pT1` and `pT2` is changed.

The text referenced by the parameter `pT1` is used as the title and is placed above the main box. This contrasts with the normal practice of `wsAlertw` in using the program name as the title.

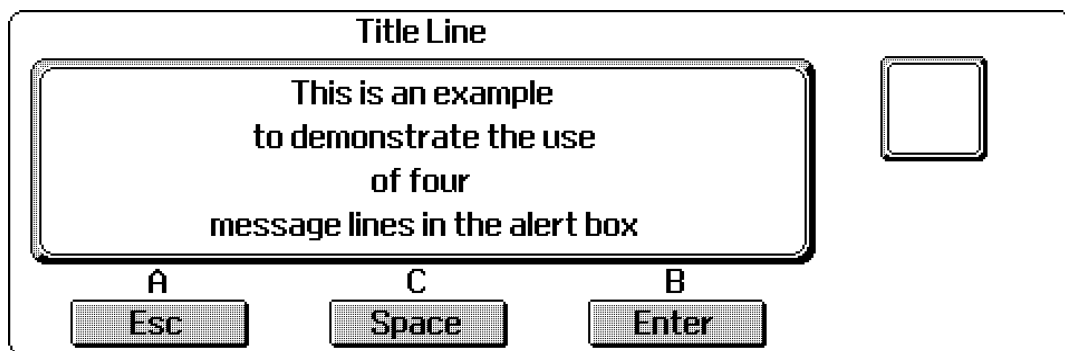
The text referenced by the parameter `pT2` is a single string but can include up to three carriage return characters (`0x13` or `'\r'` in C programs). Each carriage return character causes the remaining text to be wrapped to a new line and each line is centred within the main display box.

Note that this text need not contain carriage return characters. If the text is too long to fit onto one line, carriage returns will be inserted at appropriate points. Whole words, however, will not be split.

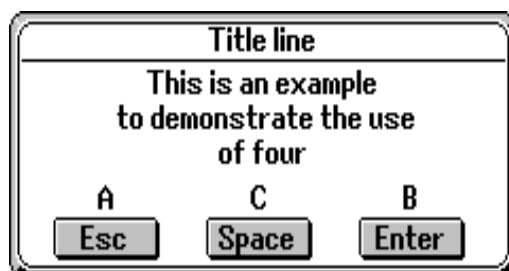
The sample code fragment below illustrates how this can be done. Note also the use of three buttons in this example:

```
...
TEXT *ptxt, text[108];
...
ptxt = p_scpy(&text[0], "This is an example \r to demo");
ptxt = p_scpy(ptxt, "nstrate the use \r");
ptxt = p_scpy(ptxt, "of four\r");
ptxt = p_scpy(ptxt, " message lines in the alert box");
...
wsAlertW(WS_ALERT_CLIENT|WS_ALERT_B, "Title Line", &text[0], "A", "B", "C");
```

This results in the following alert when run on a Series 3a machine under version 4 of the window server:



Although the *Workabout* uses version 4, its screen is not large enough to display four lines of text in an alert. If the above code is run on *Workabout*, the fourth line is not displayed and the appearance of the alert is as follows:



Note that the techniques used to access built in text and formatted text with three message lines as described earlier, *cannot* be used with the attribute `WS_ALERT_B` set.

If any illegal parameters are passed, the function will raise a `W_PANIC_ALERT` panic.

wsAlertA **Present an alert**

```
VOID wsAlertA(INT mode, WORD *pstat, TEXT *pt1, TEXT *pt2, TEXT *pb1, ...);
```

This function is not suitable for general use in applications. It is intended to be used only by the process designated to be the alarm server; any application, however, may use `wsAlertW`.

Presents a `p_notify`-like display where the user is presented with a message and prompted to respond by pressing a button.

Functionally identical to `wsAlertW` except that it returns immediately without waiting for the user to respond. It is the asynchronous form of `wsAlertW`.

Asynchronous requests are described in the chapter *Asynchronous Requests and Semaphores* in the *PLIB Reference* manual.

When the user does respond, the calling process I/O semaphore is signalled and the index of the button pressed (0, 1 or 2) is written to `*pstat`.

Once launched, there is no way of cancelling an asynchronous alert - it can only be completed by the user.

If any illegal parameters are passed, the function will raise a `W_PANIC_ALERT` panic.

wsAlertUpdate**Update a pending alert**

```
INT wsAlertUpdate(TEXT *pt1, TEXT *pt2, TEXT *pb1, ...);
```

This function is not suitable for general use in applications. It is intended to be used only by the process designated to be the alarm server; any application, however, may use `wsAlertw`.

Update an asynchronous alert where the parameters `pt1`, etc are as for `wsAlertA`.

Does nothing if the user has already responded to the alert.

If any illegal parameters are passed, the function will raise a `W_PANIC_ALERT` panic.

Status windows

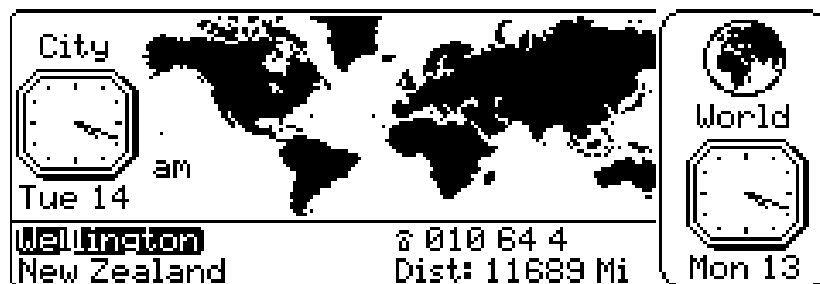
Status windows are part of the S3, S3a and *Workabout* user interfaces.

In principle, status windows are also supported on an HC that is running version 3.5 of the window server. However, their use requires the cooperation of a client which has declared itself as the application key handler by calling `wAppKeyHandler` (the application key handler is the shell on the S3, S3a and *Workabout*). In practice, it would be difficult for an external developer to set up status windows on the HC.

The window server supports two kinds of status window:

temporary	If temporary status windows are enabled, the window server displays a pop-up transient status window in front of the foreground client's existing windows when <code>PSION+MENU</code> is pressed. The status window remains for 2 to 2.5 seconds.
permanent	While enabled, the window server maintains a permanent status window to the right of the screen and behind existing windows. If an application supports a permanent status window, it is meant to "tile" its main top-level window with the status window. On the S3 and S3a, higher level software toggles the presentation of a permanent status window in response to a <code>CTRL+MENU</code> press.

The following shows the S3 World application's display with a temporary status window to the right of the screen using a version prior to version 4 of the window server:



There is no difference between the appearance of a permanent and a temporary status window.

The status window gives the user a view of (from top to bottom):

a program icon	A client's icon is determined by a structure pointed to by the application key handler.
a program name	If the reserved static <code>DatStatusNamePtr</code> contains other than <code>NULL</code> , it is assumed to point to a zero terminated string that gives the program name (of up to 8 characters, terminated by any '.'). If <code>DatStatusNamePtr</code> is <code>NULL</code> , the process name is used.
time and date	The time and data is presented following the information in the <code>E_CONFIG</code> struct as obtained by calling <code>p_getctd</code> .

The following shows the S3a World application's display with a temporary status window to the right of the screen using version 4 of the window server:



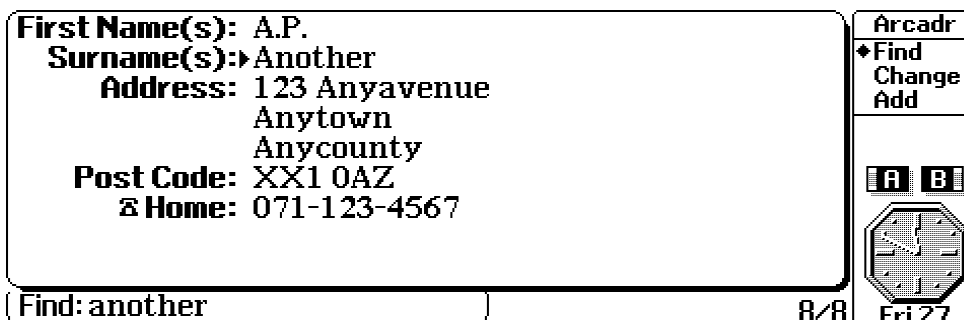
Note that the S3a has a larger and finer grained screen (480 x 160) pixels).

In version 4 of the window server, the status window has been modified. The program name and the program icon have changed places and four new features can be displayed (although not all are shown in the above example):

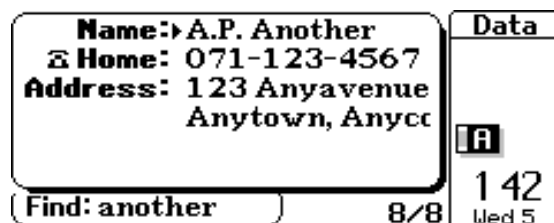
- Low battery indicator
- SSD pack indicators
- Remote link indicator
- Caps lock indicator

Each of these features can be *disabled* by setting the appropriate flags when configuring the window server using `wSystem`.

The following illustration shows the S3a Database application display with a permanent status window to the right of the screen. The application's main window has been neatly tiled with it:



On the *Workabout*, an equivalent display of the Database application appears as shown below:



Applications can have different 'modes' of operation, the precise definition being dependent on the application. As well as using menu options and 'hot' keys to switch between the different modes, an S3a application can set the diamond key to cycle around some or all of them.

By using the `wSetList` function, introduced in version 4, the program icon in the status window can be replaced by a list of modes. In the S3a display shown above, all three modes of the Database application are shown with the diamond symbol pointing to the current mode.

Note that the *Workabout* status window does not display either the application's icon or a list of modes.

Compatibility mode status window

In version 4, applications on the S3a and *Workabout* can run in S3 compatibility mode. This allows an application to have the 'look' and 'feel' of the same application running on an S3. On the S3a, this is achieved by doubling up the pixels. For example, a line which is 10 x 1 pixels on the S3 will be drawn 20 x 2 pixels on the S3a and should 'look' the same. On the *Workabout*, applications running in compatibility mode will normally exactly match the S3 appearance.

If an application is running in compatibility mode on the S3a or *Workabout*, then a call to `wsEnable` creates a compatibility mode status window which looks and behaves like an S3 status window.

Alternatively, the version 4 function `wStatusWindow` can be used to create a compatibility status window.

wsEnable Enable the permanent status window

```
VOID wsEnable(VOID);
```

Create and maintain a permanent status window, behind existing windows. Does nothing if a permanent status window already exists.

Before² calling `wsEnable`, the calling application should resize its main window such that it is tiled with the status window.

Under version 4, the required window extent should be determined by calling `wInquireStatusWindow` to get the size of the status window and then performing a simple calculation. Under earlier versions of the window server, use `wsScreenExt`.

In version 4, if running in compatibility mode on the S3a or *Workabout*, the status window will have the appearance of the S3 status window.

wStatusWindow Set the state of the status window

```
VOID wStatusWindow(INT state);
```

Available in version 4 only, this sets the permanent status window into one of a number of mutually exclusive states by setting the parameter `state` to one of the following:

<code>W_STATUS_WINDOW_OFF</code>	no status window is visible
<code>W_STATUS_WINDOW_SMALL</code>	display the small version of the status window
<code>W_STATUS_WINDOW_BIG</code>	display the full size version of the status window
<code>W_STATUS_WINDOW_CTBY</code>	display the S3 compatibility status window

Calling this function with the parameter value `W_STATUS_WINDOW_OFF` is equivalent to calling `wsDisable()`; calling this function with the parameter value `W_STATUS_WINDOW_BIG` (or `W_STATUS_WINDOW_CTBY` if in S3 compatibility mode) is equivalent to calling `wsEnable()`.

See the description of `wInquireStatusWindow` (or, for versions of the window server earlier than version 4, `wsScreenExt`) for a means of determining the size and position of a status window.

wsScreenExt Get screen extent for tile with status window

```
VOID wsScreenExt(P_EXTENT *pext);
```

This function should only be used when running versions of the window server earlier than version 4. It is available in version 4 for *compatibility only*.

It is strongly recommended that the function `wInquireStatusWindow` be used instead of `wsScreenExt` in version 4 of the window server.

The function writes the extent of the screen remaining to the data structure pointed to by `pext` when there is a permanent status window.

²The resize may fail with out of memory - so it is best to delay `wsEnable` until after the resize is successful.

The P_EXTENT struct is defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl;
    WORD width;
    WORD height;
} P_EXTENT;
```

Series 3 applications that support permanent status windows running under older versions of the window server (i.e. earlier than version 4) can use `wsScreenExt` to determine the extent of the window to use while a permanent status window is enabled.

Recall that the size of the entire screen (used in the absence of a permanent status window) may be obtained from the `WSERV_SPEC` struct filled in by `wConnect`. For example:

```
GLREF_D WSERV_SPEC *wserv_channel;

LOCAL_D P_POINT ScreenSize;

ScreenSize=wserv_channel->conn.info.pixels;
```

wsUpdate **Update the permanent status window**

```
VOID wsUpdate(INT flags);
```

Update the displayed permanent status window.

The parameter `flags` can be one of:

<code>WS_UPDATE_NAME</code>	to change the displayed permanent status window name (for example, after changing <code>DatStatusNamePtr</code>).
<code>WS_UPDATE_CLOCK</code>	to update any displayed clocks (for example, after changing settings such as 12/24 hour, analog/digital, the time separator and so on).

wsDisable **Disable the permanent status window**

```
VOID wsDisable(VOID);
```

Destroy the permanent status window (if one exists).

Before calling `wsDisable`, the calling application should resize its main window to take up the whole screen. As with `wsEnable`, the required window extent may be determined by calling `wInquireStatusWindow` and doing a simple calculation.

wsEnableTemp **Enable temporary status windows**

```
VOID wsEnableTemp(VOID);
```

Enable the window server's processing of `PSION+MENU` to present a temporary status window.

Unlike `wsEnable`, the effect of this call is system wide. On the S3 and S3a, `wsEnableTemp` is called by the shell as part of its initialisation.

wsDisableTemp **Disable temporary status windows**

```
VOID wsDisableTemp(VOID);
```

Disable the processing of `PSION+MENU` to present a temporary status window.

The effect of this call is system wide. Calling `wsDisableTemp` on the S3 and S3a will disable temporary status windows for all applications.

wsSetList **Set list of modes to display in status window**

```
INT wsSetList(UINT count, TEXT **plist, UINT pos);
```

Available in version 4 only, this function sets up the text for the list of modes to be displayed in the status window.

The `count` parameter is the number of text items in the list; the `plist` parameter is a pointer to an array of string pointers (one string per mode) and `pos` is the position within the list where the diamond symbol is to be placed. The first position is given a value of 0.

If the diamond symbol is not to be shown, `pos` should be set to `W_STATUS_WIN_NO_DIAMOND`.

To replace the list of modes with the application icon, `count` should be set to `W_STATUS_WINDOW_ICON`. With `count` set to this value, the other two parameters are ignored. Typically, a call would look like this:

```
wsSetList(W_STATUS_WINDOW_ICON, NULL, 0);
```

The function returns 0 if successful or `E_GEN_NOMEMORY` if it fails to allocate space for the new list.

Note that calling `wsSetList` on the *Workabout* has no visible effect, since its status window does not display a list of modes.

wsSelectList **Set select position in status window mode list**

```
VOID wsSelectList(INT pos);
```

Available in version 4 only, this function allows the diamond symbol in the list of modes in the status window to be (re-)positioned.

The position is specified by giving a value to the parameter `pos`. The first position is given a value of 0. If the diamond symbol is not currently shown, setting a position will cause it to reappear. Giving `pos` a value of `W_STATUS_WIN_NO_DIAMOND` causes the diamond symbol to be removed from the status window.

Note that calling `wsSelectList` on the *Workabout* has no visible effect, since its status window does not display a list of modes.

wInquireStatusWindow **Inquire state and extent of status window**

```
INT wInquireStatusWindow(INT state, P_EXTENT *pextent);
```

Available in version 4 only, this function does two things:

- it returns the current state of the status window as set by `wStatusWindow`. It returns one of the values `W_STATUS_WINDOW_OFF`, `W_STATUS_WINDOW_SMALL`, `W_STATUS_WINDOW_BIG` and `W_STATUS_WINDOW_CTBY`.
- it fills in the `pextent` of the status window corresponding to `state`. In other words, by setting `state` to one of the values `W_STATUS_WINDOW_OFF`, `W_STATUS_WINDOW_SMALL`, `W_STATUS_WINDOW_BIG` or `W_STATUS_WINDOW_CTBY`, it supplies the position, width and height of a status window of that type.
Further, if `state` is given a value of -1, the extent of the *current* status window is supplied.

It is interesting to note that if the status window is off, the extent information describes a status window located at the right hand edge of the screen with zero width and full height.

For a description of the `P_EXTENT` structure, see `wsScreenExt`.

Configuring the window server

The functions which have a system wide effect on the window server (as opposed to just affecting the calling client) are:

<code>wSystem</code>	which is described next
<code>wsEnableTemp</code>	to enable/disable permanent status windows (as described above)
<code>wsDisableTemp</code>	

These functions should only be used when an application takes over the whole machine. This is more likely on an HC than say an S3, S3a or *Workabout*.

wSystem **Configure the window server**

```
INT wSystem(UINT new_flags,UINT flag_mask);
```

Set an internal set of flags to modify the system-wide behaviour of the window server where:

<code>new_flags</code>	is a bit mask containing the values of the bit flags to be modified
<code>flag_mask</code>	is a bit mask indicating (by those bits that are set) the bit flags that are to be modified

The function returns the old value of the flags.

The flags are of the form `WSERV_FLAG_XXX` where XXX is one of:

<code>NO_SHELL_REBOOT</code>	Stops the window server from restarting the shell (that is, <i>sys\$shll.img</i>) whenever it terminates. Clearing this flag when there is no shell running causes the window server to restart the shell. Available on all machines.
<code>NO_NOTIFIER_REBOOT</code>	The same as above except it applies to the notifier process (<i>sys\$ntfy.img</i>). Available on the HC and MC. Prior to version 4, the window server on the S3 always provides the notifier itself and never starts a <i>sys\$ntfy.img</i>. Under version 4, this flag can be set for the S3a and <i>Workabout</i> because the possibility of building a separate notifier process exists.
<code>HOOK_NOTIFIER</code>	If set, the window server attempts to hook the notifier, as explained in the earlier <i>Alerts</i> section of this chapter. All MC versions are unable to hook the notifier. Prior to version 4, this flag applies to the HC only; the S3 effectively assumes that it is permanently set. Under version 4, this flag can be set for the S3a and <i>Workabout</i>.
<code>NO_PANIC_NOTIFY</code>	Disables the window server from reporting processes which terminate with a panic or a negative reason code. Available on the HC, S3, S3a and <i>Workabout</i> but not the MC. This flag is ignored unless the window server has hooked the notifier. On machines other than the MC, you would set this flag to prevent the window server from reporting abnormal terminations when <code>HOOK_NOTIFIER</code> is set.
<code>UPDATE_MSGS</code>	Enables the window server to send <code>WM_TASK_UPDATE</code> events to the shell to inform it of the termination of any process (not just clients of the window server). The MC version does not support <code>WM_TASK_UPDATE</code> events. Prior to version 4, this flag applies to the HC only; the S3 effectively assumes that it is permanently set. Under version 4, this flag can be set for the S3a and <i>Workabout</i>.

LOW_BATTERY_WARNINGS If set, the window server notifies the user of a low lithium or main battery. Note that the window server only checks for low battery when the machine is turned on. On the HC, the window server is only informed of the machine being switched on after `p_setonevent(TRUE)` has been called. The MC version of the window server does not support low battery warnings.

Prior to version 4, this flag only applies to the HC; on the S3, the window server always reports low battery warnings.

Under version 4, this flag can be set for the S3a and *Workabout*.

HUNG_UP_SW If set, the window server presents a "hung up" status window if the foreground task is not using backed-up windows and fails to respond to redraw events. The MC version of the window server does not support status windows.

Prior to version 4, this flag only applies to the HC; the S3 effectively assumes that it is permanently set.

Under version 4, this flag can be set for the S3a and *Workabout*.

The following flags are introduced in version 4 and apply only to S3a and *Workabout* machines.

SW_NO_LOW_BATTERY If set, it disables the low battery indicator in the status window.

SW_NO_PACKS If set, it disables the two pack indicators in the status window.

SW_NO_LINK If set, it disables the link indicator in the status window.

SW_NO_CAPS If set, it disables the caps lock indicator in the status window.

Prior to version 4, when the window server starts, the internal flags are all clear although, on the HC, this can be altered by setting the `$WS_FL` environment variable.

Under version 4 of the window server the internal flags on the S3a and *Workabout* can be altered, like on the HC, by setting the `$WS_FL` environment variable.

In practice, `wSystem` is more likely to be used on the HC rather than the S3, S3a, *Workabout* or MC. See the section *System start-up* in the *Introduction* chapter for further discussion (including further details on `$WS_FL`) and examples of the use of `wSystem`.

Attached Clients

This section only applies to large screen versions of the window server, such as the MC.

Clients can attach to and detach from each other by use of the `wAttachToClient`, `wAttachToForegroundClient` and `wDetachClient` calls.

The client that calls the attach function is attached in front of the client it is attaching to.

When clients are attached they move round in the task order together - when one of the attached tasks moves, it pulls the other task (or tasks) with it. When they become foreground, all attached tasks are sent a `WM_FOREGROUND` event.

Two examples of the use of attached clients on the MC are:

- The system notifier `sys$ntfy` uses `wAttachToForegroundClient` to attach itself to the foreground client to display its message.
- The voice server uses `wAttachToClient` to attach itself to *its* client (where both are clients of the window server) to implement a dialog box as a separate process.

In both cases, the attaching client is behaving as if it were part of the client it is attached to. The notifier could have been implemented using `wClientPosition` to make itself visible but the holder of the foreground would then inappropriately go into background (and get a `WM_BACKGROUND` event).

When a client attaches to another, the window server sends a `WM_ATTACHED` event to the client being attached to. When the attaching client detaches, it sends a `WM_DETACHED` event to the client being detached from.

A detaching client is positioned to the back of all clients.

wAttachToClient**Attach to client**

```
INT wAttachToClient(UINT pid);
```

Attach the caller to client `pid`.

If client `pid` does not exist, the function leaves or returns `E_FILE_NXIST`.

wAttachToForegroundClient**Attach to foreground client**

```
VOID wAttachToForegroundClient(VOID);
```

Attach the caller to the foreground client.

Does nothing if the caller has the foreground.

wDetachClient**Detach from client**

```
VOID wDetachClient(VOID);
```

Detach from a client and position to the back of all clients.

If the caller is no longer attached to another client (say because that client has terminated), the caller is just positioned to the back.

Miscellaneous

A number of general functions which do not fit under any of the previously discussed topics are described here. Unless otherwise stated, they are all introduced in version 4 of the window server.

wSupportInfo**Get information on supported features**

```
VOID wsSupportInfo(.i.W_SUPPORT_INFO *pinfo);
```

The function fills the `W_SUPPORT_INFO` type structure with information on the currently supported features. The supplied parameter `pinfo` must point to a structure of type `W_SUPPORT_INFO`.

It can set the following values in the `flags` member:

<code>W_SUPPORT_GREY</code>	if set, the window server supports the current scheme for drawing grey graphics
<code>W_SUPPORT_CTBY_S3</code>	if set, the window server supports a Series 3 compatibility mode (there is no distinction between different Series 3 compatibility modes, such the two that are available on the <i>Workabout</i>)

Currently, no other information is returned. The rest of the `W_SUPPORT_INFO` structure is set to zeros. The function is well placed for expansion in future versions and releases of the window server.

The `W_SUPPORT_INFO` structure is defined as follows:

```
typedef struct
{
    UINT flags;
    UINT fillers[15]; /* will be filled with 0's */
} W_SUPPORT_INFO;
```

wDisableKeyClick**Set or cancel key click disable state**

```
VOID wDisableKeyClick(INT state);
```

If `state` is set to `TRUE`, the behaviour of the key click for an application is changed:

- The key click is disabled while the application is in foreground.
- The key click state is reset when the application goes to background.

Setting `state` to `FALSE` cancels this state for an application.

CHAPTER 3

WINDOWS

Creating and initialising a window

Window attributes

The functions:

wCreateWindow	to create a window and set its attributes
wSetWindow	to set a window's attributes
wInquireWindow	to sense a window's attributes

take the address of a `W_WINDATA` struct as a parameter to hold the window attributes where the `W_WINDATA` struct is defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl;
    WORD width;
    WORD height;
} P_EXTENT;

typedef struct
{
    UWORD flags;
    P_EXTENT extent;
    WORD mouse_icon;
    UBYTE background;
    UBYTE filler;
} W_WINDATA;
```

where:

flags	is a set of binary attributes, described below. In the root window, the flags field is zero (so all the binary attributes are clear).
extent	is the position and size of the window relative to its parent in pixel coordinates. In the root window the extent coincides with the whole screen.
mouse_icon	is the ID of the window mouse icon (only used when the machine has a pointing device such as on the MC)
background	specifies whether the window is backed-up by a bitmap (or bitmaps) and, if not, how the window should be prepared when it is validated

background (up to version 3.5)

In versions of the window server up to and including 3.5 the `background` field of the `W_WINDATA` struct should be one of:

<code>W_WIN_BACK_BITMAP</code>	all drawing to the window is duplicated to an off-screen bitmap. Redraws are automatically done from this bitmap, so no <code>WM_REDRAW</code> events are ever generated. When the background has this value, it may not be altered by calling <code>wSetWindow</code> . Not available on version 2 of the window server.
<code>W_WIN_BACK_CLR</code>	clear the pixels in the window on validation (the root window has this value). This is the default value.
<code>W_WIN_BACK_SET</code>	set the pixels in the window on validation.
<code>W_WIN_BACK_NONE</code>	do nothing on validation (on the assumption that the drawing covers every pixel). Best for flicker-free graphics.

background (version 4)

In version 4 of the window server, there are changes in meaning and values caused by the introduction of grey. To control drawing to the normal (black) plane, one of the following must be set:

<code>W_WIN_BACK_BITMAP</code>	all drawing to the normal (black) plane of the window is duplicated to an off-screen bitmap. Redraws are automatically done from this bitmap, so no <code>WM_REDRAW</code> events are ever generated. When the background has this value, it may not be altered by calling <code>wSetWindow</code> .
<code>W_WIN_BACK_CLR</code>	clear the pixels in the normal (black) plane of the window on validation (the root window has this value). This is the default value.
<code>W_WIN_BACK_SET</code>	set the pixels in the normal (black) plane of the window on validation.
<code>W_WIN_BACK_NONE</code>	do nothing to the normal (black) plane of the window on validation (on the assumption that the drawing covers every pixel). Best for flicker-free graphics.
<code>W_WIN_BACK_CLR_NO_REDRAW</code>	to clear the pixels in the normal (black) plane of the window but prevent any drawing or redrawing to this specific plane.
<code>W_WIN_BACK_SET_NO_REDRAW</code>	to set the pixels in the normal (black) plane of the window but prevent any drawing or redrawing to this specific plane.
<code>W_WIN_BACK_NONE_NO_REDRAW</code>	do nothing to the normal (black) plane of the window on validation but also prevent any drawing or redrawing to this specific plane.

To control drawing to the grey plane, one of the following must be `ORED` into the `background` field with one of the above normal plane values:

<code>W_WIN_BACK_GREY_BITMAP</code>	all drawing to the grey plane of the window is duplicated to an off-screen bitmap. Redraws are automatically done from this bitmap, so no <code>WM_REDRAW</code> events are ever generated. When the background has this value, it may not be altered by calling <code>wSetWindow</code> .
<code>W_WIN_BACK_GREY_CLR</code>	clear the pixels in the grey plane of the window on validation (the root window has this value).
<code>W_WIN_BACK_GREY_SET</code>	set the pixels in the grey plane of the window on validation.
<code>W_WIN_BACK_GREY_NONE</code>	do nothing to the grey plane on validation (on the assumption that the drawing covers every pixel). Best for flicker-free graphics.
<code>W_WIN_BACK_GREY_CLR_NO_REDRAW</code>	to clear the pixels in the grey plane of the window but prevent any drawing or redrawing to this specific plane. This is the default value.
<code>W_WIN_BACK_GREY_SET_NO_REDRAW</code>	to set the pixels in the grey plane of the window but prevent any drawing or redrawing to this specific plane.
<code>W_WIN_BACK_GREY_NONE_NO_REDRAW</code>	do nothing to the grey plane of the window on validation but also prevent any drawing or redrawing to this specific plane.

If drawing is enabled to both the normal (black) and the grey planes, then both planes will be moved when scrolling or moving a window.

If no drawing is intended for one of the planes in a window, then overheads can be cut by disabling the unused plane. For example, if no drawing is to be done to the grey plane, disable this plane by *NOT* setting any of `W_WIN_BACK_GREY_CLR`, `W_WIN_BACK_GREY_SET`, `W_WIN_BACK_GREY_NONE` or `W_WIN_BACK_GREY_BITMAP`.

Note that if both `W_WIN_BACK_BITMAP` and `W_WIN_BACK_GREY_BITMAP` are set, then the window will be backed up to two bitmaps, one for the normal plane and one for the grey plane.

Also note that setting one of:

```
W_WIN_BACK_CLR_NO_REDRAW
W_WIN_BACK_SET_NO_REDRAW
W_WIN_BACK_NONE_NO_REDRAW
```

and setting one of:

```
W_WIN_BACK_GREY_CLR_NO_REDRAW
W_WIN_BACK_GREY_SET_NO_REDRAW
W_WIN_BACK_GREY_NONE_NO_REDRAW
```

is equivalent to setting the `W_WIN_NO_REDRAW` bit (as a parameter to `wCreateWindow`).

flags (all versions)

The following bits of `flags` apply to all versions of the window server on all machines:

<code>W_WIN_NO_REDRAW</code>	windows with this flag set never receive redraw events. This flag may not be altered by a <code>wSetWindow</code> command.
<code>W_WIN_PRIORITY</code>	redraw events for windows with this flag set have priority over redraw events for windows with this flag clear.

flags (version 4)

The following bit of `flags` applies to version 4 of the window server:

<code>W_WIN_DOUBLE_PIXEL</code>	when set, causes all graphics in this window to work in double pixel mode.
---------------------------------	--

Large screen flags

The following bit of `flags` applies only to large screen versions of the window server such as the MC:

<code>W_WIN_FOREGROUND_ONLY</code>	if set, the window is only visible while the client is foreground. Note that a descendant window of a <code>FOREGROUND_ONLY</code> window is necessarily also <code>FOREGROUND_ONLY</code> regardless of the value of this flag. This flag may not be altered by a <code>wSetWindow</code> command.
------------------------------------	---

Mouse-related flags

The following bit of `flags` apply only when the machine has a pointing device, such as on the MC:

<code>W_WIN_NO_MOUSE</code>	a window with this flag set will not receive any mouse events (however, the mouse cursor is still displayed). All other mouse-related flags have no affect when this flag is set.
<code>W_WIN_INACTIVE</code>	if a mouse click occurs anywhere in a window with this flag set or in any of its descendants, a <code>WM_ACTIVE</code> event is sent to the window.
<code>W_WIN_INPUT_ONLY</code>	if set, the window is input-only. Input-only windows are invisible and exist solely for the purpose of detecting mouse events. This flag may not be altered by a <code>wSetWindow</code> command.
<code>W_WIN_MOUSE_MOVE</code>	if set, mouse movement events are generated when the mouse button is up.
<code>W_WIN_MOUSE_DRAG</code>	if set, mouse movement events are generated when the mouse button is down.

W_WIN_MOUSE_GRAB	if set, the mouse is automatically grabbed when the mouse button is pressed. The grab is automatically released when the mouse button is released and a WM_MOUSE event of type WM_MOUSE_RELEASE is sent to the grabbing window, even if the release occurs outside the window. If W_WIN_MOUSE_DRAG is also set, any intermediate WM_MOUSE_MOVE events are also delivered to the grabbing window.
W_WIN_RUBBER_BAND_CAPTURE	If a mouse down event occurs in a window with this bit set, it and subsequent mouse and keyboard events are captured to the window server's rubber band processing until the rubber band mode is terminated. This first mouse click generates a WM_RUBBER_BAND_INIT event to which the client must respond with a call to wRubberBand. This flag may not be altered by a wSetWindow command.
W_WIN_RUBBER_BAND_COMPLETE_ON_RELEASE	If when calling wRubberBand in response to a WM_RUBBER_BAND_INIT event you specify that the rubber band should be completed on a mouse up event, you should also set this bit. This flag may not be altered by a wSetWindow command.

wCreateWindow

Create a window

```
INT wCreateWindow(UINT parent_id, UINT field_set, .i.W_WINDATA *pwindata, UWORD handle);
```

Create a window and, if successful, return the positive ID of the window where:

parent_id	is the window ID of the parent window. To create a top-level window, where the parent is the root window (ie the whole screen), pass parent_id as zero.
handle	is the client's own identifier for the window (which must be non-zero) to be embedded in events which are directed at the window (for example, redraw and mouse events). In medium to large applications, handle is commonly the address of a control block which contains the window ID.
field_set	is a set of bit flags which specify (by being set) which fields in the pwindata struct are to be used to set the window attributes. In most cases, an attribute which is not set from pwindata is inherited from parent_id (as detailed below). If field_set is zero, pwindata is ignored.
pwindata	is the address of a W_WINDATA struct as described above. If field_set is zero, pwindata is ignored.

The bits in field_set are made up of the same bit masks as for pwindata->flags to indicate that the corresponding bit in pwindata->flags should be used. And, in addition, field_set may contain:

W_WIN_EXTENT	to use pwindata->extent
W_WIN_MOUSE_ICON	to use pwindata->mouse_icon
W_WIN_BACKGROUND	to use pwindata->background

If a field_set bit is clear, the corresponding attribute is inherited from the parent window parent_id, except for:

```
W_WIN_RUBBER_BAND_CAPTURE
W_WIN_INACTIVE
W_WIN_NO_REDRAW
W_WIN_PRIORITY
W_WIN_FOREGROUND_ONLY
```

These five attributes are never inherited from any parent window. If they are not set explicitly, by setting the appropriate bit in field_set and the corresponding data value in the W_WINDATA struct pointed to by pwindata, they are set to zero.

If parent_id has one or more child windows, the new window is created in front of its siblings.

After a successful return from wCreateWindow, the window is just a dormant data structure in the window server's data segment with no visibility on the screen. You can't draw to the window and you won't get any redraw events (or any mouse events if there is a pointing device) until the window is initialised by calling wInitialiseWindowTree, described below.

If the function fails because there is insufficient memory, it calls `p_leave(E_GEN_NOMEMORY)` or returns `E_GEN_NOMEMORY`, depending on whether `wDisableLeaves` has been called.

See the description of `wConnect` for an example.

wSetWindow

Set window attributes

```
VOID wSetWindow(UINT wid, UINT field_set, .i.W_WINDATA *pwndata);
```

Set one or more window attributes of the window with ID `wid` where `field_set` and `pwndata` are as for `wCreateWindow`, described above.

The function `wSetWindow` ignores the following bits in `field_set` which correspond to window attributes which are not modifiable:

```
W_WIN_NO_REDRAW
W_WIN_INPUT_ONLY
W_WIN_FOREGROUND_ONLY
W_WIN_RUBBER_BAND_COMPLETE_ON_RELEASE
```

You cannot modify any of the attributes of the root window.

In practice, `wSetWindow` is commonly use to move and/or resize a window.

If the size of a backed-up window is increased, the backup bitmap(s) will also be increased and the additional area (to the right and below) is filled with zeros.

In this situation, the call can fail with an out-of-memory condition; it should be noted that such a failure might not be reported immediately because of the buffering of window server requests (see the *Clients and the window server* section in the *Introduction* chapter).

Any areas of the window that are exposed as a result of making the call to `wSetWindow` (because the effect has been to expand or move the window) are invalidated. Areas that are covered, moved offscreen, or lost because the window has become smaller are marked as valid, thus preventing any redraws that might have been pending for these areas. The validity of any other areas is not affected by the call.

wInquireWindow

Get window attributes

```
INT wInquireWindow(UINT wid, .i.W_WINDATA *pwndata);
```

Write a copy of window `wid`'s extent and flags to `*pwndata` (`pwndata->mouse_icon` and `pwndata->background` are left undefined).

wInitialiseWindowTree

Initialise window tree

```
VOID wInitialiseWindowTree(UINT wid);
```

Initialise window `wid` and all its descendants.

None of these windows may be initialised again.

Provided that the window is not made invisible between creation and initialisation, the following occurs:

backed-up window ¹	The backup bitmap (which is initialised with zeros when the window is created) is copied to any visible parts of the window, clearing it. In version 4 of the window server there may be two backup bitmaps which are copied to the visible parts of the window's normal and grey plane respectively.
no-redraw window ²	Any visible pixels are cleared if the background attribute is <code>W_WIN_BACK_CLR</code> or set if the background attribute is <code>W_WIN_BACK_SET</code> . In version 4 of the window server, these attributes clear or set the visible pixels in the window's normal plane while <code>W_WIN_BACK_GREY_CLR</code> and <code>W_WIN_BACK_GREY_SET</code> clear or set the visible pixels in the window's grey plane.
non-backed-up redraw window	Any visible parts are added to the update region (which will subsequently cause redraw events to be generated).

¹Where the window attribute background is `W_WIN_BACK_BITMAP` (and/or `W_WIN_BACK_GREY_BITMAP` in version 4 of the window server).

²Where the `W_WIN_NO_REDRAW` window attribute is set.

wCloseWindowTree **Destroy a window and its descendants**

```
VOID wCloseWindowTree(UINT wid);
```

Destroy window `wid` and all its descendants, freeing any associated window server resources (such as an attached graphics context).

When destroying a window system, it is sometimes difficult to destroy windows bottom-up with respect to the window parentage tree. To make life easier, the window server allows windows to be destroyed more than once as long as other window server objects, such as graphics contexts, bitmaps, fonts, icons or windows are not created in the mean time.

If you are using `wGetEvent` or `wGetEventSpecial` rather than `wGetEventWait`, you should be very careful about destroying windows while a request made by calling `wGetEvent` or `wGetEventSpecial` is pending. Bearing in mind the multi-tasking nature of the system and the fact that the window server runs at a higher priority than its clients, it is quite possible for an event to be delivered before the window is destroyed in which case there is the possibility that the next event will relate to a window which has already been destroyed (a redraw event, say).

The solution to this problem normally involves calling `wCancelGetEvent`, as described in the first chapter.

wInquireWindowOffset **Get window to window offset**

```
INT wInquireWindowOffset(UINT from_wid, UINT to_wid, .i.P_POINT *poffset);
```

Write the offset of `to_wid` relative to `from_wid` to `poffset` and return zero.

The `P_POINT` struct is:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;
```

Useful when positioning windows relative to each other. To find the absolute position of a window on the screen set `from_wid` to zero (the root window ID).

If the window server has a pending error with error number `err`, the function calls `p_leave(err)` or returns `err`, depending on whether `wDisableLeaves` has been called.

wReassignRootWindow **Reassign the root window**

```
VOID wReassignRootWindow(UINT wid);
```

Reassign the window ID of zero to mean the window `wid` rather than the root window which covers the whole screen.

The assignation applies only to the calling client.

Passing a `wid` of zero resets the window ID of zero to mean the whole screen.

Used for creating development environments on a large screen version of the window server which simulate a small screen environment.

Not available in version 2 of the window server.

Visible and invisible windows

wMakeInvisible **Make window invisible**

```
VOID wMakeInvisible(UINT wid);
```

Mark window `wid` as invisible.

When a window is marked as invisible it and all its descendants are made invisible.

The window server treats windows which are invisible as follows:

- if the window is backed-up, any drawing to it is drawn only to the backup bitmap(s)
- invalidating a window using `wInvalidateRect` or `wInvalidateWin` has no effect
- windows behave as if they do not exist with respect to mouse input (this only applies when there is a pointing device)

wMakeVisible**Make window visible**

```
VOID wMakeVisible(UINT wid);
```

Mark window `wid` as visible.

The window will actually become visible only if it is initialised and all its ascendants are visible.

Unless `wMakeInvisible` has been applied to a descendant, calling `wMakeVisible` makes all descendants visible.

Sibling positions

wWindowPosition**Change position in sibling list**

```
VOID wWindowPosition(UINT wid, UINT pos);
```

Move window `wid` to the position `pos` in its sibling list.

If `pos` is greater than the number of siblings then window `wid` will go to the back of the sibling list, if `pos` is zero then it will go to the front of the sibling list.

wGetWindowPosition**Get position in sibling list**

```
INT wGetWindowPosition(UINT wid);
```

Return the position of window `wid` in its sibling list.

The function behaves as for `wCheckPoint` in that it flushes the client-side buffer and reports any uncleared error - either by calling `p_leave` or by returning the error number.

Not available in version 2 of the window server.

Scrolling

wScrollRect**Copy a rectangle**

```
VOID wScrollRect(UINT wid, .i.P_RECT *prect, P_POINT *poffset);
```

Copy the pixels in rectangle `prect` in window `wid` to the same sized rectangle, displaced by `poffset`.

The structs `P_POINT` and `P_RECT` are defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;
```

The copy excludes the following from `prect`:

- those parts which are in the update region (that is, those parts which are invalid)
- those parts which are obscured or clipped by other windows
- those parts which are beyond the boundaries of the window

Although these parts are not copied, their existence causes the corresponding region of the displaced rectangle to be invalidated.

In version 4 of the window server, if drawing is enabled to both the normal and the grey planes, then the copying activity described above is done to both planes.

If the window is a backed-up window, the copy is also applied to the backup bitmap(s). For a backed-up window: the update region is always null; those parts which are obscured or clipped can be recovered from the backup bitmap(s); those parts which are beyond the boundaries of the window are cleared.

Note that `wScrollRect` may also be applied to a bitmap where `wid` is a bitmap ID (but not in version 2 of the window server).

wScrollWin

Scroll a window

```
VOID wScrollWin(UINT wid, .i.P_POINT *poffset);
```

Scroll the whole of window `wid` by displacement `poffset`.

Entirely equivalent to `wScrollRect` with `prect` set to a rectangle covering the whole window.

Note that `wScrollWin` may also be used to scroll a whole bitmap where `wid` is a bitmap ID (but not in version 2 of the window server).

Redrawing

There are six variants of `wBeginRedraw` which vary according to whether a temporary graphics context is created (and, if so, whether it is to be altered from its default settings) and whether a part or the whole of the window is being redrawn, as follows:

<code>wBeginRedraw</code>	to redraw a part of the window using an independently created temporary or permanent graphics context
<code>wBeginRedrawWin</code>	to redraw the whole of the window using an independently created temporary or permanent graphics context
<code>wBeginRedrawGC</code>	to redraw a part of the window using a temporary graphics context which is created and initialised with specified values
<code>wBeginRedrawGC0</code>	to redraw a part of the window using a temporary graphics context which is created with default initial values
<code>wBeginRedrawWinGC</code>	to redraw the whole of the window using a temporary graphics context which is created and initialised with specified values
<code>wBeginRedrawWinGC0</code>	to redraw the whole of the window using a temporary graphics context which is created with default initial values

Note that the last character in both `wBeginRedrawGC0` and `wBeginRedrawWinGC0` is the digit zero (and not the letter 'o').

When a begin redraw function is used to simultaneously create a temporary graphics context, the corresponding call to `wEndRedraw` automatically frees it.

All variants validate at least a part of a window and cause subsequent drawing to use the update region rather than the normal drawing region. If the background attribute of the window is `W_WIN_BACK_CLR`, validated pixels in the update region are cleared and if the background attribute is `W_WIN_BACK_SET`, the same pixels are set.

In version 4 of the window server, the above two background attributes apply to validated pixels in the normal (black) plane in the update region. In addition, if the background attribute of the window is `W_WIN_BACK_GREY_CLR` or `W_WIN_BACK_GREY_SET`, validated pixels in the grey plane in the update region are cleared or set respectively.

A way of ensuring flicker-free redrawing, is to set the background attribute to `W_WIN_BACK_NONE` and cover every pixel in the redraw.

In version 4 of the window server, `W_WIN_BACK_NONE` applies only to the normal (black) plane. To achieve the same effect when drawing to the grey plane, the corresponding attribute `W_WIN_BACK_GREY_NONE` should also be set.

wBeginRedraw**Start a partial redraw**

```
VOID wBeginRedraw(UINT wid, .i.P_RECT *prect);
```

Validate the rectangle `prect` in window `wid` and clip subsequent drawing to the intersection of `prect` and the window's update region (as it was before the validate). The normal drawing region is restored when `wEndRedraw` is called.

The struct `P_RECT` is defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;
```

wBeginRedrawWin**Start a full redraw**

```
VOID wBeginRedrawWin(UINT wid);
```

Validate the whole of window `wid` and clip subsequent drawing to the window's update region (as it was before the validate). The normal drawing region is restored when `wEndRedraw` is called.

Equivalent to calling `wBeginRedraw` with a rectangle covering all of window `wid`.

wBeginRedrawGC**Start a partial redraw (GC)**

```
VOID wBeginRedrawGC(UINT wid, .i.P_RECT *prect, UINT field_set, G_GC *pgc);
```

Validate the rectangle `prect` in window `wid` and clip subsequent drawing to the window's update region (as it was before the validate).

In addition, create a temporary graphics context which is initialised with those fields from `pgc` which have their corresponding bit fields set in `field_set`.

The `G_GC` struct is defined in *wlib.h* as:

```
typedef struct
{
    UBYTE gmode; /* G_TRMODE_SET, _CLR, _INV (line) */
    UBYTE textmode; /* G_TRMODE_SET, _CLR, _INV, _REPL (text) */
    UBYTE style; /* G_STY_NORMAL, _BOLD, _UNDERLINE, _INVERSE, _DOUBLE, _MONO,
    _ITALIC */
    UBYTE flags; /* controls use of grey & double pixel mode */
    WORD font; /* ID of font to use */
} G_GC;
```

where the bit fields for `field_set` are:

<code>G_GC_MASK_GMODE</code>	corresponding to the <code>gmode</code> member
<code>G_GC_MASK_TEXTMODE</code>	corresponding to the <code>textmode</code> member
<code>G_GC_MASK_STYLE</code>	corresponding to the <code>style</code> member
<code>G_GC_MASK_FONT</code>	corresponding to the <code>font</code> member
<code>G_GC_MASK_GREY</code>	corresponding to the <code>flags</code> member (version 4 only)
<code>G_GC_MASK_DOUBLE</code>	corresponding to the <code>flags</code> member (version 4 only)

The `flags` member is introduced in version 4 of the window server. Prior to version 4, the space occupied by this member was unused.

See the *Graphics Output* chapter for a complete description of the fields in a `G_GC` struct.

The graphics context is freed (and the normal drawing region is restored) when `wEndRedraw` is called.

wBeginRedrawGC0 **Start a partial redraw (GC0)**

```
VOID wBeginRedrawGC0(UINT wid, .i.P_RECT *prect);
```

Validate the rectangle prect in window wid and clip subsequent drawing to the window's update region (as it was before the validate).

In addition, create a temporary graphics context with default initial values.

The graphics context is freed (and the normal drawing region is restored) when wEndRedraw is called.

wBeginRedrawWinGC **Start a full redraw (GC)**

```
VOID wBeginRedrawWinGC(UINT wid, UINT field_set, .i.G_GC *pgc);
```

Validate the whole window wid and clip subsequent drawing to the window's update region (as it was before the validate).

In addition, create a temporary graphics context which is initialised with those fields from pgc which have their corresponding bit fields set in field_set.

The graphics context is freed (and the normal drawing region is restored) when wEndRedraw is called.

The G_GC struct is defined in *wlib.h* as:

```
typedef struct
{
    UBYTE gmode;      /* G_TRMODE_SET, _CLR, _INV (line) */
    UBYTE textmode;   /* G_TRMODE_SET, _CLR, _INV, _REPL (text) */
    UBYTE style;      /* G_STY_NORMAL, _BOLD, _UNDERLINE, _INVERSE, _DOUBLE, _MONO,
    _ITALIC */
    UBYTE flags;      /* controls use of grey & double pixel mode */
    WORD font;        /* ID of font to use */
} G_GC;
```

where the bit fields for field_set are:

G_GC_MASK_GMODE	corresponding to the gmode member
G_GC_MASK_TEXTMODE	corresponding to the textmode member
G_GC_MASK_STYLE	corresponding to the style member
G_GC_MASK_FONT	corresponding to the font member
G_GC_MASK_GREY	corresponding to the flags member (version 4 only)
G_GC_MASK_DOUBLE	corresponding to the flags member (version 4 only)

The flags member is introduced in version 4 of the window server. Prior to version 4, the space occupied by this member was unused.

See the *Graphics Output* chapter for a complete description of the fields in a G_GC struct.

Example

```
LOCAL_C VOID BeginRedraw(INT wid,INT fid,INT style)
{
    G_GC gc;

    gc.font=fid;
    gc.style=style;
    wBeginRedrawWinGC(wid,G_GC_MASK_FONT|G_GC_MASK_STYLE,&gc);
}
```

wBeginRedrawWinGC0 **Start a full redraw (GC0)**

```
VOID wBeginRedrawWinGC0(UINT wid);
```

Validate the whole of window wid and clip subsequent drawing to the window's update region (as it was before the validate).

In addition, create a temporary graphics context with default initial values.

The graphics context is freed (and the normal drawing region is restored) when wEndRedraw is called.

wEndRedraw**End a redraw**

```
VOID wEndRedraw(VOID);
```

End a redraw.

Drawing is set back to use the normal drawing region (which clips to the visible region of a window).

If a temporary graphics context was created by calling `wBeginRedrawGC`, `wBeginRedrawWinGC`, `wBeginRedrawGC0` or `wBeginRedrawWinGC0`, the temporary graphics context is freed.

Validating

Validating a rectangle in a window removes that rectangle from the window's update region (if the window has an update region). Validating the whole window deletes the window's update region.

When responding to the receipt of a `WM_REDRAW` event (see the *Events* chapter) it is essential to perform a validation: until a window's update region has been entirely validated, `WM_REDRAW` events will continue to be received.

The normal response to a `WM_REDRAW` event is to perform a validation and to draw all or part of the window (drawing will be clipped to the intersection with the window's update region). The rectangle that is validated should correspond exactly to the rectangle that is drawn, rather than to the rectangle specified by the `WM_REDRAW` event.

Depending on the window's background attribute, validation may also set, clear or leave unchanged all the pixels in the rectangle. You should set the background attribute to select the action that is most appropriate for the particular situation.

If the background attribute of the window is `W_WIN_BACK_CLR`, validated pixels in the drawing region are cleared and if the background attribute is `W_WIN_BACK_SET`, the same pixels are set.

In version 4 of the window server, these attributes refer to validated pixels in the normal (black) plane of the update region. In addition, if the background attribute `W_WIN_BACK_GREY_CLR` is set, validated pixels in the grey plane of the drawing region are cleared and if the background attribute `W_WIN_BACK_GREY_SET` is set, the same pixels are set.

A way of ensuring flicker-free redrawing of a window is to set the background attribute to `W_WIN_BACK_NONE` and cover every pixel in the draw.

In version 4 of the window server, the `W_WIN_BACK_NONE` attribute refers to the normal (black) plane. If you are also drawing to the grey plane, then the same result can be achieved in this plane by setting the `W_WIN_BACK_GREY_NONE` attribute.

wValidateRect**Validate a rectangle of a window**

```
VOID wValidateRect(UINT wid, .i.P_RECT *prect);
```

Validate the rectangle `prect` in window `wid`.

The struct `P_RECT` is defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;
```

wValidateWin**Validate a whole window**

```
VOID wValidateWin(UINT wid);
```

Validate the whole of window `wid`.

Equivalent to calling `wValidateRect` with a rectangle covering window `wid`.

Invalidating

Invalidating a rectangle in a window adds that rectangle to the window's update region. Invalidating the whole window sets the window's update region to cover the whole window.

When a client wishes to draw an area in one of its windows, there are two approaches:

- to draw directly to the area (normally after calling `wValidateRect` or `wValidateWin`)
- to invalidate a rectangle and make use of the code which redraws its window in response to a `WM_REDRAW` event

The advantages of invalidating are:

- It makes use of the code which must in any case be provided to redraw the window.
- An application can effectively use the window server's update region to accumulate disjoint invalid areas without having to worry about whether those areas overlap (since overlapping areas will generate a single redraw).
- Because the areas invalidated are clipped to the visible areas of the window, responding to the `WM_REDRAW` events can require less work than drawing the whole window (since the whole window may be partially or totally obscured or it may be invisible).

The main disadvantage of invalidating is the loss of performance resulting from the extra context switching from the client to the window server to process the invalidate command and then back to the client to process the redraw.

wInvalidateRect

Invalidate a rectangle

```
VOID wInvalidateRect(UINT wid, .i.P_RECT *prect);
```

Invalidate the rectangle `prect` in window `wid`.

The struct `P_RECT` is defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;
```

wInvalidateWin

Invalidate a window

```
VOID wInvalidateWin(UINT wid);
```

Invalidate the whole of window `wid`.

Equivalent to calling `wInvalidateRect` with a rectangle covering the whole of window `wid`.

Text cursor

wTextCursor

Draw a text cursor

```
VOID wTextCursor(UINT wid, .i.W_CURSOR *pcursor);
```

Present a text cursor (which is optionally flashing) in window `wid`.

There is only one text cursor per client. If a text cursor already exists, it is removed before the new cursor is positioned (you do not have to call `wEraseTextCursor` if you are moving the cursor to another position in the same or a different window).

Version 4 of the window server allows the text cursor to appear grey. This is achieved by setting `W_CURSOR_GREY` in the `flags` member of the `W_CURSOR` structure.

The `W_CURSOR` struct is defined in `wlib.h` as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT pos; /* text cursor position */
    UBYTE height; /* text cursor height */
    BYTE ascent; /* text cursor ascent */
    UBYTE width; /* text cursor width */
    UBYTE flags; /* for obloid cursor and to disable flashing */
} W_CURSOR;
```

The `height`, `ascent` and `width` members specify the height, ascent and width of the text cursor. The `flags` member may be zero or it may contain any combination of the following bit flags:

<code>W_CURSOR_OBLOID</code>	to round off the corners of the cursor
<code>W_CURSOR_NO_FLASH</code>	to disable the flashing of the cursor
<code>W_CURSOR_GREY</code>	to make the text cursor appear grey

The position of the cursor is consistent with positioning conventions for text where the top left of the cursor is ascent above the position passed to `wTextCursor`. In fact, the `x,y` position for `wTextCursor` should be the same as for drawing a text string. See the description of `gPrintText` in the *Graphics Output* chapter.

Applications which use a vertical line cursor to indicate a position between two characters should (by convention) place the cursor in the leftmost position of the character cell which is to the right of the cursor. In this case, `width` is set to 1 (or 2), and `ascent` and `height` are set according to the font currently in use.

A block or underline cursor is sometimes appropriate when using a monospaced font where `width` should be set to the width of a characters in the font.

For an underline cursor, `height` should be set to 1. To draw the underline along the baseline of the font, `ascent` should be set to zero. To draw it along the bottom line of the characters, `ascent` should be set to `(font.ascent-font.height+1)`.

The window server controls the flashing of the cursor and makes sure that it does not interfere with any drawing.

On large screen versions of the window server such as the MC, the window server automatically ensures that only the foreground text cursor is visible.

wDrawTextCursor

Draw a text cursor

```
VOID wDrawTextCursor(UINT wid, .i.W_CURSOR *pcursor);
```

A now defunct function to support older applications which leave `pcursor->flags` undefined (before version 3.5 of the window server, `pcursor->flags` was a filler for word alignment).

This function is the same as `wTextCursor` except that it ignores `pcursor->flags` and consequently does not support `W_CURSOR_OBLOID`, `W_CURSOR_NO_FLASH` or `W_CURSOR_GREY`.

New applications should use `wTextCursor`.

wEraseTextCursor

Erase a text cursor

```
VOID wEraseTextCursor(VOID);
```

Remove the caller's text cursor.

In programming terms, calling this function when there is no cursor is harmless. However, be aware that *careless* use of this function can cause problems.

Only one cursor is ever visible on the screen at any one time and `wEraseTextCursor` erases the text cursor *regardless of the window in which it appears*. An application, therefore, must ensure that a text cursor is erased at the appropriate time.

An inappropriately timed call to `wEraseTextCursor` can "steal" the cursor from the currently emphasised window.

Bitmap sequences

The window server has the ability to attach an animated sequence of bitmaps to a window. This sequence specifies the bitmap, position, bitmap source rectangle, blit mode and time to wait before advancing the sequence. The sequence is set up by the `wSetWinBitmap` call, modified by the `wChangewinBitmap` call and freed with `wFree`.

A bitmap is displayed as part of the window background, and a client can draw on top of the bitmap. Because of this the client will normally be sent a `WM_REDRAW` event telling it to redraw the window containing the bitmap every time the sequence advances. If you do not require to draw on top of the bitmap you should set the `W_WIN_NO_REDRAW` flag for the window so that your application is not slowed down by unnecessary `WM_REDRAW` events.

wSetWinBitmap **Attach bitmap sequence to window**

```
INT wSetWinBitmap(UINT wid, UINT count, .i.WS_WIN_BITMAP *pdata);
```

Attach a sequence of one to twelve bitmaps to window `wid` where `pdata` is the address of an array of `count` bitmap sequence records.

The structure of a bitmap sequence record is described by the `WS_WIN_BITMAP` struct which is defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;

typedef struct
{
    WORD bitmap; /* Bitmap ID */
    P_POINT pos; /* position of bitmap */
    P_RECT rect; /* Source rectangle in bitmap */
    UWORD mode; /* Blit mode */
    ULONG time; /* Time till next bitmap in sequence */
} WS_WIN_BITMAP;
```

where:

bitmap	is the bitmap ID. If any of the bitmaps in the sequence is freed before the bitmap sequence is freed, the window server will panic the calling client when it tries to display the freed bitmap.
pos	is the target position in the window <code>wid</code> of the top left of the bitmap
rect	is the rectangle within the bitmap to copy from
mode	is the graphics mode to use when copying the bitmap (one of <code>G_TRMODE_REPL</code> , <code>G_TRMODE_SET</code> or <code>G_TRMODE_CLR</code> or <code>G_TRMODE_INV</code>). In version 4 of the window server, a member of a bitmap sequence can be made to appear grey. This is achieved by OR'ing the flag <code>WS_WIN_BITMAP_GREY</code> into this member.
time	is the interval in tenths of a second after which the window server advances to the next bitmap (in a circular fashion). If this time is too short, the window server will hog the processor animating the bitmaps and the performance of the rest of the machine will be degraded.

Returns the ID of the bitmap sequence.

wChangeWinBitmap**Change a bitmap**

```
VOID wChangeWinBitmap(UINT bsid, UINT index, .i.WS_WIN_BITMAP *pdata);
```

Replace bitmap sequence record index in the bitmap sequence with ID bsid with the contents of pdata.

The whole of pdata must be set up even if only one of the elements of the structure is being changed.

wFree**Free a bitmap sequence**

```
VOID wFree(UINT bsid);
```

Free bitmap sequence bsid.

A bitmap sequence is automatically freed if the window it is attached to is destroyed.

Sprites

In version 4 of the window server, an animated sequence of bitmaps known as a 'sprite' can be created.

While a sprite is 'connected' to a particular window, it differs from an animated bitmap sequence in that it is not displayed as part of the window background. The window server takes care of saving and restoring the contents of the underlying window display. This can give the impression of the sprite 'floating' above the underlying display.

Each application (or client, in general) can have only one sprite at a time and each sprite consists of a sequence of up to thirteen bitmap sets. Each bitmap set consists of up to six bitmaps, three for the normal plane and three for the grey plane.

wCreateSprite**Create a sprite**

```
INT wCreateSprite(INT wid, P_POINT *pos, INT flags, INT count, .i.W_SPRITE *psprite);
```

Available in version 4 of the window server, this function creates a sprite connected to the window with ID wid based at the position specified by the P_POINT struct pointed to by pos.

The value of flags controls clipping of the sprite. If it contains the flag W_SPRITE_CLIP_CHILDREN, the sprite will be clipped by any child windows of the window to which the sprite is connected. If this flag is not set, the sprite can only be clipped by other windows or by the limits of its own window.

The use of the W_SPRITE_CLIP_CHILDREN flag needs some care. If this flag is set, the sprite should not be connected to the root window. The application's top level window is always a child window of the root which would clip or overlay a sprite.

Menus and dialog boxes are not child windows of any of the application's windows; therefore a sprite will always be clipped by menus and dialog boxes regardless of the setting of W_SPRITE_CLIP_CHILDREN.

The count parameter specifies how many bitmap sets the sprite has.

The psprite parameter points to an array of W_SPRITE structures, one for each bitmap set. Note, therefore, that there are count elements in the array.

The structure of W_SPRITE is defined as follows:

```
typedef struct
{
    WORD    bit_set;      /*(normal plane) bitmap for pixels to be set      */
    WORD    bit_clr;      /*(normal plane) bitmap for pixels to be cleared */
    WORD    bit_inv;      /*(normal plane) bitmap for pixels to be inverted*/
    WORD    bit_gr_set;   /*(grey plane)  bitmap for pixels to be set      */
    WORD    bit_gr_clr;   /*(grey plane)  bitmap for pixels to be set      */
    WORD    bit_gr_inv;   /*(grey plane)  bitmap for pixels to be set      */
    P_POINT offset;
    UWORD   time;
} W_SPRITE;
```

The `time` parameter indicates the length of time in units of 1/10th of a second that the bitmap set is to be displayed. However, this field is ignored if the sprite consists of only one bitmap set (i.e. `count` is set to one).

The members `bit_set`, `bit_clr`, and `bit_inv` contain the bitmap ids to be displayed in the normal plane using the modes `G_TRMODE_SET`, `G_TRMODE_CLR` and `G_TRMODE_INV` respectively (see the section on *Graphics contexts* in the chapter *Graphics Output*).

Similarly, `bit_gr_set`, `bit_gr_clr` and `bit_gr_inv` contain the bitmap ids to be displayed in the grey plane.

Note that setting a bitmap field to zero means that no bitmap will be used for the relevant plane and mode.

The bitmap fields can be set in any combination as appropriate. Setting all of them to zero results in the sprite being left blank for the specified time.

The `offset` parameter indicates the (x,y) offset of the top left-hand position of the bitmaps relative to the specified sprite position.

All bitmaps within a bitmap set must be the same size or else `E_GEN_ARG` will be returned.

If successful, the function returns the sprite id.

If the application (or client, in general) already has a sprite, the function panics with `panic W_PANIC_SPRITE_EXISTS`.

wSetSprite **Change a sprite's bitmaps and position**

```
INT wSetSprite(INT id, P_POINT *pos, INT index, .i.W_SPRITE *psprite);
```

Available in version 4 of the window server, this function allows the position and individual bitmap sets of an existing sprite to be changed.

The parameter `id` must be a valid sprite handle as returned from a call to `wCreateSprite`, otherwise the function panics with a `W_PANIC_SPRITE`.

If the parameter `pos` is not `NULL`, it is assumed to point to a `P_POINT` structure specifying the new position for the sprite. If the parameter is `NULL`, it is ignored and the sprite's position will be left unchanged.

If the parameter `psprite` is not `NULL`, it is assumed to point to a `W_SPRITE` type structure specifying a new bitmap set. The `index` parameter indicates which of the original bitmap sets is to be replaced; a zero value refers to the first. If the `psprite` parameter is `NULL`, then both it and the `index` parameters are ignored and the sequence of the bitmap sets will be left unchanged.

If the sprite is being enlarged, then this call can fail with an `E_GEN_MEMORY`. Changing the position of the sprite cannot fail.

wFree **Free a sprite**

```
VOID wFree(UINT id);
```

This function frees the sprite identified by the parameter `id`.

Clocks

On the Series 3, the Series 3a and an HC that is running version 3.5 and upwards of the window server, the window server can maintain a date and time clock in a window.

The various clock displays are influenced by the date and time related members of the `E_CONFIG` struct (see the description of `p_getctd` in the *Time, Timers and Dates* chapter of the *PLIB Reference* manual). On the S3 and S3a, the `E_CONFIG` struct is used to store system-wide user preferences.

On the S3, the textual components of the clock displays use the S3 system font.

On the HC, the font used for clock displays is determined by the `$WS_IF` ("Internal Font") environment variable (as with all output that is not directed at a graphics context). The "factory" setting of this environment variable selects the same font as is used on the S3.

On the S3a, the clock displays depend on the function used to draw the clock. The description and discussion of the function `wsCreateClock` (see below) applies equally to the S3 and the S3a. However, `wsCreateClock` was designed for the S3 with its 240 x 80 pixel screen. On the S3a with its 480 x 160 pixel screen, a clock drawn using `wsCreateClock` will 'work' but will appear clumsy and ungainly.

The enhanced version 4 function `wsCreateClock2` is to be preferred for applications running on the S3a.

wsCreateClock

Create a clock

```
INT wsCreateClock(UINT wid, UINT flags, INT xpos, INT ypos, INT offset);
```

In version 4 of the window server, this function is superseded by `wsCreateClock2` which includes ALL of its functionality. `wsCreateClock` should only be used if running earlier versions of the window server.

Create and maintain a clock at pixel position (xpos,ypos) in window wid. The clock displays the system time offset by offset minutes.

In all cases, (xpos,ypos) specifies the internal position of the top left corner of a rectangle containing the time display.

The appearance of the clock is controlled by the flags parameter which should be one of:

WS_CLOCK_SMALL_DIGITAL	To present a small digital clock. Displayed in the system font on the S3. Displayed in the S3 system font on the S3a (in native and compatibility mode). Displayed in the 'internal' font on the HC (as used by <code>wInfoMsg</code> and <code>wSetBusyMsg</code>). The coordinates (xpos,ypos) specify the top left corner of the rectangle containing the first character of the time display.
WS_CLOCK_MEDIUM	To display a medium sized clock in either digital (using double height characters) or analogue (using a 36x32 bitmap) form. Unless overridden by the flags described below, the selection between analogue and digital is controlled by the <code>clockType</code> member of the <code>E_CONFIG</code> struct (which should contain either <code>E_ANALOGUE_CLOCK</code> or <code>E_DIGITAL_CLOCK</code>). Whether digital or analogue, the coordinates (xpos,ypos) specify the top left corner of a 36 pixel wide by 32 pixel high rectangle containing the time display.
WS_CLOCK_LARGE_ANALOG	To display a large analogue clock using a 66x60 bitmap. The coordinates (xpos,ypos) specify the top left corner of the 66 pixel wide by 60 pixel high clock bitmap.

The above values may be qualified by ORing in combinations of the following flags:

WS_CLOCK_WITH_DATE	To also display the date. With <code>WS_CLOCK_SMALL_DIGITAL</code> , the date is displayed to the left of the time. With <code>WS_CLOCK_MEDIUM</code> , the date is displayed under the time. Not available with <code>WS_CLOCK_LARGE_ANALOG</code> .
WS_CLOCK_WITH_SECONDS	To also display seconds. With <code>WS_CLOCK_SMALL_DIGITAL</code> , the seconds field is added to the end of the display. With <code>WS_CLOCK_LARGE_ANALOG</code> , a second hand is added. Not available with <code>WS_CLOCK_MEDIUM</code> .
WS_CLOCK_FORCE_ANALOG	Valid only with <code>WS_CLOCK_MEDIUM</code> to display an analogue clock regardless of the value of the <code>clockType</code> member of the <code>E_CONFIG</code> struct.
WS_CLOCK_FORCE_DIGITAL	Valid only with <code>WS_CLOCK_MEDIUM</code> to display a digital clock regardless of the value of the <code>clockType</code> member of the <code>E_CONFIG</code> struct.
WS_CLOCK_AM_PM	To also display an am/pm indicator when the <code>timeType</code> member of the <code>E_CONFIG</code> struct is <code>E_TIME_12</code> (although the relative positions may still be altered when <code>timeType</code> is <code>E_TIME_24</code>). With <code>WS_CLOCK_SMALL_DIGITAL</code> , the am/pm indicator is displayed at the end of the time string. With a <code>WS_CLOCK_MEDIUM</code> analogue clock, the am/pm indicator is displayed to the right of the base of the clock face. With a <code>WS_CLOCK_MEDIUM</code> digital clock, the time digits are moved up and the am/pm indicator is displayed between the time digits and the position of the date. Ignored if <code>WS_CLOCK_LARGE_ANALOG</code> is set.
WS_CLOCK_CENTERED	Only applies to <code>WS_CLOCK_SMALL_DIGITAL</code> with <code>WS_CLOCK_AM_PM</code> set. Causes the time string to be centred in the wider space that allows for the am/pm indicator when the <code>timeType</code> member of the <code>E_CONFIG</code> struct is <code>E_TIME_24</code> .

If successful, the function returns the ID to use when calling `wsSetClock` and `wFree`. Otherwise, the function can leave or return `E_GEN_NOMEMORY`.

The following sample program:

```
#include <plib.h>
#include <wlib.h>

LOCAL_D WSERV_SPEC wSpec;
LOCAL_D UINT wid;

LOCAL_C VOID MainEventLoop(VOID)
{
    WS_EV event;

    for (;;)
    {
        wGetEventWait(&event);
        if (event.type==WM_REDRAW)
        {
            wBeginRedrawWinGC0(wid);
            gBorder(W_BORD_SHADOW_D|W_BORD_SHADOW_ON);
            wEndRedraw();
        }
    }
}

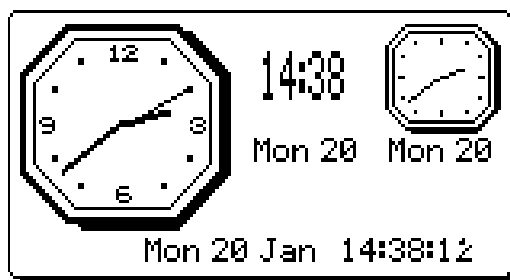
GLDEF_C VOID main(VOID)
{
    wConnect(&wSpec, 0, W_CONNECT_PRIORITY);
    wid=wCreateWindow(0,0,0,1);
    wsCreateClock(wid, WS_CLOCK_LARGE_ANALOG|WS_CLOCK_WITH_SECONDS, 4, 4, 0);

    wsCreateClock(wid, WS_CLOCK_MEDIUM|WS_CLOCK_FORCE_DIGITAL|WS_CLOCK_WITH_DATE, 4+66+6, 4, 0);

    wsCreateClock(wid, WS_CLOCK_MEDIUM|WS_CLOCK_FORCE_ANALOG|WS_CLOCK_WITH_DATE, 4+66+6+36+6, 4, 0);

    wsCreateClock(wid, WS_CLOCK_SMALL_DIGITAL|WS_CLOCK_WITH_DATE|WS_CLOCK_WITH_SECONDS, 104, 66, 0);
    wInitialiseWindowTree(wid);
    MainEventLoop();
}
```

when run on an HC, produces:



Note that the seconds field of the small digital clock is badly drawn because it changed from 11 to 12 while the screen was being captured.

The screen was in fact captured using the *scapt* program described in the *Bitmaps* section of the first chapter. It is left as an exercise for the reader to convert *pcxsave.c* (which is used by *scapt*) to take true snapshots of a changing screen³.

³One strategy is to save the screen as a bitmap to a temporary local file using `gSaveBit` and then to convert the file.

wsCreateClock2**Create a clock - Enhanced version**

```
INT wsCreateClock2( .i.WS_CREATE_CLOCK *pClock, TEXT *pfmt );
```

This function is introduced in version 4 of the window server and is an enhanced version of the `wsCreateClock` function. It is much preferred and should be used for applications designed to run on the Series 3a.

Note that this function includes the functionality of `wsCreateClock`.

The clock it creates is based on the values in the `WS_CREATE_CLOCK` structure pointed to by the parameter `pClock` and the value in the second parameter `pfmt`.

The structure `WS_CREATE_CLOCK` can be found in *wlib.h* but is defined as follows:

```
typedef struct
{
    UINT id;          /* window ID */
    UINT type;        /* clock type */
    P_POINT pos;      /* position */
    INT offset;       /*time offset */
    INT flags;        /*clock flags */
    INT font;
    INT style;
} WS_CREATE_CLOCK;
```

The function creates a clock in window `id` at position `pos` of the specified type. The clock displays the system time offset by the number of minutes specified in the `offset` member.

The possible values for `type` and `flags` include those values which can be specified in the `flags` parameter in the old `wsCreateClock` function. However, in `wsCreateClock2`, some values apply to `type` while the others apply to `flags`.

The second parameter `pfmt` and the `WS_CREATE_CLOCK` members `font` and `style` are only relevant when the type of clock being created is `WS_CLOCK_FORMATTED`. For all other types of clock `pfmt` *must* be set to `NULL`.

To summarise, `type` should be one of the following:

<code>WS_CLOCK_SMALL_DIGITAL</code>	To present a small digital clock. This clock has the appearance as described in <code>wsCreateClock</code> and is designed for the Series 3 screen. On the Series 3a in non-compatibility mode it will appear small and is not recommended.
<code>WS_CLOCK_MEDIUM</code>	To present a medium sized clock. This clock has the appearance as described in <code>wsCreateClock</code> and is designed for the Series 3 screen. On the Series 3a in non-compatibility mode it will appear small and is not recommended.
<code>WS_CLOCK_MEDIUM2</code>	To display a medium sized clock that is larger than the old medium sized clock. It behaves in a similar way to the old medium clock in that, unless overridden by the flags described below, the selection between analogue and digital is controlled by the <code>clockType</code> member of the <code>E_CONFIG</code> struct. This clock is drawn using black/white and grey. The analog clock uses a 58x51 bitmap.
<code>WS_CLOCK_LARGE_ANALOG</code>	To display a large analog clock. This clock has the appearance as described in <code>wsCreateClock</code> and is designed for the Series 3 screen. On the Series 3a in non-compatibility mode it will appear small and is not recommended.
<code>WS_CLOCK_XL_ANALOG</code>	To display an extra-large analog clock as used in alerts. This clock is drawn using black/white and grey. It uses a 99x99 bitmap.
<code>WS_CLOCK_FORMATTED</code>	To display a formatted digital clock/date. The display is controlled by the format string whose address is passed in the second parameter <code>pfmt</code> . The meaning of the format string is the same as for the PLIB function <code>p_dt2str</code> . See the PLIB manual for more detail on the syntax and meaning of this string. The font and style are specified by the <code>font</code> and <code>style</code> parameters in the <code>WS_CREATE_CLOCK</code> structure.

The clock types described above can be modified by setting combinations of the following flags in the `flags` member of the `WS_CREATE_CLOCK` structure:

WS_CLOCK_WITH_DATE	To also display the date. With WS_CLOCK_SMALL_DIGITAL, the date is displayed to the left of the time. With WS_CLOCK_MEDIUM and WS_CLOCK_MEDIUM2, the date is displayed under the time. Not available with WS_CLOCK_LARGE_ANALOG or WS_CLOCK_XL_ANALOG. Not applicable to WS_CLOCK_FORMATTED.
WS_CLOCK_WITH_SECONDS	To also display seconds. With WS_CLOCK_SMALL_DIGITAL, the seconds field is added to the end of the display. With WS_CLOCK_LARGE_ANALOG, WS_CLOCK_XL_ANALOG and the analog version of WS_CLOCK_MEDIUM2, a second hand is added. Not available with WS_CLOCK_MEDIUM or the digital version of WS_CLOCK_MEDIUM2. Not applicable to WS_CLOCK_FORMATTED.
WS_CLOCK_FORCE_ANALOG	Valid only with WS_CLOCK_MEDIUM and WS_CLOCK_MEDIUM2 to display an analogue clock regardless of the value of the <code>clockType</code> member of the <code>E_CONFIG</code> struct. Not applicable to WS_CLOCK_FORMATTED.
WS_CLOCK_FORCE_DIGITAL	Valid only with WS_CLOCK_MEDIUM and WS_CLOCK_MEDIUM2 to display a digital clock regardless of the value of the <code>clockType</code> member of the <code>E_CONFIG</code> struct. Not applicable to WS_CLOCK_FORMATTED.
WS_CLOCK_AM_PM	Valid as for <code>wsCreateClock</code> . In addition, this is not available for WS_CLOCK_MEDIUM2 and is not applicable to WS_CLOCK_FORMATTED.
WS_CLOCK_CENTERED	Only applies to WS_CLOCK_SMALL_DIGITAL with WS_CLOCK_AM_PM set. Causes the time string to be centred in the wider space that allows for the am/pm indicator when the <code>timeType</code> member of the <code>E_CONFIG</code> struct is <code>E_TIME_24</code> . Not applicable to WS_CLOCK_FORMATTED.
WS_CLOCK_BOX	Only applies to WS_CLOCK_FORMATTED. Causes graphics to be drawn enclosing the formatted clock as shown in one of the examples.
WS_CLOCK_GREY	If set, it causes those clocks which are normally drawn in black & white only, to be drawn in grey. It has no effect on those clocks which are drawn in both black/white and grey.

If successful, the function returns the ID to use when calling other 'clock' functions such as `wsSetClock` and `wFree`.

For example, when run on the S3a, the code:

```
#include <plib.h>
#include <wlib.h>

GLREF_D WSERV_SPEC wSpec;
GLREF_D UINT      wMainWid;

LOCAL_C VOID MainEventLoop(VOID)
{
    WS_EVENT event;

    for (;;)
    {
        wGetEventWait(&event);
        if (event.type==WM_KEY)
        {
            if (event.p.key.keycode==W_KEY_RETURN)
                break;
        }
    }
}
```

```
GLDEF_C INT main(void)
{
    WS_CREATE_CLOCK clock;

    wConnect(&wSpec,0,W_CONNECT_PRIORITY);
    wCompatibilityMode(0,&wSpec);
    wMainWid = wCreateWindow(0,0,0,1);
    wInitialiseWindowTree(wMainWid);

    clock.id      = wMainWid;
    clock.type    = WS_CLOCK_XL_ANALOG;
    clock.pos.x   = 200;
    clock.pos.y   = 40;
    clock.offset  = 0;
    clock.flags   = WS_CLOCK_WITH_SECONDS;

    wsCreateClock2(&clock,NULL);
    MainEventLoop();
    return(0);
}
```

displays the extra large analog clock with a seconds hand as shown opposite.



An example of a formatted clock is given next. The digital clock/date is displayed in bold, with double height and surrounded by a neat box. The code used to display the clock is as follows:

```
#include <plib.h>
#include <wlib.h>
#include <fonts.h>

GLREF_D WSERV_SPEC wSpec;
GLREF_D UINT       wMainWid;

LOCAL_C VOID MainEventLoop(VOID)
{
    WS_EVENT event;

    for (;;)
    {
        wGetEventWait(&event);
        if (event.type==WM_KEY)
        {
            if (event.p.key.keycode==W_KEY_RETURN)
                break;
        }
    }
}

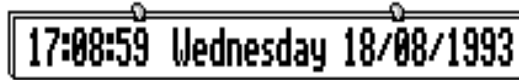
GLDEF_C INT main(void)
{
    WS_CREATE_CLOCK clock;

    wConnect(&wSpec,0,W_CONNECT_PRIORITY);
    wCompatibilityMode(0,&wSpec);
    wMainWid = wCreateWindow(0,0,0,1);
    wInitialiseWindowTree(wMainWid);
```

```
clock.id      = wMainWid;
clock.type    = WS_CLOCK_FORMATTED;
clock.pos.x   = 150;
clock.pos.y   = 70;
clock.offset  = 0;
clock.flags   = WS_CLOCK_BOX;
clock.font    = FONT_ID_SWISS_8;
clock.style   = G_STY_DOUBLE|G_STY_BOLD;

wsCreateClock2(&clock, "%h%:~m%:~s   %e   %d%/%m%/%y");
MainEventLoop();
return(0);
}
```

Take particular note of the text string forming the second parameter to `wsCreateClock2`. The format and structure of this text string governs the display of this clock.



wsSetClock

Set the clock offset

```
VOID wsSetClock(INT clock_id, INT offset);
```

Change the time offset (in minutes from the system time) of clock `clock_id` (where `clock_id` was returned from a call to `wsCreateClock`).

wFree

Free a clock

```
VOID wFree(INT clock_id);
```

Free the clock `clock_id` (where `clock_id` was returned from a call to `wsCreateClock`).

A clock is automatically freed if the window it contains is closed as a result of a call to `wCloseWindowTree`. If a clock is freed by `wCloseWindowTree`, it must not be freed a second time by a call to `wFree`.

Mouse icons

On the large screen version of the window server (such as that on the MC), each window has an associated mouse icon - as specified by the mouse icon ID in the `mouse_icon` window attribute. The `mouse_icon` window attribute is set when you create the window by calling `wCreateWindow` and it may subsequently be changed by calling `wSetWindow`.

Mouse icons can be selected from one of three categories:

- The two built-in icons, the default icon `W_WIN_MI_STANDARD` and the invisible mouse icon `W_WIN_MI_NULL`.
- ROM-based icons which are automatically loaded by the window server when it boots up.
- External mouse icons which are loaded from a file by calling `gOpenMouseIcon` (which returns the mouse icon ID).

On MC machines, the following mouse icons are built into the ROM:

<code>W_WIN_MI_STANDARD</code>	standard mouse icon.
<code>W_WIN_MI_NULL</code>	invisible mouse icon.
<code>W_WIN_MI_PUSHER</code>	hollow standard mouse icon.
<code>W_WIN_MI_TEXT</code>	text window mouse icon.
<code>W_WIN_MI_CROSS</code>	cross.

<code>W_WIN_MI_MARGIN</code>	text window margin icon.
<code>W_WIN_MI_PG_DOWN</code>	page down scroll bar icon.
<code>W_WIN_MI_PG_UP</code>	page up scroll bar icon.
<code>W_WIN_MI_VSLIDE</code>	scroll bar vertical slider icon.
<code>W_WIN_MI_HSLIDE</code>	scroll bar horizontal slider icon.
<code>W_WIN_MI_TO_BIG</code>	resize gadget expand window icon.
<code>W_WIN_MI_TO_SMALL</code>	resize gadget shrink window icon.
<code>W_WIN_MI_RESIZE</code>	resize gadget move-resize icon.
<code>W_WIN_MI_MOVE</code>	move window icon.
<code>W_WIN_MI_LEFT</code>	horizontal scroll bar move left icon.
<code>W_WIN_MI_RIGHT</code>	horizontal scroll bar move right icon.

gOpenMouseIcon

Load a mouse icon

```
INT gOpenMouseIcon(TEXT *filename, UINT index);
```

Load mouse icon index from file `filename`.

Returns the positive ID of the mouse icon to use in the `w_WINDATA` structure when calling `wSetWindow` or `wCreateWindow`.

If an error with error number `err` occurs while loading the mouse icon, the function calls `p_leave(err)` or returns `err`, depending on whether `wDisableLeaves` has been called.

If `filename` is not a full file specification, the unspecified components are taken from the window server's default path which, in practice, is always the internal drive `M:\`.

See also `gSetOpenAddress` in the next chapter for loading a mouse icon file which is embedded in another file.

wFree

Free a mouse icon

```
VOID wFree(UINT mouse_icon_id);
```

Free a mouse icon.

Any windows using the freed icon revert to using the default icon.

CHAPTER 4

GRAPHICS OUTPUT

Graphics contexts

You must create a graphics context before performing any graphics output (apart from the screen and window directed graphics described in the previous two chapters).

The use of a graphics context reduces the number of parameters required when calling graphics functions.

A graphics context contains the following:

- the ID of a drawable (a window or a bitmap) that is the ultimate recipient of the graphics output
- whether to set, clear or invert pixels when drawing lines using `gDrawLine`, `gDrawPolyLine`, `gDrawBox`, `gBorderRect`, `gBorder`, `gBorder2Rect` or `gBorder2`
- the font, style and transfer mode to use when drawing text using `gPrintText`, `gPrintClipText`, `gXPrintText`, `gPrintBoxText` or `gShadowText`.

The drawable is set once and for all when the graphics context is created (using `gCreateGC` or a variant thereof). The rest of the content can be set up when the graphics context is created and it can also be altered subsequently (using `gSetGC`).

A graphics context is set with the aid of a `G_GC` struct, which is defined in *wlib.h* as:

```
typedef struct
{
    UBYTE gmode; /* mode for line drawing */
    UBYTE textmode; /* mode for writing text */
    UBYTE style; /* style of text: bold, underline etc. */
    UBYTE flags; /* controls use of grey & double pixel mode */
    WORD font; /* ID of font to use */
} G_GC;
```

Note that the `flags` member is introduced in version 4 of the window server. Prior to version 4, the space occupied by this member was unused.

Introduced in version 4, grey is available (in one shade only) on the S3a and *Workabout*. At a software level this is implemented by introducing the concept of a plane.

There are two planes to which drawing can be directed. The normal plane can be thought of as being associated with the drawing of black while the grey plane, as its name implies, is associated with the display of grey.

Drawing is normally done to one or both planes. However, they are not entirely independent; for example, to display grey, the normal plane should be clear and the grey plane set. If a pixel in the normal plane is set, it is displayed black regardless of the grey plane setting. This is best thought of as the normal plane 'overlying' the grey plane.

Also introduced in version 4, all graphic commands can be set to perform all drawing with double sized pixels. This feature is motivated by the need to run Series 3 applications on the Series 3a; in other words, to use the S3a in S3 compatibility mode.

gmode

The gmode field is used by the line drawing functions gDrawLine, gDrawPolyLine, gDrawBox, gBorderRect, gBorder2Rect, gBorder, gBorder2, wDrawButton and wDrawButton2 and may be one of:

G_TRMODE_SET	set pixels in the line. This is the default.
G_TRMODE_CLR	clear pixels in the line.
G_TRMODE_INV	invert pixels in the line.

textmode

The textmode field controls the method of writing text in gPrintText and gPrintClipText and may be any of:

G_TRMODE_SET	where 1s in the font set bits in the destination and 0s do not change bits in the destination (used to print on to a previously cleared area). This is the default.
G_TRMODE_REPL	where 1s and 0s in the font overwrite the destination (used to print over unprepared areas).
G_TRMODE_CLR	where 1s in the font clear bits in the destination and 0s do not change bits in the destination (used to print on to a previously set area).
G_TRMODE_INV	where 1s in the font toggle corresponding bits in the destination and 0s in the source pattern do not change bits in the destination (used to print over an existing image and may be reversed by a second application).

style

The style field controls the style of text in gPrintText, gPrintClipText, gXPrintText, gPrintBoxText, gShadowText, wDrawButton and wDrawButton2. The precise mechanism of applying a style depends on the version of the window server. Prior to version 4, styles may be any combination of:

G_STY_NORMAL	text is drawn as it is in the font. This is the default.
G_STY_BOLD	text is drawn bolded (generally bolded characters are one pixel wider than normal characters).
G_STY_UNDERLINE	text is drawn underlined where each character (including space) is drawn with a horizontal line beneath its graphic.
G_STY_INVERSE	text is drawn in inverse video (where the bits in the font are inverted before drawing).
G_STY_DOUBLE	text is drawn with double height characters where each row of pixels in the character graphic is doubled up before drawing.
G_STY_MONO	text is drawn with additional space around the characters of a proportional font to turn it into a monospaced font. For this to be effective, the proportional font should be designed with monospacing in mind (as is the built in font on the Series 3).
G_STY_ITALIC	text is drawn italicised by shifting the top half of each character across by one pixel.
G_STY_SUPERSCRIPT	indicates an intention to draw in superscript, not processed by the window server.
G_STY_SUBSCRIPT	indicates an intention to draw in subscript, not processed by the window server.

Fast text printing of a fast font only works with normal style or with just G_STY_MONO. The use of any other style will cause text drawing to fall back to the slower algorithm.

In version 4 of the window server and upwards, the above styles still apply. However, in version 4, font groups are available as discussed in the section on *ROM-based fonts* in the *Introduction* chapter of this manual and in the description of gConfigureFonts later in this chapter.

If the font to which the above styles (except G_STY_SUPERSCRIPT and G_STY_SUBSCRIPT) are applied is a font group, then the window server will select the most appropriate font from within that group.

Depending on how the font group is configured and the combination of styles to be applied, some further algorithmic styling, as described above, may be necessary.

In version 4 of the window server and upwards, the following two styles are also available:

G_STY_SUPERSCRIPT2	text is drawn in superscript. This style <i>only</i> has meaning when used in conjunction with a font group configured with a suitable superscript font. If this style is used with a single font (ie not a font group), then the style is ignored.
G_STY_SUBSCRIPT2	text is drawn in subscript. This style <i>only</i> has meaning when used in conjunction with a font group configured with a suitable subscript font. If this style is used with a single font (ie not a font group), then the style is ignored.

Note that when configuring a font group to include fonts for the G_STY_SUPERSCRIPT2 and G_STY_SUBSCRIPT2 styles, the AscentAdjust field in the G_FONT_CONFIG data structure can be used to adjust the font's ascent when printing (see the gConfigureFonts function later in this chapter).

The bits G_STY_SUPERSCRIPT and G_STY_SUBSCRIPT are reserved for higher level software. For example:

```
LOCAL_C VOID DrawTextBox(INT fid,INT style,TEXT *str,P_RECT *prect,INT ascent)
{
    G_GC gc;

    gc.font=fid;
    gc.style=style;
    gSetGC(0,G_GC_MASK_FONT|G_GC_MASK_STYLE,&gc);
    if (style&G_STY_SUPERSCRIPT)
        ascent-=1;
    if (style&G_STY_SUBSCRIPT)
        ascent+=1;
    gPrintBoxText(prect,ascent,G_TEXT_ALIGN_LEFT,0,str,p_slen(str));
}
```

flags

Introduced in version 4 of the window server, the flags member is used to indicate the plane to which drawing is to be directed. It is also used to indicate whether drawing should be done in double or single pixel mode. Possible values are:

G_GC_FLAG_GREY_PLANE	graphics drawing is directed to the grey plane only
G_GC_FLAG_BOTH_PLANES	graphics drawing is directed to both the grey plane and the normal plane
G_GC_FLAG_DOUBLE	if set, all drawing is done in double pixel mode. Unsetting this puts the GC back into single pixel mode; this can be used to reverse a previous call to set double pixel mode or in windows that are in this mode by default either because the application is in compatibility mode or the window has the W_WIN_DOUBLE_PIXEL flag set.

Note that this flag should not be set if drawing is done to a backed-up window. If the window needs to be restored from the backup, then the drawing will be displayed in single pixel mode. In this situation, it is best to set the whole window into double pixel mode.

font

The font field contains the ID of the font to be used in gPrintText, gPrintClipText, gXPrintText, gPrintBoxText, gShadowText, wDrawButton and wDrawButton2.

The font ID may be that of a ROM-based font or an ID of a font that was loaded from a file by calling gOpenFont.

The ROM-based font IDs start at WS_FONT_BASE and you can use WS_FONT_BASE+1 etc for as many fonts as are built into the ROM.

By default, a graphics context is initialised with the ID of the system font (as described in the section *Text fonts* in the first chapter). You can reset a graphics context back to the system font by specifying WS_FONT_SYSTEM.

Creating a permanent graphics context

gCreateGC

Create a permanent GC

```
INT gCreateGC(UINT drawable_id, UINT field_set, G_GC *pgc);
```

Create a permanent graphics context that is assigned to the drawable `drawable_id` (the ID of a previously created window or bitmap) and select the created graphics context as current.

The `G_GC` struct is defined as:

```
typedef struct
{
    UBYTE gmode; /* G_TRMODE_SET, _CLR, _INV (line) */
    UBYTE textmode; /* G_TRMODE_SET, _CLR, _INV, _REPL (text) */
    UBYTE style; /* G_STY_NORMAL, _BOLD, _UNDERLINE, _INVERSE, _DOUBLE, _MONO,
    _ITALIC */
    UBYTE flags; /* G_GC_FLAG_DOUBLE, _GREY_PLANE, _BOTH_PLANES */
    WORD font; /* ID of font to use */
} G_GC;
```

Note that the `flags` member is introduced in version 4 of the window server. Prior to version 4, the space occupied by this member was unused.

If `field_set` is zero, `pgc` is ignored and the graphics context is created with default values (but, in this case, you should use `gCreateGC0`).

If `field_set` is non-zero it should contain a bit mask to specify which fields in `pgc` are used to set the graphics context, as follows:

<code>G_GC_MASK_GMODE</code>	to use <code>pgc->gmode</code>
<code>G_GC_MASK_TEXTMODE</code>	to use <code>pgc->textmode</code>
<code>G_GC_MASK_STYLE</code>	to use <code>pgc->style</code>
<code>G_GC_MASK_FONT</code>	to use <code>pgc->font</code>
<code>G_GC_MASK_GREY</code>	to use <code>pgc->flags</code> (looks at <code>G_GC_FLAG_GREY_PLANE</code> and <code>G_GC_FLAG_BOTH_PLANES</code> only) (version 4 only)
<code>G_GC_MASK_DOUBLE</code>	to use <code>pgc->flags</code> (looks at <code>G_GC_FLAG_DOUBLE</code> only) (version 4 only)

Returns the ID of the graphics context if successful.

If an error (with error number `err`) occurs, the function calls `p_leave(err)` or returns `err`, depending on whether `wDisableLeaves` has been called. Possible values for `err` are:

<code>E_GEN_NOMEMORY</code>	insufficient system memory
<code>E_GEN_ARG</code>	an invalid <code>gmode</code> , <code>textmode</code> , <code>style</code> or <code>font</code> was specified
<code>E_GEN_FAIL</code>	none of the grey background modes had been previously set for the window and an attempt was made to set <code>G_GC_FLAG_GREY_PLANE</code> . This error can only be returned in version 4 of the window server.

In version 2 of the window server, the function calls `p_panic` if an invalid `gmode`, `textmode`, `style` or `font` is specified.

In version 4 of the window server, setting the `G_GC_MASK_GREY` bit in `field_set` causes the window server to look at the two grey flags in the `flags` member; setting the `G_GC_MASK_DOUBLE` bit in `field_set` causes the window server to look at the double pixel flag in the `flags` member.

Setting `G_GC_FLAG_GREY_PLANE` directs graphics to the grey plane only, while setting `G_GC_FLAG_BOTH_PLANES` directs graphics to both the normal and the grey planes.

Setting `G_GC_FLAG_DOUBLE` causes all drawing to be done in double pixel mode.

gCreateGC0 **Create a permanent GC with default values**

```
INT gCreateGC0(UINT drawable_id);
```

A code saving convenience routine, equivalent to:

```
gCreateGC(drawable_id, 0, 0);
```

wFree **Free a permanent GC**

```
VOID wFree(UINT gc_id);
```

Free the permanent graphics context with ID `gc_id`.

To free a temporary graphics context, use `gFreeTempGC` **not** `wFree(WS_TEMPORARY_GC)`.

Because graphics contexts consume memory, they should be freed when they are no longer required.

Creating a temporary graphics context

gCreateTempGC **Create a temporary GC**

```
VOID gCreateTempGC(UINT drawable_id, UINT field_set, G_GC *pgc);
```

Create a temporary graphics context that is initialised with those fields from `pgc` which have their corresponding bit fields set in `field_set`.

The `G_GC` struct is defined as:

```
typedef struct
{
    UBYTE gmode; /* G_TRMODE_SET, _CLR, _INV (line) */
    UBYTE textmode; /* G_TRMODE_SET, _CLR, _INV, _REPL (text) */
    UBYTE style; /* G_STY_NORMAL, _BOLD, _UNDERLINE, _INVERSE, _DOUBLE, _MONO,
    _ITALIC */
    UBYTE flags; /* G_GC_FLAG_DOUBLE, _GREY_PLANE, _BOTH_PLANES */
    WORD font; /* ID of font to use */
} G_GC;
```

where the bit fields for `field_set` are:

<code>G_GC_MASK_GMODE</code>	corresponding to <code>gmode</code>
<code>G_GC_MASK_TEXTMODE</code>	corresponding to <code>textmode</code>
<code>G_GC_MASK_STYLE</code>	corresponding to <code>style</code>
<code>G_GC_MASK_GREY</code>	corresponding to <code>flags</code> (version 4 only)
<code>G_GC_MASK_DOUBLE</code>	corresponding to <code>flags</code> (version 4 only)
<code>G_GC_MASK_FONT</code>	corresponding to <code>font</code>

Note that the `flags` member is introduced in version 4 of the window server. Prior to version 4, the space occupied by this member was unused.

The behaviour is as for `gCreateGC`, apart from the following:

- `gCreateTempGC` does not return an ID. This improves efficiency, because the window server does not have to reply to the `gCreateTempGC`.
- `gCreateTempGC` remembers the currently selected permanent graphics context (if any), and `gFreeTempGC` reselects that same graphics context.

While a temporary graphics context exists, you may not call:

- `gCreateGC`
- `gCreateTempGC`
- `gSetGC0`
- `gSetGC` to set a graphics context other than `WS_TEMPORARY_GC`
- `wFree` (for any graphics context)
- `wBeginRedrawGC`, `wBeginRedrawGC0`, `wBeginRedrawWinGC` or `wBeginRedrawWinGC0`

gCreateTempGC0 Create a temporary GC with default values

```
VOID gCreateTempGC0(UINT drawable_id);
```

A code saving convenience routine, equivalent to:

```
gCreateTempGC(drawable_id, 0, 0);
```

gFreeTempGC Free a temporary GC

```
VOID gFreeTempGC(VOID);
```

Free the temporary graphics context created by `gCreateTempGC` or `gCreateTempGC0`.

If a permanent graphics context was current before the temporary graphics context was created, it is reselected.

Setting a graphics context

gSetGC Set a graphics context

```
VOID gSetGC(UINT gc_id, UINT field_set, G_GC *pgc);
```

Select `gc_id` as the current graphics context and alter those fields from `pgc` which have their corresponding bit fields set in `field_set`.

The `G_GC` struct is defined as:

```
typedef struct
{
    UBYTE gmode;      /* G_TRMODE_SET, _CLR, _INV (line) */
    UBYTE textmode;   /* G_TRMODE_SET, _CLR, _INV, _REPL (text) */
    UBYTE style;      /* G_STY_NORMAL, _BOLD, _UNDERLINE, _INVERSE, _DOUBLE, _MONO,
    _ITALIC */
    UBYTE flags;      /* G_GC_FLAG_DOUBLE, _GREY_PLANE, _BOTH_PLANES */
    WORD font;        /* ID of font to use */
} G_GC;
```

where the bit fields for `field_set` are:

<code>G_GC_MASK_GMODE</code>	corresponding to <code>gmode</code>
<code>G_GC_MASK_TEXTMODE</code>	corresponding to <code>textmode</code>
<code>G_GC_MASK_STYLE</code>	corresponding to <code>style</code>
<code>G_GC_MASK_GREY</code>	corresponding to <code>flags</code>
<code>G_GC_MASK_DOUBLE</code>	corresponding to <code>flags</code>
<code>G_GC_MASK_FONT</code>	corresponding to <code>font</code>

Note that the `flags` member is introduced in version 4 of the window server. Prior to version 4, the space occupied by this member was unused.

To alter a temporary graphics context with `gSetGC`, pass `gc_id` as `WS_TEMPORARY_GC`.

In version 3 and upwards of the window server, passing `gc_id` as zero will modify the current graphics context, be it temporary or permanent. In version 2, passing a `gc_id` of zero will panic the process with panic number 85 (illegal graphics context ID).

gSetGC0

Make a permanent GC current

```
VOID gSetGC0(UINT gc_id);
```

Make the permanent graphics context with ID `gc_id` current.

Equivalent to:

```
gSetGC(gc_id, 0, 0);
```

Line drawing

The functions in this section are all directed at the drawable associated with the current graphics context and are subject to the `gmode` field of the current graphics context.

gDrawLine

Draw a line

```
VOID gDrawLine(INT x1, INT y1, INT x2, INT y2);
```

Draw a line between pixel (x_1, y_1) and pixel (x_2, y_2) .

When drawing a horizontal line with y_1 equal to y_2 , the line includes the pixel with the lower x coordinate and excludes the pixel with the higher x coordinate.

Similarly, when drawing a vertical line with x_1 equal to x_2 the line includes the pixel with the lower y coordinate and excludes the pixel with the higher y coordinate.

When drawing a line in which both coordinates change, the window server turns the coordinates of the end pixels into a rectangle with a top-left internal pixel and a bottom-right external pixel. The line drawing algorithm then fills in those pixels that are intersected by a mathematical line between the corners of the mathematical rectangle on the boundary of those pixels that are in the rectangle.

gDrawPolyLine

Draw a sequence of lines

```
VOID gDrawPolyLine(INT x, INT y, WORD *plist);
```

Draw a sequence of lines as specified in the polyline `plist`, starting at the position (x, y) .

A polyline is a sequence of line drawing and movement commands. The drawing is self-relative in that each operation is relative to the end point of the last.

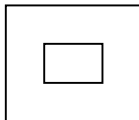
The polyline `list` has the following structure:

```
UWORD n;      /* number of word-pairs following */
WORD x1,      /* flag and x displacement */
WORD y1;      /* y displacement */
/*
Bit          5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+-----+-----+-----+
|          X-displacement          | | Move/draw flag
+-----+-----+-----+-----+
|          Y-displacement          |
+-----+-----+-----+-----+
*/
WORD x2,y2;    /* 2nd flag and x, y displacement */
...
WORD xn,yn;    /* nth flag and x, y displacement */
```

Each element of the polyline list consists of an x,y displacement, and a bit flag that is set to move and clear to draw. The move/draw flag is held in bit 0 of the x displacement word where bits 1..15 contain the actual x displacement¹. The y displacement is stored as normal.

In version 2 of the window server `plist` is limited to 61 move/draw operations.

For example, the following polyline draws a button consisting of two concentric squares of dimensions 50x50 and 30x30 as in the following diagram:



```
static WORD button[]=
{
    9,          /* 9 operations follow */
    50*2, 0,     /* draw right 50 */
    0*2, 50,     /* draw down 50 */
    -50*2, 0,    /* draw left 50 */
    0*2, -50,    /* draw up 50 */
    (10*2)|1, 10, /* move right 10, down 10 */
    30*2, 0,     /* draw right 30 */
    0*2, 30,     /* draw down 30 */
    -30*2, 0,    /* draw left 30 */
    0*2, -30     /* draw up 30 */
};
...
gDrawPolyLine(0,0,&button[0]);
```

gDrawBox

Draw a box

```
VOID gDrawBox(P_RECT *prect);
```

Draw a box composed of the outermost pixels in the rectangular block of pixels specified by `prect`.

The `P_RECT` struct is defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;
```

gBorderRect

Border a rectangle

```
VOID gBorderRect(P_RECT *prect, UINT flags);
```

This function is not available in version 2 of the window server.

Draw a border inside the rectangular block of pixels specified by `prect`, as controlled by `flags` where `flags` should be one of:

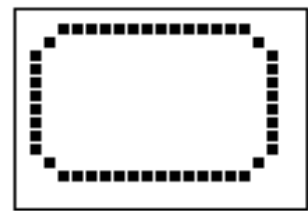
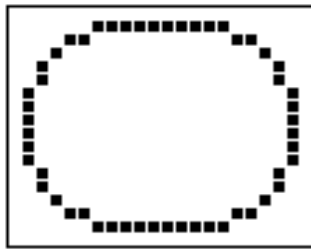
<code>zero</code>	to use corner type 2
<code>W_BORD_CORNER_1</code>	to use corner type 1
<code>W_BORD_CORNER_4</code>	to use corner type 4
<code>W_BORD_OPEN</code>	a special case used for menus on the S3 and the S3a

¹For example, you can get the x displacement into bits 1..15 by multiplying the required x displacement by 2 (which also clears bit zero). If you want to move rather than draw, you must then set bit zero.

One of the three corner types may be qualified by ORing in a combination of the following bit fields:

<code>W_BORD_CUSHION</code>	to leave a one pixel clear cushion all around
<code>W_BORD_SHADOW_S</code>	for a single shadow area (mutually exclusive with <code>W_BORD_SHADOW_D</code>)
<code>W_BORD_SHADOW_D</code>	for a double shadow area (mutually exclusive with <code>W_BORD_SHADOW_S</code>)
<code>W_BORD_SHADOW_ON</code>	the pixels in the shadow area are set - otherwise they are cleared (meaningless unless either the <code>W_BORD_SHADOW_S</code> or <code>W_BORD_SHADOW_D</code> flag is set)
<code>W_BORD_TOP_ON</code>	to draw an arrow in the top right corner
<code>W_BORD_TOP_OFF</code>	to clear an arrow in the top right corner
<code>W_BORD_BOT_ON</code>	to draw an arrow in the bottom right corner
<code>W_BORD_BOT_OFF</code>	to clear an arrow in the bottom right corner

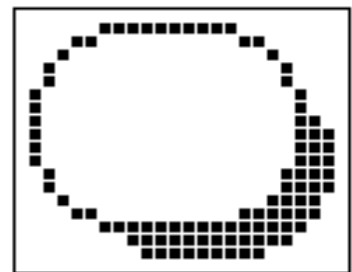
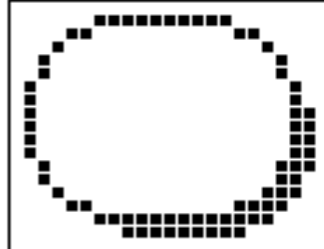
The meanings of the flags are illustrated by the following diagrams:



`W_BORD_CUSHION|W_BORD_CORNER_1`

`W_BORD_CUSHION|W_BORD_CORNER_4`

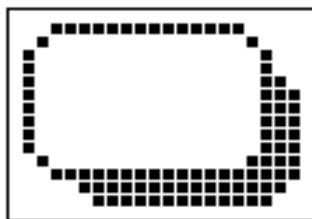
`W_BORD_CUSHION|W_BORD_SHADOW_S`



`W_BORD_CUSHION|W_BORD_SHADOW_S|W_BORD_SHADOW_ON`

`W_BORD_CUSHION|W_BORD_CORNER_4|W_BORD_SHADOW_S|W_BORD_SHADOW_ON`

`W_BORD_CUSHION|W_BORD_CORNER_4|W_BORD_SHADOW_D|W_BORD_SHADOW_ON`



`W_BORD_CUSHION|W_BORD_SHADOW_D|W_BORD_SHADOW_ON`

`W_BORD_OPEN`

`W_BORD_SHADOW_D|W_BORD_SHADOW_ON`

Although some of the effects may look gross on the above diagrams, bear in mind they will typically be used on much larger windows.

Some of the flag combinations presuppose a minimum size of rectangle.

The `W_BORD_OPEN` flag is a special case that overrides all the others. The top two corners are drawn as for `W_BORD_CUSHION|W_BORD_SHADOW_S|W_BORD_SHADOW_ON`, but the bottom two are laws unto themselves. Note also the non-appearance of lines along the bottom. This is used for the header of a Series 3 or Series 3a pull-down menu.

Except for those combinations designed to change shadows and arrows, the borders are designed to be drawn over a clear background. For example, those pixels at the perimeter of the rectangle that are obtained with `W_BORD_CUSHION` are not explicitly cleared.

The above assumes that `gmode` in the graphics context is set to `G_TRMODE_SET` (its default value).

Drawing shadows

The first line of a shadow is inset by 2 pixels at the bottom left and at the top right, and a second line (for double shadowing) is inset a further one pixel.

At the bottom right, the outside line matches the inside line, just being displaced either one or two pixels diagonally downwards and outwards.

The pixels that are set for `W_BORD_SHADOW_ON` are explicitly cleared when this flag is absent, so that the call

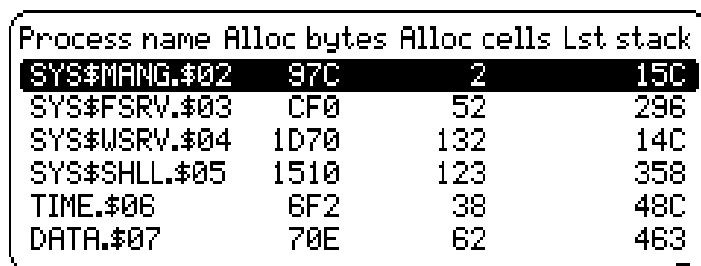
```
gBorderRect(prect, W_BORD_SHADOW_S);
```

can be used to de-emphasise a window formerly emphasised using

```
gBorderRect(prect, W_BORD_SHADOW_S|W_BORD_SHADOW_ON);
```

Arrows

The following shows the use of `W_BORD_BOT_ON` to draw an arrow in the bottom right corner on the S3 or the S3a:



Process name	Alloc bytes	Alloc cells	Lst stack
SYS\$MANG.\$02	97C	2	15C
SYS\$FSRV.\$03	CF0	52	296
SYS\$USRV.\$04	1D70	132	14C
SYS\$SHLL.\$05	1510	123	358
TIME.\$06	6F2	38	48C
DATA.\$07	70E	62	463

Remember, that when running version 4 of the window server on the Series 3a in Series 3 compatibility mode, the image will be drawn in double pixel mode!

gBorder

Border a drawable

```
VOID gBorder(UINT flags);
```

Equivalent to `gBorderRect` where the rectangle covers the entire drawable (bitmap or window).

Not available in version 2 of the window server.

gBorder2Rect

Draw a 'shadowed' border

```
VOID gBorder2Rect(INT type, P_RECT *prect, INT flags);
```

Introduced in version 4 of the window server, this function is similar to `gBorderRect` but includes the ability to draw a 3-dimensional style border.

This function can be regarded as a generalisation of `gBorderRect` as not only can it draw the 3D style borders but includes the functionality of `gBorderRect` itself.

It draws a border inside the rectangular block of pixels specified by the parameter `prect` with a style specified by the parameter `type`. The `flags` parameter 'fine-tunes' the border display.

The type can be one of:

- `W_BORDER_TYPE_0` - to draw a border in the old style as done by `gBorderRect`
- `W_BORDER_TYPE_1` - to draw a 3-dimensional grey and black border.

Note that for `W_BORDER_TYPE_1` borders, the window must be enabled for drawing grey.

The `flags` parameter can be used to fine-tune the border display. They are, to all intents and purposes, the same as those used in the function `gBorderRect` with some minor changes in meaning.

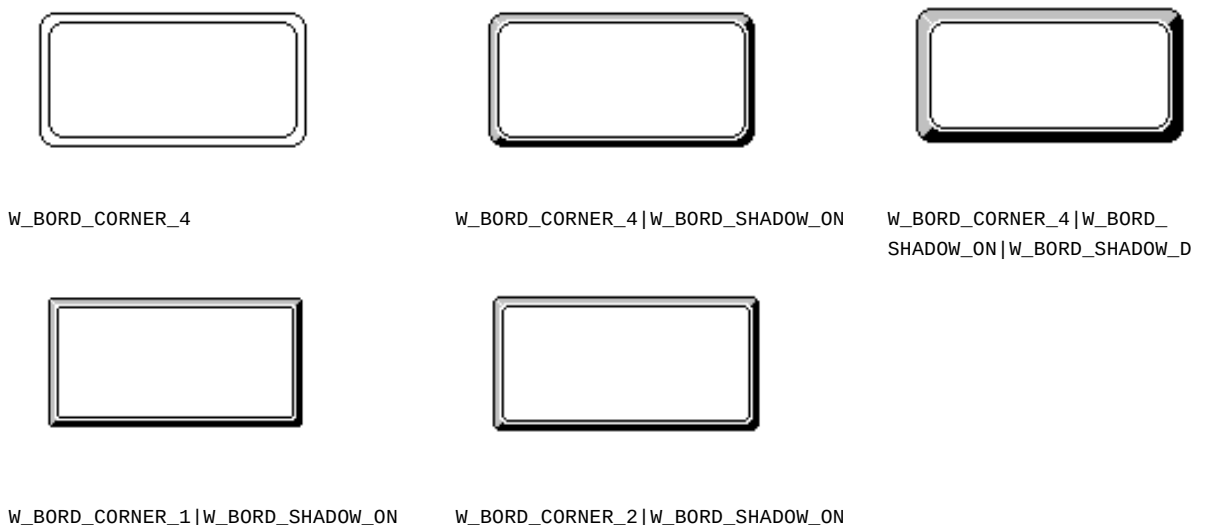
Flags should be one of:

<code>W_BORD_CORNER_2</code>	to draw a corner type 2, the same as that drawn by <code>gBorderRect</code> . This is the default corner and need not be explicitly coded. This flag applies to both border types.
<code>W_BORD_CORNER_1</code>	to draw a corner type 1, the same as that drawn by <code>gBorderRect</code> . This flag applies to both border types.
<code>W_BORD_CORNER_4</code>	to draw a corner type 4, the same as that drawn by <code>gBorderRect</code> . This flag applies to both border types.
<code>W_BORD_OPEN</code>	to draw a special corner used for menus on the S3 and S3a and is the same as that drawn by <code>gBorderRect</code> . This flag applies to <code>W_BORDER_TYPE_0</code> borders only.

The corner types can be qualified by OR'ing a combination of the following bit fields:

<code>W_BORD_CUSHION</code>	to leave a 1 pixel clear cushion right around the border. This flag applies to both border types
<code>W_BORD_SHADOW_S</code>	to draw a single shadow area. This flag applies to a <code>W_BORDER_TYPE_0</code> border only.
<code>W_BORD_SHADOW_D</code>	to draw a double shadow area for a <code>W_BORDER_TYPE_0</code> border. to set the thickness of the grey and black areas, which give the 3-D effect, to 4 pixels (compared to a default value of 2 pixels) for a <code>W_BORDER_TYPE_1</code> border.
<code>W_BORD_SHADOW_ON</code>	to set the pixels in the shadow area for a <code>W_BORDER_TYPE_0</code> border. One of <code>W_BORD_SHADOW_S</code> or <code>W_BORD_SHADOW_D</code> must also be set. If this flag is not set, the pixels are cleared. to draw a rectangle with the shadowed effect in grey and black as shown in the diagrams below for a <code>W_BORDER_TYPE_1</code> border. In drawing, it draws only the grey and black parts of the border. It leaves the white parts untouched because it assumes a pre-cleared background. If this flag is not set, it draws the outline of the border as shown and clears the area between the two outlines. This allows the shadow effects to be turned off by simply calling <code>gBorder2Rect</code> again.

The following diagrams show examples of the various 3-dimensional style borders introduced with this function. All are of type `W_BORDER_TYPE_1`. The caption below each diagram shows the flag combinations used to draw it.



gBorder2**Draw a 'shadowed' border**

```
VOID gBorder2(INT type, INT flags);
```

Introduced in version 4 of the window server, this is equivalent to `gBorder2Rect` but the rectangle is set to be the whole drawable, either the whole window or a bitmap.

gDrawObject**Draw a graphics object**

```
VOID gDrawObject(INT type, P_RECT *prect, INT flags);
```

Introduced in version 4 of the window server, this draws the scaleable graphics object specified by `type`. The object is scaled to fit inside the rectangle specified by the parameter `prect`.

The type of objects currently available are:

`G_DRAW_OBJECT_TYPE_0` a 3-dimensional box

The flags applicable to this type of object are as follows:

`W_BORD_CORNER_2` to draw the box with a corner type 2, the same as that drawn by `gBorder`. This is the default and need not be explicitly coded.

`W_BORD_CORNER_1` to draw the box with a corner type 1, the same as that drawn by `gBorder`.

`W_BORD_CORNER_4` to draw the box with a corner type 4, the same as that drawn by `gBorder`.

`W_BORD_SHADOW_D` to draw the box with double the thickness of the dark and light edge effects.

Area filling

The functions in this section act upon the drawable associated with the current graphics context.

gClrRect**Change a rectangle**

```
VOID gClrRect(P_RECT *prect, UINT mode);
```

Change all pixels within the rectangular block specified by `prect` where the change depends on `mode` as follows:

`G_TRMODE_SET` set the pixels

`G_TRMODE_CLR` clear the pixels

`G_TRMODE_INV` invert the pixels (this is reversible by another invert)

The `P_RECT` struct is defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;
```

gInvertObloid**Invert an obloid**

```
VOID gInvertObloid(P_EXTENT *pext);
```

Invert all the pixels (except the four corner pixels) in the rectangular block specified by pext.

The P_EXTENT struct is defined as:

```
typedef struct
{
    P_POINT tl;
    WORD width;
    WORD height;
} P_EXTENT;
```

Not available in version 2 of the window server.

gFillPattern**Fill a rectangle with a bitmap**

```
VOID gFillPattern(P_RECT *prect, UINT bitmap_id, UINT mode);
```

Repeatedly copy the bitmap with ID bitmap_id over the rectangular block of pixels specified by prect as many times as is necessary to fill the rectangle.

The parameter mode should be one of:

G_TRMODE_REPL	where bits in the source pattern replace corresponding bits in the destination.
G_TRMODE_SET	where 1s in the source pattern set corresponding bits in the destination (0s in the source do not change corresponding bits in the destination).
G_TRMODE_CLR	where 1s in the source pattern clear corresponding bits in the destination (0s in the source pattern do not change corresponding bits in the destination).
G_TRMODE_INV	where 1s in the source pattern toggle corresponding bits in the destination (0s in the source pattern do not change corresponding bits in the destination).

A larger bitmap will give improved performance. If the pattern is all ones or all zeros, gFillPattern is equivalent to gClrRect, but less efficient.

There is a built in grey bitmap with the handle WS_BITMAP_GREY and size WS_BITMAP_GREY_SIZE_X by WS_BITMAP_GREY_SIZE_Y. This is not true grey as found on the Series 3a but is a pseudo-grey, built up from a pattern of alternate black and white pixels (ie pixels alternately set on and off in the normal plane).

The parameter bitmap_id can also be the ID of a backed-up window to refer to the backup bitmap (but not in version 2).

For example to fill the current drawable with the grey cheque board pattern:

```
LOCAL_C VOID GreyWin(VOID)
{
    P_RECT rect;

    rect.tl.x=0;
    rect.tl.y=0;
    rect.br.x=10000;
    rect.br.y=10000;
    gFillPattern(&rect, WS_BITMAP_GREY, G_TRMODE_REPL);
}
```

In version 4 of the window server, *when* bitmap_id *refers to a backed-up window*, the function has a special way of handling grey:

- In normal mode, the black plane only is copied from the source to the destination.
- In G_GC_FLAG_GREY_PLANE mode, the grey plane from the source is selected and copied to the grey plane of the destination. If the source only has one plane, then that is used as the source.
- In G_GC_FLAG_BOTH_PLANES mode, and with a two plane source, both planes are copied to their respective destination planes. If only one plane exists, then it will be copied to both planes of the destination.

Note that this does *not* apply when copying from any other form of bitmap - in all other cases the black and grey planes must be copied by two separate calls to gFillPattern.

Text fonts

Fonts are generally described in terms of:

- ascent, descent, and vertical leading
- width and horizontal leading
- low character and high character (the range of ASCII values covered).

To clarify the meaning of some of these terms, refer to the following diagram:



The string 'Specify' has been printed at a point (x, y) which is indicated in the diagram by a pair of partially drawn lines. The horizontal of the pair of lines is known as the baseline.

The descent of a character is the number of pixels that it extends below the baseline. Thus in the font shown, 'S' has a zero descent whereas 'p' has a descent of 1.

The ascent of a character is the number of pixels that it extends above the baseline. In the font shown, 'S' has an ascent of 7 whereas 'p' has an ascent of 5. The ascent quoted for a font is always the maximum ascent of all the characters in the font, and likewise for the descent. The sum of the ascent and the descent is the height of the font.

Vertical leading is the number of additional pixels that separate adjacent lines of text, over and above the vertical height of each line of text. This is usually at least one, to prevent the highest ascent of one line ever joining up with the lowest descent of the line above. On occasion, however, such joining up may actually be intended, for example to support box-drawing via the IBM graphics characters in the extended portion of a font. In any case, it should be noted that, strictly speaking, vertical leading is not a characteristic of a font as such; rather, it describes how a font is used on a particular occasion. To complicate matters, different uses of a font will in fact often have different vertical leading.

Horizontal leading is the number of pixels that separate adjacent characters in a line of text.

The term width is, unfortunately, possessed of two subtly different meanings. The more useful of the two meanings is that the width of some text is the number of pixels from the start of that text to where a piece of text following on from the first one would start. This is the value returned by window server inquiry functions such as `gTextwidth`. Thus the width of the character 'S' in the above font is the number of pixels from the start of the 'S' to the start of the following character ('p'), namely 6. With this meaning, all the characters shown in the diagram have width 6, except for 'i' (4) and 'f' (5).

The second of the two meanings of width discounts the horizontal leading, so that, by this reckoning, the width of 'S' is just 5 pixels. This latter meaning of width is sometimes referred to as basic width.

There is an important difference between horizontal and vertical leading: horizontal leading is always supplied as part of the font; on the other hand the amount of vertical leading (if any) to be applied is up to the user of the font. Stated otherwise, applications have no choice about horizontal leading, but do have choice over vertical leading. For this reason, the simple term leading is commonly used to denote what is here being called vertical leading (since horizontal leading is usually just taken for granted).

The width of the widest character in a font is called the maximum width, and the width of the numeric character '0' is called the numeric width, or sometimes the column width. (In any well designed font, all numeric characters will have the same width).

Occasionally, a font is described in terms of its body cell. The font depicted has a basic body cell of 5 by 8 (which excludes both vertical and horizontal leading), and a corresponding expanded body cell of 6 by 9.

Not all the character codes within the range need have a representation within the font. When drawn, these characters will be represented by the last character in the font.

An application can switch between many different fonts of varying sizes. Each Graphics Context may have a different font. The appearance of text can also be altered by various style options such as underline or bold (these are independent of the font).

Fonts are generally proportional, ie the characters within the font can be of differing width. It is left to an application to determine the width of text and make decisions about layout accordingly.

Information is provided to allow fonts of differing heights to be aligned vertically about a baseline. When characters are drawn, the y-coordinate given corresponds to the baseline of the character. If a line of text is drawn using characters from fonts of different height, this ensures that the text lines up vertically.

gOpenFont

Open a font

```
INT gOpenFont(TEXT *filename);
```

Load the font from `filename` and return the font ID.

If an error (with error number `err`) occurs (for example, the file does not exist or if it is not a valid font file), the function calls `p_leave(err)` or returns `err`, depending on the `wDisableLeaves` state.

To use the font for text drawing functions, the font ID must be assigned to a graphics context (using `gSetGC`).

Note that the parameter `filename` is ultimately passed to `p_open` by the window server process - not the client process. If `filename` is not a full file specification, the unspecified components are taken from the window server's default path which, in practice, is always the internal drive `M:\`. This is unlikely to meet the requirements of a finished product so `filename` should specify the drive and directory as well as the file name. Typically, you might place a font file in the same location as the application file in which case you would use something like:

```
GLREF_D TEXT *DatCommandPtr;

LOCAL_C INT OpenFont(TEXT *name)
{
    TEXT full[P_FNAME_SIZE];

    f_parse(name, DatCommandPtr, &full[0], NULL);
    gOpenFont(&full[0]);
}
```

See also `gSetOpenAddress` for loading a font file which is embedded in another file.

gSetOpenAddress

Set pos to open font/bitmap/mouse icon

```
VOID gSetOpenAddress(UINT mode, ULONG pos);
```

Set the file position for the next call to `gOpenFont`, `gOpenFontIndex`, `gInitBit`, `gOpenBit` or `gOpenMouseIcon` as a function of `mode` and `pos`.

The possible values of `mode` are defined by constants of the form `G_OPEN_MODE_XXX` where `XXX` is one of:

<code>OFFSET</code>	to indicate that <code>pos</code> is the file position of the data
<code>WORD_PTR_OFFSET</code>	to indicate that <code>pos</code> is the file position of a <code>UWORD</code> containing the file position of the data
<code>LONG_PTR_OFFSET</code>	to indicate that <code>pos</code> is the file position of a <code>ULONG</code> containing the file position of the data
<code>NORMAL</code>	to cancel the effect of any previous unused call to <code>gSetOpenAddress</code> (<code>pos</code> is ignored)

The effect of the call only lasts until the next call to `gOpenFont`, `gOpenFontIndex`, `gInitBit`, `gOpenBit` or `gOpenMouseIcon` so you wouldn't normally need `G_OPEN_MODE_NORMAL` - especially as a call to `gSetOpenAddress` would normally occur immediately before the `gOpenxxx` call it is intended to effect.

The data at the effective file position should be the entire contents of the normal font, multiple font, bitmap or mouse icon file - including all headers.

wFree**Free a font**

```
VOID wFree(UINT font_id);
```

Free a previously loaded font.

gFontInfo**Get font information**

```
INT gFontInfo(UINT font_id, UINT style, G_FONT_INFO *pinfo);
```

Write information about the font with ID `font_id` as modified by the text style `style` to the `G_FONT_INFO` struct at `pinfo` where the `G_FONT_INFO` struct is defined in *wlib.h* as:

```
typedef struct
{
    UWORD low_ch;          /* lowest character code in font */
    UWORD high_ch;         /* highest character code in font */
    UWORD height;          /* height of font */
    UWORD descent;         /* height of bottom part of a character */
    UWORD ascent;          /* height of top part of a character */
    UWORD numeric_width;   /* width of the '0' character */
    UWORD max_width;       /* width of widest character in the font */
    UWORD flags;           /* flags specifying information about the font */
    TEXT name[16];         /* Text name of the font */
} G_FONT_INFO;
```

The field `numeric_width` is actually the width of the '0' (zero) character, however any well designed font will have all its numeric characters the same width. If a font is monospaced, all characters are of width `max_width`.

The field `max_width` is set by the font designer when the font is created. It is not necessarily the widest character in the font, since the font designer will normally exclude any special characters that are rarely used - `max_width` is normally the width of 'M' or 'W'.

The fields `numeric_width`, `max_width`, `height`, `descent` and `ascent` can be modified from their base values by the style given.

The `flags` field consists of the following bit fields:

<code>G_FONT_FLAG_ASCII</code>	the font contains the standard ASCII character set
<code>G_FONT_FLAG_CP850</code>	the font contains the IBM code page 850 character set
<code>G_FONT_FLAG_BOLD</code>	the font is designed to look bolded
<code>G_FONT_FLAG_ITALIC</code>	the font is designed to look italic
<code>G_FONT_FLAG_SERIF</code>	the font character graphics have serifs

Returns zero if successful or, if `font_id` is invalid, it calls `p_leave(E_GEN_NOFONT)` or returns `E_GEN_NOFONT`, depending on whether `wDisableLeaves` has been called.

To get information on the system font, `font_id` may be set to `WS_FONT_SYSTEM`.

On version 2 of the window server you should not call `gFontInfo` with an invalid `font_id` (if you do, `gFontInfo` writes garbage to `pinfo`).

gTextWidth**Get text width**

```
INT gTextWidth(UINT font_id, UINT style, TEXT *pbuf, UINT len);
```

Return the width in pixels of the `len` characters at `pbuf` when drawn with font `font_id` and style `style`.

Any character code in `pbuf` that does not have a corresponding character graphic in the font is taken to have the width of the last character in the font.

Note that the style can affect the width returned because bolding, italicising, mono-spacing etc widens each character.

To get the text width when using the system font, `font_id` may be set to `WS_FONT_SYSTEM`.

If `font_id` is invalid, the function calls `p_leave(E_GEN_NOFONT)` or returns `E_GEN_NOFONT`, depending on whether `wDisableLeaves` has been called.

gTextCount**Clip text to pixel width**

```
INT gTextCount(UINT font_id, UINT style, TEXT *pbuf, UINT len,UINT *pwidth);
```

Return the number of characters from pbuf (up to len) that will fit in their entirety in *pwidth pixels when drawn with font font_id and style style. Also overwrite *pwidth with the remaining width.

Any character code in pbuf that does not have a corresponding character graphic in the font is taken to have the width of the last character in the font.

Note that the style can affect the width returned because bolding, italicising, mono-spacing etc widens each character.

When using the system font, font_id may be set to WS_FONT_SYSTEM.

If font_id is invalid, the function calls p_leave(E_GEN_NOFONT) or returns E_GEN_NOFONT, depending on whether wDisableLeaves has been called.

gGetWidthTable**Get a font width table**

```
INT gGetWidthTable(UINT font_id, UINT style, UBYTE *ptab);
```

Generate a UBYTE array of character pixel widths in *ptab for font font_id and style style.

The first byte in *ptab is the width of the character with code low_ch and the length of the array written is high_ch-low_ch+1 (where low_ch and high_ch are from the font's G_FONT_INFO struct).

The function is provided to speed optimise calculations based on character widths - such as those performed by gTextwidth and gTextCount.

Any character code that does not have a corresponding character graphic in the font is taken to have the width of the last character in the font.

Note that the style can affect the width returned because bolding, italicising, mono-spacing etc widens each character.

When using the system font, font_id may be set to WS_FONT_SYSTEM.

Returns zero if successful or, if font_id is invalid, it calls p_leave(E_GEN_NOFONT) or returns E_GEN_NOFONT, depending on whether wDisableLeaves has been called.

wSetSystemFont**Set the system and internal fonts**

```
INT wSetSystemFont(INT mode,INT handle,UINT style);
```

Available in version 4 of the window server, this function may be used on the Series 3a and Workabout to set up the system and internal fonts.

The mode parameter is used to indicate which font type is to be the target of the set up. This parameter can take one of the following values:

W_SYSTEM_FONT_S3B	sets up the native system font
W_SYSTEM_FONT_S3	sets up the Series 3 compatibility mode system font
W_SYSTEM_FONT_INTERNAL_S3B	sets up the native internal font
W_SYSTEM_FONT_INTERNAL_S3	sets up the Series 3 compatibility mode internal font

The handle parameter references the font to be used as the source for the setup. This can be:

- the handle returned for a loaded font
- the handle of one of the built-in fonts
- the handle of a font group created by gConfigureFonts

The style parameter can be used to modify the style of text when setting up either of the two internal font types. This parameter is ignored if setting up any of the other font types. The style parameter adopts the same values as those applicable to the style field in the graphics context.

The function returns zero if successful, `E_GEN_NOFONT` if an invalid font handle is supplied or `E_GEN_ARG` if an invalid mode is supplied. Alternatively, it calls `p_leave` if `wDisableLeaves` has been called.

Note that at the time of writing, there is no way of notifying applications that the system font has changed.

gOpenFontIndex **Open a font from a multiple font file**

```
INT gOpenFontIndex(TEXT *fname,UINT index)
```

Available in version 4 of the window server, this function is similar to `gopenFont` except that it allows a font to be loaded from a file containing more than one font.

The `index` parameter is used to indicate which font within the file is to be loaded. The first font in the file corresponds to an index of zero, while the second font corresponds to an index value of one and so on.

Single-font files can be opened by setting `index` to zero.

Multiple-font files are invalid to older versions of the window server.

If `index` refers to a font file beyond the final one in the file, error `E_FILE_EOF` is reported.

If an error (with error number `err`) occurs, the function calls `p_leave(err)` or returns `err`, depending on whether `wDisableLeaves` has been called.

Multiple-font files can be created by versions 2.00 upwards of the font compiler `WSFCOMP`.

gConfigureFonts **Configure a font group**

```
INT gConfigureFonts(INT count,G_FONT_CONFIG *pfcfg[])
```

Available in version 4, this function is used to create a font group and return its id.

A font group is a compromise solution to the problem of deciding whether to use the window server's algorithmic method of obtaining a style (see *style* subsection of the *Graphics context* section) or to use a font designed specifically for the purpose. This is even more important where the situation is ambiguous. For example, given a bold font and an italicised font, if the application wishes to print in bold *and* italics, does it choose to apply the bold style to the italicised font or apply the italic style to the bold font?

Designing a font for every combination of style is impractical while the result of applying the window server's algorithmic method may not always give satisfactory results.

Essentially, a font group is a list of font id's and style combinations.

When an application wishes to print text, typically it will specify a (group) font ID and a combination of styles to be applied. In response, the window server scans down each entry in the list searching for the best match. It then uses the font in this entry for printing.

The font group is specified by the parameter `pfcfg` which points to an array of `G_FONT_CONFIG` structures. The array has `count` elements. Each element in the array is, in effect, an entry in the list discussed above. The `G_FONT_CONFIG` structure is defined as follows:

```
typedef struct
{
    UINT  FontId;           /*Replacement Font           */
    UINT  RepStyles;        /*Styles needing to be replaced*/
    UINT  FontStyles;       /*Styles used with this Font  */
    UINT  AscentAdjust;     /*Adjustment to Font's ascent */
} G_FONT_CONFIG;
```

The order of the elements in the array is important because of the way the search algorithm works. This proceeds as follows:

1. The window server initially chooses the base font as the 'best'. In other words, the first entry in the array.
2. Initially, each entry in the array is scanned, starting with the first, until one is found where the `RepStyles` field contains styles which are a *subset* of the styles to be printed.

3. Subsequently, the scan continues, searching for entries whose `RepStyles` field contains styles which are a subset of the styles to be printed *and* which also (as a minimum) contain the styles of the current 'best' entry. Where such an entry is found, this becomes the 'best'.
4. The process, numbered 3 above, is repeated until the array is exhausted, The window server uses the font corresponding to the `FontId` of the 'best' entry in the array.

Having found a suitable font, the window server then applies those styles which the application wants printed but which are not specified in the `RepStyles` field. For example, suppose the application wants bold *and* italic printed. Suppose also that the 'best' entry in the array specifies only the bold style in the `RepStyles` field. The window server will use the `FontId` as the font to be used and will apply the italic style to this font.

In addition, the `FontStyles` member of each entry in the array specifies styles which are also to be applied to the chosen font when it is drawn.

The `AscentAdjust` field specifies an adjustment to be made (positive or negative) to the font's ascent when printing. This is primarily of use for superscript and subscript fonts. See the `G_STY_SUBSCRIPT2` and `G_STY_SUPERSCRIPT2` styles at the beginning of this chapter.

gReadFontHeader

Read a font header from a file

```
INT gReadFontHeader(TEXT *fname, INT index, UBYTE *pbuffer)
```

Available in version 4 of the window server, this function reads up to a maximum of `FONT_MAX_HEADER_LEN` bytes of font data from the font referenced by `fname` and `index` into the buffer pointed to by `pbuffer`.

`fname` references the font file while the `index` parameter indicates the actual font within the file. For a single font file, `index` must be set to 0; for a multiple font file, an `index` value of zero refers to the first file, an `index` value of one refers to the second file and so on.

If the call is successful, the function returns the length of data actually read.

If an error (with error number `err`) occurs (for example, the file does not exist or it is not a valid font file), the function calls `p_leave(err)` or returns `err`, depending on the `wDisableLeaves` state.

gReadFontGroupHeader

Read a font group header from a file

```
INT gReadFontGroupHeader(TEXT *fname, UBYTE *pbuffer)
```

Available in version 4 of the window server, this function reads up to a maximum of `FONT_MAX_HEADER_LEN` bytes of font header data from the font file referenced by `fname` into the buffer pointed to by `pbuffer`.

In the first word of the buffer, the function places the number of fonts contained in the file. This word is followed by the group header. This means that the actual maximum length of header data that can be stored in the file is `FONT_MAX_HEADER_LEN - 2` bytes.

If the call is successful, the function returns the length of data actually read.

If an error (with error number `err`) occurs (for example, the file does not exist or it is not a valid font file), the function calls `p_leave(err)` or returns `err`, depending on the `wDisableLeaves` state.

See the description of `wsfcomp` for information on how to add header data.

Text output functions

The functions in this section are all directed at the drawable associated with the current graphics context and are all subject to the `style` and `font` fields of the current graphics context.

The functions `gPrintText` and `gPrintClipText` are also subject to the `textmode` field. The functions `wDrawButton`, and `wDrawButton2` are also subject to the `gmode` field.

In version 4 of the window server, the plane to which drawing is directed in the current graphics context will affect the 'colour' of the display.

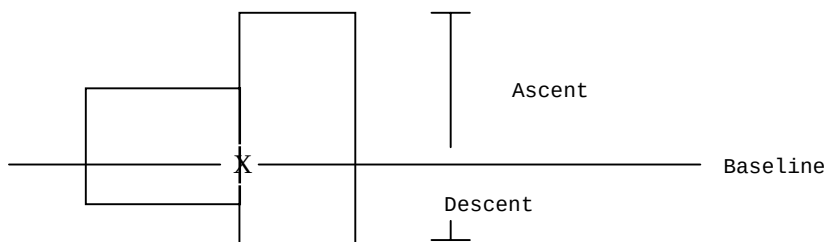
gPrintText**Print text**

```
VOID gPrintText(INT x, INT y, TEXT *pbuf, UINT len);
```

Print the `len` characters at `pbuf` horizontally from pixel position `x, y` where `len` must not be greater than `WS_MAX_PRINT_TEXT_LEN`.

The first character graphic is positioned such that its leftmost pixel that is just above the baseline is over pixel `(x, y)` of the drawable. (The baseline is the mathematical line between the upper ascent pixels and the lower descent pixels that make up the height of the font.)

The text is printed relative to the baseline so that the characters in a line of text that contains different fonts (with potentially different ascents and descents) line up correctly, as illustrated by the following diagram:



`X` is the printing position.

The text is printed according to the `textmode`, style and font in the current graphics context.

Any character code in `pbuf` that does not have a corresponding character graphic in the font is printed as the last character in the font.

Note that `gPrintText` performs no special processing on control characters (ie characters with a code that is less than 32) and if they are not represented in the font then they will also be printed as the last character in the font.

gPrintClipText**Print clipped text**

```
INT gPrintClipText(INT x, INT y, TEXT *pbuf, UINT len, UINT clip_width);
```

Similar to `gPrintText` except that it only draws as many characters as will fit inside `clip_width`.

Returns the number of characters actually printed.

gPrintBoxText**Print text in a box**

```
VOID gPrintBoxText(P_RECT *prect, UINT ascent, UINT align, INT margin, TEXT *pbuf, UINT len);
```

Print the `len` characters at `pbuf` with a fixed `textmode` of `G_TRMODE_REPL` within the rectangular block of pixels specified by `prect`, clearing (or setting) those pixels that are not replaced by characters from the font (`len` must not be greater than `WS_MAX_PRINT_BOX_TEXT_LEN`).

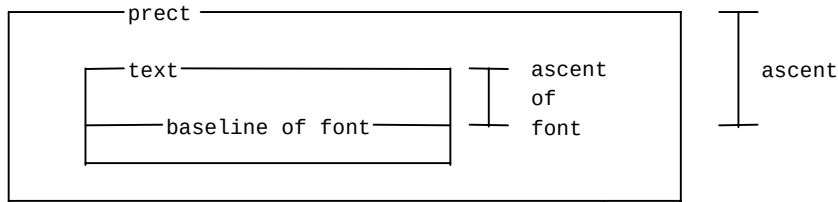
If necessary, the drawing is pixel-clipped to the rectangle defined by `prect`.

The `P_RECT` struct is defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;
```

The characters are positioned vertically such that there are ascent pixels between the top of the rectangle and the base line of the characters (unless you intend to clip the top off the text, ascent should be greater than or equal to the current font's ascent) as illustrated by the following diagram:

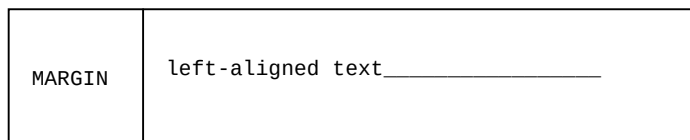


The excess space around the text is cleared or set depending on the `inverse` bit in the style.

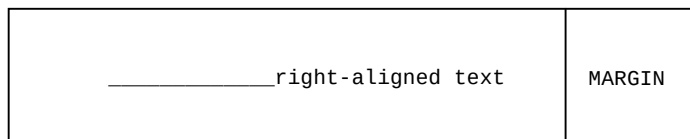
The text may be left or right aligned, or centred within `prect` depending on whether `align` is `G_TEXT_ALIGN_LEFT`, `G_TEXT_ALIGN_RIGHT` or `G_TEXT_ALIGN_CENTRE`.

The parameter `margin` is used to clear extra space to the left or right of the text where `margin` is interpreted as follows:

`align = G_TEXT_ALIGN_LEFT`



`align = G_TEXT_ALIGN_RIGHT`

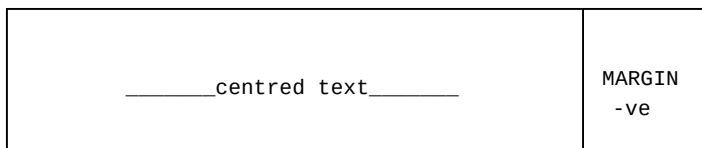


For centred text, the margin can be placed either to the right or to the left of the text according to the sign of `margin`, as follows:

`align = G_TEXT_ALIGN_CENTRE, margin >= 0`



`align = G_TEXT_ALIGN_CENTRE, margin < 0`



The text is printed according to the style and font in the current graphics context (but not `textmode`).

Using `gPrintBoxText` avoids the flicker that is generated between a separate clear and print.

gXPrintText

Print text with embellishment

```
VOID gXPrintText(INT x, INT y, TEXT *pbuf, UINT len, UINT flags);
```

Print the `len` characters at `pbuf` horizontally from pixel position `x, y` with highlighting in a style that depends on the value of `flags` (`len` must not be greater than `WS_MAX_PRINT_TEXT_LEN`).

The text is printed according to the style and font in the current graphics context (but not textmode).

Equivalent to calling `gPrintText` with a fixed textmode of `G_TRMODE_REPL` followed by some embellishment depending on the value of `flags`. Possible flag values are illustrated below for the string 'Specify', in each case printed with the same value of `x` and `y`.

Zero (no embellishment).

The partially drawn lines indicate the point to which the coordinates `x` and `y` apply.



`G_XP_INV_BLOCK`



`G_XP_INV_OBLOID`



`G_XP_INV_BLOCK | G_XP_REduced`



`G_XP_INV_OBLOID | G_XP_REduced`



`G_XP_UND_BLOCK`



`G_XP_UND_BLOCK | G_XP_REduced`



All the reduced forms are intended for use on text strings with no characters having descenders, such as numbers and upper case letters.

A zero flags is equivalent to `gPrintText` with a textmode of `G_TRMODE_REPL` and may be used to cancel an embellishment set up previously.

Not available in version 2 of the window server.

In version 4 of the window server, by setting the graphics context to draw to the grey plane and making sure that the window is enabled for drawing grey, all of the text and embellishments in the above examples will be drawn in grey.

gShadowText**Print shadowed text**

```
VOID gShadowText(INT posx,INT posy, G_SHADOW *pshadow,TEXT *ptxt,INT len);
```

Introduced in version 4 of the window server, this function prints the `len` characters at `ptxt` with a shadowed effect from pixel position `posx`, `posy` where `len` must not be greater than `WS_MAX_PRINT_TEXT_LEN`.

It uses the current font and style (ie the current graphics context) but ignores the current `textmode`.

The `pshadow` parameter must point to a structure of type `G_SHADOW` which is defined as follows:

```
typedef struct
{
    UBYTE BodyColour;
    UBYTE ShadowColour;
    UBYTE LightColour;
    UBYTE filler;
    UWORD Flags;
    WORD ShadowSizeX;
    WORD ShadowSizeY;
    WORD LightSizeX;
    WORD LightSizeY;
    WORD Spacing;
} G_SHADOW;
```

The shadow is always placed at the bottom right of the text while the lighting effect is always placed at the top left of the text.

The size of the shadow effect is specified by `ShadowSizeX` and `ShadowSizeY`. The size of the lighting effect is specified by `LightSizeX` and `LightSizeY`.

The 'colours' for the body, shadow and light can be one of black, grey, white or none by setting the `G_SHADOW` members `BodyColour`, `ShadowColour` and `LightColour` to one of `G_COLOUR_BLACK`, `G_COLOUR_GREY`, `G_COLOUR_WHITE` or `G_COLOUR_NONE`.

The display can consist of either a single copy of the text giving an impression of the text floating above the shadow or a solid block linking the text to the background. By default, the floating style shadow is used. The solid block effect is achieved by setting the `Flags` member to `G_SHADOW_SOLID`.

The gap between characters can be set by giving the `spacing` member a suitable value. This is useful if a character's shadow effects are not to overlap the following character.

The following picture shows four examples of the effect of using `gShadowText`:



In all examples, `ShadowSizeX` was set to 6, `ShadowSizeY` was set to 6, `LightSizeX` was set to 2 and `LightSizeY` was set to 2.

Spacing was set to 6 to allow sufficient space between the characters so that the shadow effects could be seen.

Looking at each example from left to right, the following values were used:

1. BodyColour set to G_COLOUR_BLACK
ShadowColour set to G_COLOUR_GREY
LightColour set to G_COLOUR_NONE
2. BodyColour set to G_COLOUR_BLACK
ShadowColour set to G_COLOUR_GREY
LightColour set to G_COLOUR_NONE
Flags set to G_SHADOW_SOLID
3. BodyColour set to G_COLOUR_BLACK
ShadowColour set to G_COLOUR_GREY
LightColour set to G_COLOUR_WHITE
Flags set to G_SHADOW_SOLID
4. BodyColour set to G_COLOUR_WHITE
ShadowColour set to G_COLOUR_BLACK
LightColour set to G_COLOUR_GREY
Flags set to G_SHADOW_SOLID

wDrawButton

Draw a text button

```
VOID wDrawButton(P_RECT *prect, TEXT *pstr, UINT depressed);
```

Draw a button within rectangle prect containing the zero terminated string pstr. If depressed is TRUE, draw the button with the text and the box displaced to give a 3D illusion of a depressed button.

The length of pstr must not be greater than W_DRAW_BUTTON_MAX_LEN (240).

The text is printed according to the style and font in the current graphics context. Both the box lines and the text string are drawn according to the gmode (the value of textmode is ignored).

The P_RECT struct is defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;
```

The intended use of wDrawButton is illustrated by the following example:

```
#include <plib.h>
#include <wlib.h>

LOCAL_D WSERV_SPEC wSpec;
LOCAL_D UINT wMainWid;
LOCAL_D UINT FontID;
LOCAL_D UINT FontStyle;
LOCAL_D G_FONT_INFO FontInfo;

LOCAL_C VOID SetFont(INT fid,INT style)
{
    gFontInfo(FontID=fid,FontStyle=style,&FontInfo);
}

LOCAL_C VOID SetGC(VOID)
{
    G_GC gc;

    gc.font=FontID;
    gc.style=FontStyle;
    gSetGC(0,G_GC_MASK_FONT|G_GC_MASK_STYLE,&gc);
}
```

```

LOCAL_C VOID DrawButton(INT state)
{
    P_RECT rect;

    rect.tl.x=20;
    rect.tl.y=(40-2)-FontInfo.height;
    rect.br.x=140;
    rect.br.y=(40+2)+FontInfo.height;
    wDrawButton(&rect,"Press any key",state);
}

LOCAL_C VOID MainEventLoop(VOID)
{
    WS_EV event;

    for (;;)
    {
        wGetEventWait(&event);
        if (event.type==WM_REDRAW)
        {
            wBeginRedrawWinGC0(wMainWid);
            gBorder(W_BORD_SHADOW_D|W_BORD_SHADOW_ON);
            SetGC();
            DrawButton(FALSE);
            wEndRedraw();
            continue;
        }
        if (event.type==WM_KEY)
        {
            gCreateTempGC0(wMainWid);
            SetGC();
            DrawButton(TRUE);
            wFlush();
            p_sleep(51);
            DrawButton(FALSE);
            gFreeTempGC();
        }
    }
}

GLDEF_C VOID main(VOID)
{
    wConnect(&wSpec,0,W_CONNECT_PRIORITY);
    wMainWid=wCreateWindow(0,0,0,1);
    wInitialiseWindowTree(wMainWid);
    SetFont(WS_FONT_SYSTEM,G_STY_BOLD);
    MainEventLoop();
}

```

wDrawButton2

Draw a text button

```
VOID wDrawButton2(INT type, P_RECT *prect, TEXT *ptext, UINT state);
```

Introduced in version 4 of the window server, this function not only draws the new style Series 3a buttons, but also the old style Series 3 types.

The parameter `prect` points to a `P_RECT` structure that specifies a rectangle that fully encloses the button in all of its states.

The `ptext` parameter specifies a zero terminated string to be drawn inside the button in the current font and style. It is the responsibility of the caller to make sure that the text will fit inside the button; there is no clipping of text. The maximum length of text is `W_DRAW_BUTTON_MAX_TEXT`.

The parameter `type` indicates which style of button is to be drawn. This can have the following values and meanings:

<code>W_BUTTON_TYPE_1</code>	draws the Series 3 style buttons
<code>W_BUTTON_TYPE_2</code>	draws the Series 3a style buttons

The state parameter has different meanings for the different types.

For `W_BUTTON_TYPE_1` buttons, 0 draws a raised button while 1 draws a depressed(flat) button.

For `W_BUTTON_TYPE_2` buttons, 0 draws a raised button, 1 draws a semi-depressed button while 2 draws a fully depressed(sunken) button.

The following picture shows three examples of `W_BUTTON_TYPE_2` buttons. From left to right, the examples show: a state 0 button (raised), a state 1 button (semi-depressed) and a state 2 button (fully depressed).



For the purpose of comparison, the following picture shows examples of `W_BUTTON_TYPE_1` buttons. From left to right, the examples show: a state 0 button (raised) and a state 1 button (fully depressed).



It is important to note that before calling `wDrawButton2` to draw `W_BUTTON_TYPE_2` buttons, the window must be enabled for drawing grey.

Bitmaps

Although directed at the drawable associated with the current graphics context, the graphics output functions in this section do not depend on any of the settable fields in the graphics context.

gCreateBit

Create a bitmap

```
INT gCreateBit(UINT flags, W_OPEN_BIT_SEG *pbitseg);
```

Create an uninitialised bitmap where the size of the bitmap and its method of storage is controlled by `flags` and the `W_OPEN_BIT_SEG` struct at `pbitseg`.

The `W_OPEN_BIT_SEG` struct is defined as:

```
typedef struct
{
    P_POINT size;
    TEXT seg_name[14];
} W_OPEN_BIT_SEG
```

where `pbitseg->size` specifies the dimensions of the bitmap in pixels and `pbitseg->seg_name` is written to by `gCreateBit` when requested by the bit in `flags` (as described next).

If `flags` is zero, the bitmap may be stored in the window server's data space or in a named memory segment depending on the size of the bitmap. If the bitmap requires less than 2.5K bytes, it is stored in the window server's data space.

The storage of the bitmap is controlled by setting the following bits in `flags`:

<code>WS_BIT_SEG</code>	to store the created bitmap in its own memory segment (regardless of its size)
<code>WS_BIT_SEG_ACCESS</code>	to create the bitmap in its own memory segment and to write the name of the segment as a zero terminated string to <code>pbitseg->seg_name</code> . The segment name can be used to access the bitmap directly using <code>p_sgcopyfr</code> and <code>p_sgcopyto</code> (described in the <i>Memory Allocation</i> chapter of the <i>PLIB Reference</i> manual) or otherwise.
<code>WS_BIT_SEG_ZERO_SIZE</code>	Used in conjunction with <code>WS_BIT_SEG_ACCESS</code> to create the bitmap's memory segment with zero size. You must subsequently increase the size at a later date. This is designed to be used in conjunction with graphics functions that are added using <code>wLoadDYL</code> .

If `flags` is either zero or `WS_BIT_SEG`, `pbitseg->seg_name` is not written to and, in this case, `pbitseg` may just be the address of a `P_POINT` struct. The prototype for `gCreateBit` actually declares `pbitseg` as a `VOID *` so that you can pass either a `WS_BIT_SEG *` or a `P_POINT *`.

You may use all the window server graphics functions to draw to a bitmap unless the `WS_BIT_SEG_ZERO_SIZE` flag is set, in which case the graphics functions will have no effect.

Returns the positive ID of the bitmap if successful.

If the function fails because there is insufficient memory, it calls `p_leave(E_GEN_NOMEMORY)` or returns `E_GEN_NOMEMORY`, depending on whether `wDisableLeaves` has been called.

gOpenBit

Load a bitmap

```
INT gOpenBit(TEXT *filename, UINT index, UINT flags, W_OPEN_BIT_SEG *pbitseg);
```

Load bitmap `index` from file `filename` where `index` is used to select a bitmap from a file that contains multiple bitmaps (a zero index selects the first bitmap). To load from a file containing a single bitmap, pass the index as zero.

The function writes to the `W_OPEN_BIT_SEG` struct at `pbitseg` where `W_OPEN_BIT_SEG` is defined as:

```
typedef struct
{
    P_POINT size;
    TEXT seg_name[14];
} W_OPEN_BIT_SEG
```

If the bitmap is successfully loaded, the dimensions of the bitmap in pixels is written to `pbitseg->size` and the ID of the bitmap is returned.

If `flags` is zero, the bitmap may be stored in the window server's data space or in a named memory segment depending on the size of the bitmap. If the bitmap requires less than 2.5K bytes, it is stored in the window server's data space.

The storage of the bitmap is controlled by setting the following bits in `flags`:

<code>WS_BIT_SEG</code>	to store the loaded bitmap in its own memory segment (regardless of its size)
<code>WS_BIT_SEG_ACCESS</code>	to store the loaded bitmap in its own memory segment and to write the name of the segment as a zero terminated string to <code>pbitseg->seg_name</code> . The segment name can be used to access the bitmap directly using <code>p_sgcopyfr</code> and <code>p_sgcopyto</code> (described in the <i>Memory Allocation</i> chapter of the <i>PLIB Reference</i> manual) or otherwise.
<code>WS_BIT_WRITE</code>	if set, you are given write access to the bitmap and a new bitmap is always created. If the flag is not set, the bitmap will only be created once and if any client calls <code>gopenBit</code> on the same bitmap, the loaded bitmap will be shared.

Provided that the `WS_BIT_WRITE` flag is set, you may use all the window server graphics functions to draw to a bitmap. If the `WS_BIT_WRITE` flag is not set, the graphics output functions will have no effect.

If `WS_BIT_SEG_ACCESS` is not set in `flags`, `pbitseg->seg_name` is not written to and, in this case, `pbitseg` may just be the address of a `P_POINT` struct. The prototype for `gopenBit` actually declares `pbitseg` as a `VOID *` so that you can pass either a `WS_BIT_SEG *` or a `P_POINT *`.

If an error (with error number `err`) occurs, the function calls `p_leave(err)` or returns `err`, depending on whether `wDisableLeaves` has been called.

Note that the parameter `filename` is ultimately passed to `p_open` by the window server process - not the client process. If `filename` is not a full file specification, the unspecified components are taken from the window server's default path which, in practice, is always the internal drive `M:\`. This is unlikely to meet the requirements of a finished product so `filename` should specify the drive and directory.

Typically, the bitmap file might be built into the application to create a *.app* file as described in the *Building An Application* chapter in the *General Programming* manual. In this situation you might have a code fragment that looks like:

```
GLREF_D TEXT *DatCommandPtr;

LOCAL_C INT OpenBit(TEXT *name)
{
    W_OPEN_BIT_SEG bseg;
    TEXT full[P_FNAME_SIZE];

    f_fparse(name, DatCommandPtr, &full[0], NULL);
    gOpenBit(&full[0], 0, 0, &bseg);
}
```

See also `gSetOpenAddress` for loading a bitmap file which is embedded in another file.

wFree

Free a bitmap

```
VOID wFree(UINT bitmap_id);
```

Free the bitmap with ID `bitmap_id`.

Any Graphics Contexts drawing to `bitmap_id` is automatically freed.

gSaveBit

Save a bitmap

```
INT gSaveBit(TEXT *filename, UINT bitmap_id);
```

Save the bitmap with ID `bitmap_id` to file `filename` or, if `bitmap_id` is zero, save the screen to file `filename`.

The parameter `bitmap_id` can also be the ID of a backed-up window to refer to the backup bitmap (but not in version 2).

Returns zero if successful.

If an error (with error number `err`) occurs, the function calls `p_leave(err)` or returns `err`, depending on whether `wDisableLeaves` has been called.

If `filename` is not a full file specification, the unspecified components are taken from the window server's default path which, in practice, is always the internal drive `M:\`.

In version 4 of the window server, the function will save a double bitmap when saving the screen or a backed-up window with a grey plane.

gSaveRect

Save part of a bitmap

```
INT gSaveRect(TEXT *filename, UINT bitmap_id, P_RECT *prect);
```

Save the rectangular block of pixels specified by `prect` from the bitmap with ID `bitmap_id` to file `filename` or, if `bitmap_id` is zero, save the rectangular block of pixels specified by `prect` from the screen to file `filename`.

The `P_RECT` struct is defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;
```

The parameter `bitmap_id` can also be the ID of a backed-up window to refer to the backup bitmap (but not in version 2).

Returns zero if successful.

If an error (with error number `err`) occurs, the function calls `p_leave(err)` or returns `err`, depending on whether `wDisableLeaves` has been called.

If `filename` is not a full file specification, the unspecified components are taken from the window server's default path which, in practice, is always the internal drive `M:\`.

Not available in version 2 of the window server.

In version 4 of the window server, the function will save a double bitmap when saving the screen or a backed-up window with a grey plane.

gCopyBit

Copy a bitmap to a window

```
VOID gCopyBit(P_POINT *pos, UINT bitmap_id, P_RECT *prect, UINT mode);
```

Copy the rectangular block of pixels specified by `prect` in the bitmap with ID `bitmap_id` to position `pos` in the destination.

The parameter `mode` should be one of:

<code>G_TRMODE_REPL</code>	where bits in the source pattern replace corresponding bits in the destination.
<code>G_TRMODE_SET</code>	where 1s in the source pattern set corresponding bits in the destination (0s in the source do not change corresponding bits in the destination).
<code>G_TRMODE_CLR</code>	where 1s in the source pattern clear corresponding bits in the destination (0s in the source pattern do not change corresponding bits in the destination).
<code>G_TRMODE_INV</code>	where 1s in the source pattern toggle corresponding bits in the destination (0s in the source pattern do not change corresponding bits in the destination).

The `P_POINT` and `P_RECT` structs are defined as:

```
typedef struct
{
    WORD x;
    WORD y;
} P_POINT;

typedef struct
{
    P_POINT tl; /* top left pixel (internal) */
    P_POINT br; /* bottom right pixel (external) */
} P_RECT;
```

This function should not be used to copy from a bitmap onto itself, since it does not handle overlapping source and destination areas - `gCopyRect` should be used instead.

The parameter `bitmap_id` can also be the ID of a backed-up window to refer to the backup bitmap (but not in version 2).

In version 4 of the window server, *when* `bitmap_id` refers to a backed-up window, the function has a special way of handling grey:

- In normal mode, the black plane only is copied from the source to the destination.
- In `G_GC_FLAG_GREY_PLANE` mode, the grey plane from the source is selected and copied to the grey plane of the destination. If the source only has one plane, then that is used as the source.
- In `G_GC_FLAG_BOTH_PLANES` mode, and with a two plane source, both planes are copied to their respective destination planes. If only one plane exists, then it will be copied to both planes of the destination.

Note that this does *not* apply when copying from any other form of bitmap - in all other cases the black and grey planes must be copied by two separate calls to `gCopyBit`.

gCopyRect**Copy a bitmap onto itself**

```
VOID gCopyRect(P_RECT *prect, P_POINT *pos, UINT mode);
```

Copy the rectangular block of pixels specified by prect to position pos.

The parameter mode is as for gCopyBit, described above.

This function should only be used when the current graphics context is assigned to a bitmap. It should not be used to copy parts of windows since it does not handle invalid areas or possible obscuring windows - wScrollRect should be used instead.

gPeekBit**Read a bitmap**

```
INT gPeekBit(UINT bitmap_id, P_POINT *pstart, UINT len, UBYTE *presult);
```

Copy a horizontal slice of len bits starting at (pstart->x, pstart->y) from bitmap_id to presult (which must be at least $((len+15)/8) \& (~1)$ bytes long).

To copy from the screen, set bitmap_id to zero.

If the section specified by start and len extend outside the bitmap then gPeekBit will call p_panic.

The parameter bitmap_id can also be the ID of a backed-up window to refer to the backup bitmap.

On version 2 of the window server, len must be less than $(8*MAX_WSERV_TO_CLIENT_BUFFER)$ bits long.

In version 4 of the window server, if the most significant bit of the bitmap_id is set, then the function will 'peek' from the grey plane. For example, the following code fragment re-directs the function to 'peek' from the grey plane:

```
bitmap_id |= 0x8000;  
gPeekBit(bitmap_id, ... );
```

gCheckBitmapID**Check if a bitmap is valid**

```
INT gCheckBitmapID(UINT bitmap_id);
```

Check if a bitmap is valid.

Returns zero if it is valid. Otherwise the function calls p_leave(E_FILE_NXIST) or returns E_FILE_NXIST, depending on whether wDisableLeaves has been called.

Not available in version 2 of the window server.

Multiple bitmaps

In version 4 of the window server, a set of bitmap functions is available that allows a bitmap file to be opened so that bitmaps within the file can be loaded or drawn direct from the file.

Loading multiple bitmaps from a file with this method is considerably quicker than repeated calls to gOpenBit as the file does not have to be opened and closed for every bitmap loaded.

Drawing bitmaps direct from a file is advantageous when drawing part of a large bitmap; only the parts actually drawn are loaded, saving on access time and storage.

When using this method to draw the whole bitmap, there is a trade-off between memory usage and the time taken to perform the draw. While loading the bitmap "bit by bit" as it is drawn makes it slower than loading the whole bitmap in one go and then drawing it, it needs less memory, as only that part of the bitmap to be drawn needs to be in memory at any one time.

gInitBit**Open a bitmap file**

```
INT gInitBit(TEXT *filename, INT *pcount);
```

Available in version 4 of the window server, this function opens the bitmap file `filename` ready for calls to `gGetBit` or `gDrawBit`.

If the open is successful, the function returns the handle of the open file and the number of bitmaps held in the file is written to `*pcount`.

If `filename` is not a full file specification, the unspecified components are taken from the window server's default path which, in practice, is always the internal drive M:\

If an error (with error number `err`) occurs, the function calls `p_leave(err)` or returns the (negative) error number, depending on whether `wDisableLeaves` has been called.

See also `gSetOpenAddress` for opening a bitmap file which is embedded within another file.

gGetBit**Load a bitmap from an open file**

```
INT gGetBit(UINT handle,UINT index,UINT flags, W_OPEN_BIT_SEG *pbitseg);
```

Available in version 4 of the window server, this function loads a bitmap from the bitmap file referenced by `handle` (as returned from a previous call to `gInitBit`).

`index` indicates the position of the bitmap within the file; zero indicates the first bitmap, one indicates the second and so on.

The behaviour of the function and the meaning of the parameters `flags` and `pbitseg` are the same as for `gOpenBit`.

Returns the ID of the bitmap if successful.

If an error (with error number `err`) occurs, the function calls `p_leave(err)` or returns the (negative) error number, depending on whether `wDisableLeaves` has been called.

gDrawBit**Draw a bitmap from an open file**

```
INT gDrawBit(P_POINT *pos,INT handle,P_RECT *prect,INT mode,INT index);
```

Available in version 4 of the window server, this effectively performs a `gGetBit`, `gCopyBit` and `wFree`. The `pos`, `prect` and `mode` parameters are the same as for `gCopyBit` in that they specify what is drawn. The `handle` and `index` parameters are the same as for `gGetBit` in that they reference the open bitmap file and indicate the position of the source bitmap within the file (relative to zero).

Only the relevant scan lines for the parts to be drawn are loaded; these are loaded one at a time. This means that no extra memory needs allocating.

Returns zero if successful.

If an error (with error number `err`) occurs, the function calls `p_leave(err)` or returns, depending on whether `wDisableLeaves` has been called.

gQueryBit**Query the size of a bitmap**

```
INT gQueryBit(INT handle,INT index, P_POINT *psize);
```

Available in version 4 of the window server, this function queries the size of a bitmap within a file opened by `gInitBit`. The open file is referenced by `handle` while `index` indicates the position of the bitmap within the file (relative to zero). The structure pointed to by `psize` is filled in with the size of the bitmap.

Returns zero if successful.

If an error (with error number `err`) occurs, the function calls `p_leave(err)` or returns, depending on whether `wDisableLeaves` has been called.

wFree **Close an open bitmap**

```
VOID wFree(UINT handle);
```

In version 4 of the window server, if passed the handle referencing an opened bitmap file (as returned from a call to `gInitBit`), the function closes the bitmap file but leaves loaded any bitmaps that came from that file.

gInitMultiSave **Initialise a multiple bitmap file**

```
INT gInitMultiSave(TEXT *fname, INT count);
```

Available in version 4 of the window server, this function creates a file `fname` prepared to receive up to `count` bitmaps.

The call returns a positive handle if the file was successfully opened and initialised. This handle should be used in subsequent calls to `gSaveMultiBit`, `gSaveMultiRect` and `gEndMultiSave`.

Any errors generated by the filing system when creating the file will be returned (or leave called). In this event, any file created will be deleted.

gSaveMultiBit **Save a bitmap to a multi bitmap file**

```
INT gSaveMultiBit(INT handle, INT bitmap);
```

Available in version 4 of the window server, this function attempts to save the bitmap with ID `bitmap` to the initialised multiple bitmap file referenced by `handle`. If `bitmap` is zero, the screen is saved to the file.

The call returns zero if successful.

Any error generated by the filing system when writing to the file will be returned (or leave called) in the same way as for `gSaveBit`. If an attempt is made to save more than the maximum permitted number of bitmaps (set by the parameter `count` in the call to `gInitMultiSave`) the function will return `E_GEN_TOOMANY`.

If the call fails (for whatever reason), the multiple bitmap file will be left in good condition and any bitmaps already saved will still be accessible. However, any further attempts to save to the file will not be allowed and it must be closed by a call to `gEndMultiSave`.

gSaveMultiRect **Save part of bitmap to multi bitmap file**

```
INT gSaveMultiRect(INT handle, INT bitmap, P_RECT *prect);
```

Available in version 4 of the window server, this function behaves in the same way as `gSaveMultiBit`. However, only that part of the bitmap specified by the rectangle defined by `prect` is saved.

The call can fail for exactly the same reasons as `gSaveMultiBit` returning the same values.

gEndMultiSave **End multiple bitmap save and close file**

```
INT gEndMultiSave(INT handle);
```

Available in version 4 of the window server, this function ends the saving of bitmaps to a multiple bitmap file referenced by `handle` and closes the file.

The call returns zero if successful.

Any error generated by the filing system when closing the file will be returned (or leave called).

gInquireChecksum **Inquire screen or bitmap checksum**

```
VOID gInquireChecksum(INT handle, UWORD *pchecksum);
```

Available in version 4 of the window server, this function calculates a checksum and places the value in a `UWORD` pointed to by `pchecksum`.

`handle` references the object of the checksum operation and can be the ID of a bitmap or a backed-up window. If `handle` is zero, the object of the checksum operation is the whole screen.

Adding graphics output functions

The window server can be extended by building additional low-level graphics output services, that write directly to the screen memory, into a dynamic link library (DYL). Such a DYL, when loaded using `wLoadDYL`, effectively becomes part of the window server.

Note that the services described in this section are not suitable for loading and accessing any other type of DYL.

Producing the window server extension DYL itself is an advanced topic and is not documented in this Software Development Kit. At the time of writing, no such DYL exists.

A DYL that extends the window server must first be loaded by calling `p_loadlib` (described in the *Object Oriented Programming* chapter in the *PLIB Reference* manual). It must then be loaded by the window server using `wLoadDYL`. The services supplied by the DYL can then be accessed using `wCallDYL` and `wCallDYLReply`.

wLoadDYL

Load a DYL

```
INT wLoadDYL(TEXT *name);
```

Load the loaded window server extension DYL name into the window server.

The DYL should already have already been loaded into memory by a call to `p_loadlib`.

The parameter `name` is the DYL segment name (which is the same as the DYL file name, but does not include the directory or the device).

Returns the ID of the DYL (to be used by `wCallDYL` and `wCallDYLReply`).

Example

```
GLDEF_C UINT LoadWservDYL(VOID)
{
    f_leave(p_loadlib("C:\\DYL\\TEST.DYL", &test_dyl_handle, TRUE));
    return(wLoadDYL("TEST.DYL"));
}
```

wCallDYL

Call a DYL function

```
VOID wCallDYL(UINT dyl_id, UINT class, UINT size_of_data, VOID *pdata);
```

Call a graphic function within the DYL `dyl_id` where `class` specifies which function within the DYL to call (as specified by the builder of the DYL).

`pdata` points to the parameters to be passed to the DYL.

`size_of_data` is the number of bytes of data at `pdata`.

wCallDYLReply

Call a DYL enquiry function

```
INT wCallDYLReply(UINT dyl_id, UINT class, UINT size_of_data, VOID *pdata, VOID *presult);
```

Call an enquiry function within the DYL where the function called should not do any graphics output.

`pdata` points to the parameters to be passed to the DYL.

`size_of_data` is the number of bytes of data at `pdata`.

Returns the value returned by the DYL function (negative return values cause `p_leave` to be called, positive values are returned normally).

The DYL function may also return a buffer of data, if it does `presult` should point to a buffer large enough to hold the returned data.

CHAPTER 5

EVENTS

Getting the next event

wGetEventWait

Wait for an event

```
VOID wGetEventWait(WS_EV *event);
```

Wait for a window server event and return with the event type and parameters written to the WS_EV struct at address event where WS_EV is defined as:

```
typedef struct
{
    WORD type;
    UWORD handle;
    UWORD time;
    WS_EVENT_UNION p;
} WS_EV;
```

where:

type	is the positive event type of the form WM_XXXX
handle	For events that are directed at a window (eg WM_REDRAW, WM_MOUSE), this is the handle that was specified to wCreateWindow. For events that are not directed at a window (eg WM_KEY, WM_FOREGROUND), it is set to the value specified to wConnect.
time	is set by key and mouse events. It gives the low order word of the system tick count (a tick is 1/32 of a second) when the event occurred. It may be used, for example, to detect double clicks.
p	is a union of event type-dependent parameters

The WS_EVENT_UNION union is defined as:

```
typedef union
{
    UWORD uword;
    UBYTE *dpoint;
    P_RECT rect;
    WMSG_KEY key;
    WMSG_MOUSE mouse;
    WMSG_RUBBER rubber;
    WMSG_CAPS caps;
} WS_EVENT_UNION;
```

These event-specific parts are described under the description of the associated event type, in the course of this chapter.

wGetEvent **Asynchronously request an event**

```
VOID wGetEvent(WS_EV *event);
```

Request an event from the window server and return without waiting for the request to complete. This is the asynchronous version of `wGetEventWait`. For more information on asynchronous events, see the chapter *Asynchronous Requests and Semaphores* in the *PLIB Reference* manual.

Calling `wGetEvent` initially sets `event->type` to `E_FILE_PENDING` to indicate that no event has yet arrived. When there is an event to deliver, the window server sets `event->type` to the event type and also sets the rest of event as appropriate for `event->type`. It then signals the caller's I/O semaphore.

Only one `wGetEvent` may be outstanding at a time. The window server panics the process if a second `wGetEvent` is called when one is already pending.

wGetEventSpecial **Asynchronously request selected events**

```
VOID wGetEventSpecial(WS_EV *event, UINT flags);
```

Introduced in version 4 of the window server, this is an enhanced version of the function `wGetEvent`, where the `flags` parameter is used to select which type of event(s) the window server is to deliver.

There can only be one call outstanding to either `wGetEvent` or `wGetEventSpecial` at any one time.

To change the type of event(s) selected in an outstanding `wGetEvent` or `wGetEventSpecial`, use the function `wGetEventUpdate`.

The following flags can be used to select the corresponding events. They can be ORED together:

<code>WE_KEY</code>	Selects key and task key events
<code>WE_REDRAW</code>	Selects <code>WM_REDRAW</code> events
<code>WE_STATUS</code>	Selects the <code>WM_FOREGROUND</code> , <code>WM_BACKGROUND</code> and <code>WM_ON</code> events
<code>WE_MOUSE</code>	Selects mouse and rubber band events
<code>WE_OTHERS</code>	Selects all events other than those listed above.
<code>WE_NORMAL</code>	Selects <i>all</i> of the above events
<code>WE_ESC</code>	This is only relevant when <code>WE_KEY</code> is <i>not</i> set. If the ESC key is pressed, the keyboard buffer is thrown away and a <code>WM_ESCAPE</code> event is delivered to the application.

Note that calling `wGetEventSpecial` with the `WE_NORMAL` flag set is equivalent to calling `wGetEvent`.

wGetEventUpdate **Change event types selected**

```
VOID wGetEventUpdate(UINT flags);
```

Introduced in version 4 of the window server, this function modifies the type of event(s) that the window server is to deliver where there is an outstanding `wGetEvent` or `wGetEventSpecial` call.

The previously selected event type(s) are discarded and replaced with a new set as specified in the `flags` parameter. The possible values for `flags` are the same as those described in `wGetEventSpecial`.

Note that if there is no outstanding call to either `wGetEvent` or `wGetEventSpecial` then calling `wGetEventUpdate` will have no effect.

Event types

This section describes event types that are common to more than one machine type.

The following descriptions assume that event is declared as:

```
WS_EV event;
```

WM_KEY

Key press event

Sent when a key is pressed where the key press is described by the WMSG_KEY struct event.p.key where WMSG_KEY is defined as:

```
typedef struct
{
    UWORD keycode; /* Code for the key pressed */
    UBYTE modifiers; /* State of mouse button, shift keys etc */
    UBYTE count; /* Used to accumulate auto-repeat counts */
} WMSG_KEY;
```

Count

For single key presses, event.p.key.count is 1.

If a key is held down, count will get to be greater than 1 when the client is unable to process keys at the rate at which the system generates repeated keys.

As described in the *Keyboard input* section of the first chapter, application programmers are best advised just to ignore the repeat count.

Modifiers

event.p.key.modifiers is a set of bit flags:

W_SHIFT_MODIFIER (0x02)	SHIFT key down
W_CTRL_MODIFIER (0x04)	CTRL key down
W_PSION_MODIFIER (0x08)	PSION key down
W_CAPS_MODIFIER (0x10)	caps lock on
W_NUM_LOCK_MODIFIER (0x20)	num lock on (MC only)

Keycode

When a "standard" key that represents a character from the SIBO character set¹ is pressed, event.p.key.keycode contains the character code of the corresponding character in the range 0x20 to (nominally) 0xff but excluding 0x7f. The actual upper limit on the code which can be directly produced from the keyboard is less than 0xff and depends on what language (eg French, German) the keyboard is produced for.

The SHIFT and CAPS LOCK keys modify the keycode following the normal conventions and as suggested by the labelling of the key.

In most cases on the HC and MC and for those keys that are used as accelerators on the S3 and S3a, the PSION shift key produces a keycode that is generated by adding 0x200 (W_SPECIAL_KEY) to the unshifted keycode with caps lock off. For example, pressing PSION+A with or without CTRL and SHIFT and regardless of the caps lock state produces a keycode of 0x261 (the code for lower case 'a' plus W_SPECIAL_KEY).

¹Similar to IBM's code page 850 - see the section *Text Fonts* in the first chapter.

On machines which have a CTRL key (that is, excluding the HC), you can generate any keycode from 0x00 to 0xff indirectly by holding down the CTRL key and typing the required code as a 3-digit decimal number (using leading zeros as necessary). In this case, a single key event is generated after the third decimal number is pressed. Also, following normal conventions, pressing CTRL-A to CTRL-Z produces a keycode from 0x01 to 0x1a.

Many of the keys do not represent printable characters from the SIBO character set (these keys are sometimes called "special keys"). Such keys generate a keycode which is either less than 0x20, 0x7f or greater than 0xff.

The following lists the key codes produced by the special keys:

W_KEY_TAB (0x9 or '\t')	Produced by TAB, with or without SHIFT or CTRL. On the HC and MC only, the PSION key adds W_SPECIAL_KEY. On the <i>Workabout</i> , the PSION key (but not SHIFT+PSION) converts the keypress to W_KEY_TASK.
W_KEY_DELETE_LEFT (0x08 or '\b')	Produced by DEL on the HC, DELETE with or without CTRL or PSION on the S3 and S3a, BACKSPACE with or without SHIFT or CTRL on the MC, and DEL without SHIFT on the <i>Workabout</i> . On the HC and MC, the PSION key adds W_SPECIAL_KEY.
W_KEY_DELETE_RIGHT (0x7f)	Produced by SHIFT+DEL on the HC and <i>Workabout</i> , SHIFT+DELETE on the S3 and S3a, DELETE with or without SHIFT or CTRL on the MC. On the MC only, the PSION key adds W_SPECIAL_KEY.
W_KEY_RETURN (0x0d or '\r')	Produced by ENTER, with or without SHIFT or CTRL. On the HC and MC only, the PSION key adds W_SPECIAL_KEY.
W_KEY_ESCAPE (0x1b)	Produced by ESC, without SHIFT or CTRL on the <i>Workabout</i> ; with or without SHIFT or CTRL on all other machines. On the HC and MC only, the PSION key adds W_SPECIAL_KEY.
W_KEY_UP (0x100)	Produced by UP ARROW, with or without SHIFT or CTRL. On the HC and MC only, the PSION key adds W_SPECIAL_KEY.
W_KEY_DOWN (0x101)	Produced by DOWN ARROW, with or without SHIFT or CTRL. On the HC and MC only, the PSION key adds W_SPECIAL_KEY.
W_KEY_RIGHT (0x102)	Produced by RIGHT ARROW on the HC, RIGHT ARROW with or without SHIFT or CTRL on the S3, S3a, <i>Workabout</i> and MC. On the HC and MC only, the PSION key adds W_SPECIAL_KEY.
W_KEY_LEFT (0x103)	Produced by LEFT ARROW on the HC, LEFT ARROW with or without SHIFT or CTRL on the S3, S3a, <i>Workabout</i> and MC. On the HC and MC only, the PSION key adds W_SPECIAL_KEY.
W_KEY_PAGE_UP (0x104)	Produced by PSION+UP ARROW on the S3, S3a and <i>Workabout</i> , PAGE UP with or without SHIFT or CTRL on the MC. On the MC only, the PSION key adds W_SPECIAL_KEY.
W_KEY_PAGE_DOWN (0x105)	Produced by PSION+DOWN ARROW on the S3, S3a and <i>Workabout</i> , PAGE DOWN with or without SHIFT or CTRL on the MC. On the MC only, the PSION key adds W_SPECIAL_KEY.
W_KEY_HOME (0x106)	Produced by PSION+LEFT ARROW on the S3, S3a and <i>Workabout</i> , HOME with or without SHIFT or CTRL on the MC. On the MC only, the PSION key adds W_SPECIAL_KEY.
W_KEY_END (0x107)	Produced by PSION+RIGHT ARROW on the S3, S3a and <i>Workabout</i> , END with or without SHIFT or CTRL on the MC. On the MC only, the PSION key adds W_SPECIAL_KEY.
W_KEY_TASK (0x108)	Produced by SHIFT+LEFT ARROW (TASK) on the HC, TASK with or without SHIFT or CTRL on the MC and PSION+TAB on the <i>Workabout</i> . On the MC only, the PSION key adds W_SPECIAL_KEY. Normally processed by the window server to switch the foreground task and not passed to clients. However, you can use <code>wCaptureKey</code> to capture the W_KEY_TASK key (as described under <code>wCaptureKey</code>).
W_KEY_VOICE (0x109)	Produced by RECORD with or without SHIFT or CTRL on the MC only. The PSION key adds W_SPECIAL_KEY.

W_KEY_CAPS_LOCK (0x10c)	Produced by CAPS LOCK on the S3 and MC, by PSION+DIAMOND on the S3a and by PSION+SPACE on the <i>Workabout</i>. This key press is processed by the operating system to set the caps lock state and passed to the window server on the S3, S3a and MC (but not on the HC). On the MC, the window server generates a WM_KEYBOARD_STATE_CHANGE event to the shell. The window server does not normally pass it on to the foreground client. You can use wCaptureKey to capture the W_KEY_CAPS_LOCK key.
W_KEY_BACKLIGHT (0x120)	Produced by the BACKLIGHT key with or without SHIFT on the HC and the <i>Workabout</i>. On the HC only, the PSION key adds W_SPECIAL_KEY. This key is normally processed by the operating system although you can disable it by calling p_setbacklight as described in the <i>General System Services</i> section of the <i>PLIB Reference</i> manual. The key is normally passed through to the foreground client (unless captured by a client using wCaptureKey).
W_KEY_INFO (0x121)	Produced by SHIFT+RIGHT ARROW (INFO) on the HC only.
W_KEY_MENU (0x122)	Produced by MENU with or without SHIFT on the HC and by MENU with or without SHIFT, CTRL or PSION on the S3, S3a and <i>Workabout</i>. On the HC only, the PSION key adds W_SPECIAL_KEY. If wsEnableTemp has been called, the window server processes PSION+MENU to present a temporary status window (in which case it does not pass the key press on to the client). By convention on the S3 and S3a, CTRL+MENU is processed by clients to present a permanent status window.
W_KEY_HELP (0x123)	Produced by HELP with or without SHIFT, CTRL or PSION on the S3 and S3a, and by ESC with either SHIFT or CTRL, but not PSION, on the <i>Workabout</i>. As suggested by the S3/S3a key labels, PSION+HELP on these machines should be interpreted as a DIAL key.
W_KEY_DIAMOND (0x124)	Produced by DIAMOND on the S3a only. Used by applications to switch from one mode to another.
W_KEY_APP1 to W_KEY_APP8 (0x131 to 0x138)	Produced by the 8 membrane keys on the S3 from left to right (also called application keys). These are normally handled by the window server in co-operation with the system task. For more information, see the section on <i>Clients and the Window Server</i> in the <i>Introduction</i> chapter of this manual.
W_KEY_MODE (0x130)	Produced on the S3 only when the application key that is associated with the foreground application is pressed. Applications normally cycle through their display modes in response to this key event. Produced by DIAMOND on the S3a when running in S3 compatibility mode.
W_KEY_LCD (0x2000) W_KEY_LCD_MINUS (0x2001)	Produced by the LCD BRIGHTER and LCD DIMMER keys on all machines except the <i>Workabout</i>. On the <i>Workabout</i> the single LCD BRIGHTER key produces W_KEY_LCD only, and SHIFT+LCD BRIGHTER is used to dim the LCD. These keys are processed by the operating system rather than the window server so you can't use wCaptureKey to disable them. These keys are normally passed through to the foreground client.
W_KEY_ON (0x2002)	Produced on an HC with version 3.5 of the window server, and on the S3, S3a and <i>Workabout</i> when the ON key is pressed. Also, when the machine switches on for <i>any</i> reason (such as the expiry of an absolute timer) the operating system manufactures an W_KEY_ON event to the window server².

²On the HC, the window server is only informed of the machine being switched on after p_setonevent(TRUE) has been called.

This event is normally processed by the window server to:

1. pass a WM_ON event to the foreground client (provided it has called `wInformOn`) or, in version 4 of the window server, pass a WM_ON event to a client whether it is in foreground or background (provided it has called `wInformOnAll(TRUE)`)
2. present an info message to inform the user of any low battery state³
3. to present the password alert if a password has been set.

When processed by the window server, the WM_KEY_ON event is not passed to the foreground client.

You can use `wCaptureKey` to capture the WM_KEY_ON key. This will disable all window server processing of this event.

WM_KEY_OFF (0x2003)

Produced by the OFF key on the HC, S3, S3a and Workabout only.

Normally processed by the window server to turn the machine off and not passed to clients. However, you can use `wCaptureKey` to capture the OFF key (as described under `wCaptureKey`). The capturer can turn the machine off by calling `p_off` - as described in the *General System Services* section of the *PLIB Reference* manual. (The same section also describes `p_setauto` which can be used to stop the machine from automatically switching off.) You don't get a WM_KEY_OFF event when the machine automatically switches off.

WM_REDRAW

Redraw event (WM

Sent when the client's event queue is empty and one or more windows has an update region.

The parameter `event.p.rect` describes a rectangular block of pixels from the update region (and which needs to be redrawn).

The only event type that has a lower priority than WM_REDRAW is WM_USER_MSG.

WM_BACKGROUND

Background event

Sent to a foreground client when it goes background.

Only `event.type` is set.

On all machines except the MC, you generally do not need to do anything when you receive a WM_BACKGROUND event. However, if you are doing anything that requires real-time input from the user (a game, for example) or you are doing an animated display, you should suspend the operation until you receive a WM_FOREGROUND event.

WM_FOREGROUND

Foreground event

Sent to a background client when it becomes foreground.

Only `event.type` is set.

WM_CANCELLED

Cancellation event

Sent in response to a call to `wCancelGetEvent` command - see the description of `wCancelGetEvent` in this chapter.

Only `event.type` is set.

³See also the description of the `WSERV_FLAG_LOW_BATTERY_WARNINGS` flag in `wSystem`.

WM_USER_MSG**User message event**

Sent in response to a call to `wUserMsg`. This event has the lowest priority of all and can be used to indicate that the window server has no more messages to send - see the description of `wUserMsg` in this chapter.

Only `event.type` is set.

WM_ON**Machine switched on event**

Available in version 3.5 of the window server; if the client has called `wInformOn`, it is sent this event when the machine is switched on and it is in foreground.

The event is designed to prompt the foreground client to update its display.

On the HC, the window server is only informed of the machine being switched on after `p_setonevent(TRUE)` has been called.

In version 4 of the window server, if the client has called `wInformOnAll(TRUE)`, it is sent this event when the machine is switched on, whether it is in foreground or background.

Only `event.type` is set.

WM_COMMAND**Command received from another client**

This is sent in response to a `wSendCommand` from another client to prompt the receiver of the event to call `wGetCommand` to get the command data. It is only available in version 3.5 of the window server.

Only `event.type` is set.

WM_TASK_UPDATE**Inform shell of process termination**

Sent to the shell if it is foreground and any process terminates (not just clients of the window server).

Available only on the S3, S3a and *Workabout*, and on an HC running version 3.5 upwards of the window server. Disabled by default on the HC - see the description of `wSystem`.

Only `event.type` is set.

WM_TASK_KEY**Inform application key handler**

Sent to the application key handler when:

- an application key is pressed and no process of that application exists
- a `PSION` shifted application key is pressed

Applies only to the S3, S3a and *Workabout*.

The index of the application key in the range 0 to 15 is written to `event.p.key.keycode`. The window server handles 16 application keys, where a second set of 8 keys are accessed by holding down the `CONTROL` key.

WM_DATE_CHANGED**Change of date event**

Introduced in version 4 of the window server. Sent whenever the date changes, either because the date has been reset or the clock has gone past midnight.

The message is sent to any Series 3a or *Workabout* application which is not in Series 3 compatibility mode and is in foreground at the time of the event

Non compatibility mode applications which are in background will receive the message the next time they come into foreground.

If the machine is off at the time of the event, the message is delivered when the machine is next turned on.

WM_ESCAPE**Escape-key event**

In version 4 of the window server, this message is delivered to an application when the ESC key is pressed in the following circumstances:

- The application must have asynchronously requested selected events by calling `wGetEventSpecial`.
- Among the events selected for delivery, `WE_EVENT_ESC` must be *included* but `WE_EVENT_KEY` must be *excluded*.

In this situation, the content of the keyboard buffer is discarded.

Large screen events

The event types in this section are only generated on large screen versions of the window server, such as on the MC200 and MC400 machines.

WM_DEICONISE**Deiconisation event**

Sent to a client to tell it to deiconise. The client will have previously declared itself iconised with a `wClientIconised` call.

Only `event.type` is set.

A client receives this message when another client calls `wClientPosition` to position it to the foreground.

It is also sent to the system application (`sys$shll.img`) when it is iconised and selected by the PSION-TASK key press.

WM_ATTACHED**Attachment event**

Sent to a client to tell it another client has attached itself on top of it. The `uword` field of the `WS_EVENT_UNION` structure is set to the process ID of attached client. After receiving this message the client will not be able to receive `WM_KEY` events until the attached client detaches, terminates or disconnects from the window server.

Only `event.type` is set.

A client will receive a `WM_ATTACHED` event when the notifier process (`sys$ntfy`) attaches itself to the foreground client when any process calls `p_notify` or `p_notifyerr`.

WM_DETACHED**Detachment event**

Sent to a client when a previously attached client detaches (either by a `wDetachClient` call, by terminating or by disconnecting from the window server).

Only `event.type` is set.

WM_KEYBOARD_STATE_CHANGE Keyboard state change event

This event is only ever sent to the system application (`sys$shll.img`). It is sent when either the numlock or capslock state changes. The new states of these can be read from `event.p.caps.modifiers`.

Mouse events

The event types in this section are only generated on machines with a pointing device, such as on the MC200 and MC400.

WM_MOUSE

Mouse event

Sent when ever there is a change of state on the mouse (digitiser).

The `WMSG_MOUSE` structure is defined as:

```
typedef struct
{
    UBYTE event; /* type of mouse event */
    UBYTE state; /* state of mouse button, shift keys etc */
    P_POINT pos; /* mouse position (relative to window) */
} WMSG_MOUSE;
```

`event.p.mouse.event` gives the type of the mouse event that occurred and is one of:

<code>WM_MOUSE_MOVE</code>	mouse movement event (this is filtered out by default)
<code>WM_MOUSE_PRESS</code>	mouse press event
<code>WM_MOUSE_RELEASE</code>	mouse release event

Mouse movement events (`WM_MOUSE_MOVE`) are filtered out unless explicitly enabled, on a per-window basis. See `wCreateWindow` and `wSetWindow` for details.

`event.p.mouse.state` gives the state of the mouse and the key modifiers when the event occurred, it may be tested using the bit masks:

<code>W_MOUSE_DOWN</code>	mouse button down
<code>W_MOUSE_OUTSIDE</code>	mouse event occurred outside window
<code>W_SHIFT_MODIFIER</code>	SHIFT key down
<code>W_CTRL_MODIFIER</code>	CTRL key down
<code>W_PSION_MODIFIER</code>	PSION key down
<code>W_CAPS_MODIFIER</code>	Caps lock on
<code>W_NUM_LOCK_MODIFIER</code>	Num lock on

The `W_MOUSE_OUTSIDE` bit is set when a mouse event occurs outside the visible portion of the given window. This can happen when either:

a window has grabbed the mouse, by specifying the `W_WIN_MOUSE_GRAB` bit, `WM_MOUSE_RELEASE` events are sent to the same window that received the `WM_MOUSE_PRESS` event, even if the mouse has subsequently moved outside the visible portion of the window.

or:

a window has captured the mouse by calling `wCaptureMouse`.

Testing the `W_MOUSE_OUTSIDE` bit is not equivalent to checking the mouse position against the extent of the window, because a client can never know if part of the window has been obscured.

WM_RUBBER_BAND_INIT

Start rubber band

This is a special version of the `WM_MOUSE` message, it also uses the `WMSG_MOUSE` structure. It is sent instead of a `WM_MOUSE` event when a mouse press occurs inside a window with the `W_WIN_RUBBER_BAND_CAPTURE` flag set.

See the *Rubber Band* section for details.

WM_RUBBER**Complete rubber band**

Sent on completion of a rubber band.

See the *Rubber Band* section for details.

WM_ACTIVE**Activation event**

Sent to a window that has previously set the `W_WIN_INACTIVE` bit (in a call to `wCreateWindow` or `wSetWindow`) whenever a `WM_MOUSE` event of type `WM_MOUSE_PRESS` is sent to the window or any of its descendants.

Only event . type is set.

`WM_ACTIVE` notifies a parent that the mouse has clicked somewhere in its window tree. The `WM_MOUSE` event is then sent straight to the window where the click occurred (unless that window has the `W_WIN_NO_MOUSE` bit set).

If a window and its descendant both have the `W_WIN_INACTIVE` bit set, they both receive a `WM_ACTIVE` event if there is a click in a descendant of the descendant window.

Event functions**wCancelGetEvent****Request a cancel event**

```
VOID wCancelGetEvent(VOID);
```

Instruct the window server to send the caller a `WM_CANCELLED` event.

After a call to `wCancelGetEvent`, the window server delivers the `WM_CANCELLED` event at the highest priority - any other events waiting in the window server client event queue are overtaken.

wUserMsg**Request a user event**

```
VOID wUserMsg(VOID);
```

Instruct the window server to send the client a `WM_USER_MSG` as soon as it has no other event to report.

A second call to this function before the first `WM_USER_MSG` is delivered will have no effect.

wSendCommand**Send a command to another client**

```
INT wSendCommand(HANDLE pid, VOID *pbuf, UINT len);
```

Send the `len` bytes of data at `pbuf` to the window server client with process ID `pid`.

If the call is successful, the function returns zero and client `pid` will receive a `WM_COMMAND` event to which it should respond by calling `wGetCommand` (as described below).

If no client with process ID `pid` exists, the function leaves or returns with the error number `E_FILE_NXIST`.

The function can be used to send up to 127 bytes. If `len` is 1 or 2, the function does not allocate any memory. If `len` is greater than 2, the function could leave or return with the error number `E_GEN_NOMEMORY`.

Only available in version 3.5 and upwards of the window server.

wGetCommand **Get a command from another client**

```
INT wGetCommand(VOID *pbuf);
```

Write to pbuf, the command data that was last sent to this process (with a call to wSendCommand).

This function should be called in response to the receipt of a WM_COMMAND event.

There should be at least 127 bytes of memory at pbuf.

If another command is sent to the client before it has read the old command, the old command is overwritten with the new data.

The function behaves as for wCheckPoint in that it flushes the client-side buffer and reports any uncleared error - either by calling p_leave or by returning the error number.

Only available in version 3.5 and upwards of the window server.

wInformOn **Enable the reception of WM_ON events**

```
VOID wInformOn(VOID);
```

Enable the reception of a WM_ON event when the machine is switched on.

The window server only sends a WM_ON event to the foreground client.

On the HC, the window server is only informed of the machine being switched on after p_setonevent(TRUE) has been called (normally by the shell).

Only available in version 3.5 and upwards of the window server.

wInformOnAll **Enable/disable the reception of WM_ON events**

```
VOID wInformOnAll(UINT state);
```

Available in version 4 of the window server, this function is similar to wInformOn. However, there are some subtle differences.

If state is TRUE, it enables the reception of WM_ON events; on the other hand, if state is FALSE, it disables the reception of WM_ON events.

When enabled by this call, WM_ON events are delivered whenever the machine is switched on, regardless of whether the calling client is in foreground or in background.

Disabling WM_ON events with this call disables the reception of WM_ON events regardless of whether they were originally enabled by a call to wInformOnAll or wInformOn.

Capturing keys

wCaptureKey **Capture a key**

```
INT wCaptureKey(UINT keycode, UINT modifiers, UINT modifier_mask);
```

Send the specified key press(es) to the calling client, whether it is foreground or not.

Every time a key is pressed the window server evaluates

```
(key_pressed_code==keycode) && ((key_pressed_modifiers&modifier_mask)==modifiers)
```

and if the result is TRUE then the keyboard event is sent to the client that specified the capture.

For example:

```
wCaptureKey(W_SPECIAL_KEY|'a',W_PSION_MODIFIER,W_PSION_MODIFIER);
```

captures PSION+A, PSION+SHIFT+A, PSION+SHIFT+CTRL+A, and PSION+CTRL+A. Whereas:

```
wCaptureKey(W_SPECIAL_KEY|'a',W_PSION_MODIFIER,W_PSION_MODIFIER|W_SHIFT_MODIFIER);
```

captures PSION+A and PSION+CTRL+A.

Note from the above that it is possible for two different but similar key/modifier combinations to capture the same key presses. This is significant if the two calls to `wCaptureKey` came from different clients. Where two key capture records select the same key press, the key event is delivered to the first client to call `wCaptureKey`.

The function is useful for implementing "hotkeys" which select a particular task. However, you should only capture relatively obscure key combinations which are not normally used by the tasks themselves (capturing the unmodified A key, for example, would be disastrous).

The function returns zero if successful. Errors include `E_GEN_NOMEMORY` and, if there is already a capture record with a matching keycode, `modifiers` and `modifier_mask` (even as a result of a `wCaptureKey` from another client), `E_FILE_EXIST`. The function either leaves or returns the error, depending on whether `wDisableLeaves` has been called.

The window server automatically cancels any calls a client has made to `wCaptureKey` when that client disconnects or terminates.

On an HC, an application can disable the window server's processing of the TASK key by capturing it with:

```
wCaptureKey(W_KEY_TASK, 0, 0);
```

Not available in version 2 of the window server.

wCancelCaptureKey

Cancel key capture

```
INT wCancelCaptureKey(UINT keycode, UINT modifiers, UINT modifier_mask);
```

Cancel a key capture set up by `wCaptureKey`, the keycode and masks must exactly match those used to initiate the capture.

The function returns zero if successful. If the keycode/modifier combination is not marked as captured then the function will leave or return `E_FILE_NXIST`.

Not available in version 2 of the window server.

Setting task switch keys

wSetTaskKey

Set a task switch key

```
INT wSetTaskKey(UINT keycode, UINT modifiers, UINT modifier_mask);
```

Set the specified keypress(es) to move the foreground client to the end of the task list and bring the client previously at position 1 (where the foreground process has position zero) to the foreground.

As for captured keys, every time a key is pressed the window server evaluates

```
(key_pressed_code==keycode) && ((key_pressed_modifiers&modifier_mask)==modifiers)
```

and if the result is `TRUE`, the tasks are cycled.

The effect of setting the task key does not cease when the calling client disconnects or terminates. The only way to stop the key press from being a task key is to call `wCancelTaskKey`.

Any number of task switch keys may be set. On the HC and MC, these operate in addition to the `W_KEY_TASK` key (unless the `W_KEY_TASK` key has been captured).

The function returns zero if successful. If there is already a set task key record with a matching keycode, `modifiers` and `modifier_mask`, the function leaves or returns `E_FILE_EXIST`. It can also fail with `E_GEN_NOMEMORY`. If there is already a *capture* record with a matching keycode, `modifiers` and `modifier_mask`, the capture key record is cancelled and replaced by the task key record.

The shell on the S3 and S3a calls `wSetTaskKey` to assign `SHIFT+SYSTEM` as a task key.

Only available in version 3.5 upwards of the window server.

wCancelTaskKey**Cancel a task switch key**

```
INT wCancelTaskKey(UINT keycode, UINT modifiers, UINT modifier_mask);
```

Cancel a task key setting, set up with wSetTaskKey.

The function returns zero if successful. If the keycode/modifier combination is not marked as a task key then the function will leave or return E_FILE_NXIST.

wSetBackTaskKey**Set a back task switch key**

```
INT wSetBackTaskKey(UINT keycode, UINT modifiers, UINT modifier_mask);
```

Set the specified keypress(es) to bring the client furthest from the front to the foreground.

Except that it cycles tasks in the opposite direction, wSetBackTaskKey is identical to wSetTaskKey.

The shell on the S3 and S3a calls wSetBackTaskKey to assign SHIFT+PSION+SYSTEM to the "back-task" key which brings the task furthest from the front to the foreground.

wCancelBackTaskKey**Cancel a back task switch key**

```
INT wCancelBackTaskKey(UINT keycode, UINT modifiers, UINT modifier_mask);
```

Cancel a back task key setting, set up with wSetBackTaskKey.

The function returns zero if successful. If the keycode/modifier combination is not marked as a back task key then the function will leave or return E_FILE_NXIST.

Capturing the mouse**wCaptureMouse****Capture the mouse**

```
VOID wCaptureMouse(UINT wid);
```

Capture the mouse within window wid and all its descendants. This function does not capture with respect to other clients' windows.

It is used, for example, by the dialog box that allows the user to click on it or any of its constituent windows, but ignores clicks to other windows in the application (ie the menu bar and the application's client window).

If mouse capture is already active in another window then the previous capture will be cancelled before the new capture is activated.

If a window with capture is destroyed the mouse is automatically released.

wReleaseMouse**Release the mouse**

```
VOID wReleaseMouse(VOID);
```

Cancel the mouse capture, does nothing if there was no capture active.

The rubber band

The rubber band is only implemented on machines with a pointing device, such as on the MC200 and MC400.

wRubberBand

Rubber banding

```
VOID wRubberBand(UINT msg_window, UINT band_window, W_RUBBER_BAND *prubber);
```

Start the rubber band and return immediately where the result is returned later as a WM_RUBBER event (which might just indicate that the parameters are illegal).

msg_window is the ID of the window to which the WM_RUBBER event will be sent.

band_window is the ID of the window in which the rubber band will be drawn, it is usually set to zero (the whole screen).

The rubber band is displayed as specified by the W_RUBBER_BAND struct at address prubber where W_RUBBER_BAND is defined as:

```
typedef struct
{
    P_EXTENT start;      /* initial size and position */
    P_EXTENT outer;      /* outer bounding rectangle */
    P_EXTENT inner;      /* inner bounding rectangle */
    UWORD flags;
    UWORD minx;          /* max and min size limits */
    UWORD miny;
    UWORD maxx;
    UWORD maxy;
    P_POINT grid_snap;   /* x and y grid snap values */
} W_RUBBER_BAND;
```

If flags is set to zero then the following default values will be used:

resizing	Disabled.
minx	Not applicable when resizing disabled.
miny	
maxx	
maxy	
start	The extent of msg_window.
outer	No outer bounds.
inner	The visible extent of band_window.
grid_snap	(1,1) in the x and y directions.
complete on release	Disabled.

Each of these defaults may be overridden by setting the following bits in flags:

W_BAND_RESIZE	enables resizing of the rubber band, if this is selected the rubber band will appear on screen with its resize triangles, if resizing is disabled the rubber band appears as a rectangle. When resizing is enabled then minx, miny, maxx and maxy (which determine the maximum and minimum sizes of the rubber band) must be set.
W_BAND_START	sets the start position and size to start. If this conflicts with inner, outer or the maximum or minimum size limits then it will be modified appropriately.
W_BAND_INNER	sets the inner rectangle to inner. The movement of the rubber band is restricted such that part of the rubber band stays within this rectangle.
W_BAND_OUTER	sets the outer rectangle to outer. The movement of the rubber band is restricted such that no part of the rubber band extends outside this rectangle.

<code>WM_BAND_GRID_SNAP</code>	sets the grid snap values to <code>grid_snap</code> . The rubber band will move/resize in steps of <code>grid_snap</code> .
<code>WM_BAND_GRID_SNAP_SIZE</code>	as for <code>WM_BAND_GRID_SNAP</code> except that only the size of the rubber band (and not its position) is grid snapped.
<code>WM_BAND_COMPLETE_ON_UP</code>	causes the rubber band to complete on the first mouse up event.

The values in `start`, `outer` and `inner` are all relative to `band_window`.

If there is no legal position for the rubber band then the rubber band completes immediately with a `WM_RUBBER` message with the `state` set to `WM_BAND_ERROR`. This could happen (say) if `minx` is greater than `maxx` or if `inner` does not intersect with `outer`.

WM_RUBBER events

The window server sends a `WM_RUBBER` message when the rubber banding completes. The format of the `WM_RUBBER` message is:

```
typedef struct
{
    UWORD state;      /* completion state */
    P_EXTENT extent; /* the selected extent */
} WMSG_RUBBER;
```

`sstate` is set to one of the following:

<code>WM_BAND_NOMOVE</code>	the band position was selected without any moving or resizing, <code>extent</code> is the same as set in <code>wRubberBand</code> .
<code>WM_BAND_MOVE</code>	the rubber band moved but did not change size. <code>extent</code> is set to the new position and the old height and width.
<code>WM_BAND_RESIZE</code>	the rubber band has been resized (and perhaps also moved). <code>extent</code> contains the new position and size.
<code>WM_BAND_CANCEL</code>	the rubber band was cancelled. <code>extent</code> is undefined.
<code>WM_BAND_ERROR</code>	the rubber band was not displayed because of illegal parameters in <code>wRubberBand</code> .

Capturing mouse and keyboard events

The flag `W_WIN_RUBBER_BAND_CAPTURE` used in the `wSetWindow` and `wCreateWindow` commands can be used to capture all mouse and keyboard events to the rubber band from the moment the mouse was pressed in the specified window. The first click in the window will be sent to the window as a `WM_RUBBER_BAND_INIT` event and held in a buffer as a `WM_MOUSE` event (of type `WM_MOUSE_PRESS`). Subsequent mouse and keyboard events will also be buffered. When the rubber band becomes active it will receive all the buffered events. The capture is cancelled on completion of the rubber band.

When a client receives a `WM_RUBBER_BAND_INIT` event it **MUST** call `wRubberBand` immediately. This is because all clients will have all their mouse events and keys blocked. If the client decides that it does not want to launch a rubber band then it should set the `WM_BAND_KILL_CAPTURE` flag in `flags` and call `wRubberBand`, this will cancel the mouse capture without actually launching a rubber band (all parameters in the `WM_RUBBER_BAND` structure are ignored except that `msg_window` must be a valid window ID).

CHAPTER 6

WINDOW SERVER REFERENCE UPDATE

This document is a beta version and may be subject to change.

This chapter describes the changes and additions that have been made to the Window server as a result of the introduction of the Siena and Series 3c machines into the SIBO range.

Note: Siena was codenamed Vine by Psion during development, hence the naming of some of the constants given below.

Screen sizes

The true screen and pixel dimensions of the various LCD screens on all SIBO machines are as follows:

Machine type	Screen (pixels)	Pixel pitch (mm)	Pixel size (mm)	Screen size (cm)	Screen size (in)
HC	160x80	0.34x0.43	0.31x0.40	5.44x3.44	2.14x1.35
S3	240x80	0.385x0.43	0.355x0.40	9.24x3.44	3.64x1.35
Workabout	240x100	0.26x0.30	0.23x0.27	6.24x3.00	2.45x1.18
Siena	240x160	0.25x0.25	0.23x0.23	6.00x4.00	2.36x1.57
S3a	480x160	0.259x0.259	0.20x0.20	12.477x4.156	4.915x1.637
S3c	480x160	0.26x0.26	0.20x0.20	12.478x4.158	4.915x1.637
MC200	640x200	0.33x0.33	0.30x0.30	21.12x6.60	8.31x2.60
MC400	640x400	0.33x0.33	0.30x0.30	21.12x13.20	8.31x5.20

In the above table, the horizontal measure is shown before the vertical measure. The pixel pitch measures the horizontal and vertical distance between the same points on adjacent pixels. The difference between the pixel size and the pixel pitch gives the gap between pixels.

The Series 3c and the Siena screens support the use of grey in exactly the same way as for the Series 3a.

Keyboard

Except where stated below, the keycodes produced on the Series 3c and the Siena are identical to those produced on the Series 3a.

The following table lists the additional and/or modified key codes produced by the special keys on the Siena and the Series 3c, as listed in *wskeys.h*:

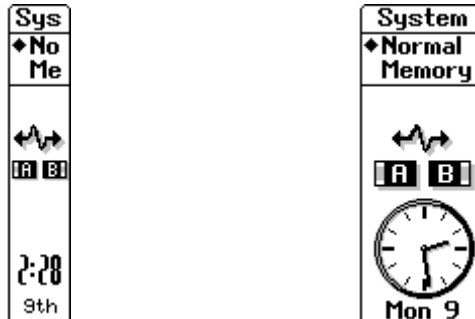
<code>W_KEY_TAB</code> (0x9 or '\t')	Produced by TAB, with or without SHIFT or CTRL. On the HC and MC only, the PSION key adds <code>W_SPECIAL_KEY</code> . On the <i>Workabout</i> , the PSION key (but not SHIFT+PSION) converts the keypress to <code>W_KEY_TASK</code> . On the Series 3c, the PSION key (including SHIFT+PSION) converts the keypress to <code>W_KEY_IR_LINK</code> (0x142).
--------------------------------------	---

W_FUNC_MODIFIER (0x80)	On the Siena only, an FN modifier key is provided that produces this additional modifier code. Note that the same value is used for Mouse Down (W_MOUSE_DOWN) on the Psion MC range of machines.
W_KEY_APP1 to W_KEY_APP9 (0x131 to 0x139)	On the Series 3 and Series 3a, key codes W_KEY_APP1 to W_KEY_APP8 are produced by the eight membrane keys (also called application keys) in order, from left to right. The Series 3c has a ninth membrane key, on the extreme right, that produces W_KEY_APP9.
W_KEY_IR_BRING (0x140)	On the Siena only, this key code is produced by the IR receive key.
W_KEY_IR_SEND (0x141)	On the Siena only, this key code is produced by the IR send key.
W_KEY_IR_LINK (0x142)	On the S3c only, this key code is produced by the PSION + TAB key combination.
W_KEY_CALC_CLEAR (0x01b)	On the Siena only, this key code is produced by the ON/CE key.
W_KEY_CALC_MEM_CLEAR (0x150)	On the Siena only, this key code is produced by FN + the '3' key on the numeric keypad.
W_KEY_CALC_MEM_RECALL (0x151)	On the Siena only, this key code is produced by FN + the '2' key on the numeric keypad.
W_KEY_CALC_MEM_MINUS (0x152)	On the Siena only, this key code is produced by FN + the '-' key on the numeric keypad.
W_KEY_CALC_MEM_PLUS (0x153)	On the Siena only, this key code is produced by FN + the '+' key on the numeric keypad.
W_KEY_CALC_CHNG_SIGN (0x154)	On the Siena only, this key code is produced by FN + the '.' key on the numeric keypad.
W_KEY_CALC_PERCENT (0x155)	On the Siena only, this key code is produced by FN + the '=' key on the numeric keypad.
W_KEY_CALC_DECIMAL (0x156)	On the Siena only, this key code is produced by the '.' key on the numeric keypad.
W_KEY_CALC_MEM_INPUT (0x158)	On the Siena only, this key code is produced by FN + the '1' key on the numeric keypad.
W_RUSSIAN_MODIFIER (0x1000)	On the Russian variant Series 3a and Series 3c only, (for Psion internal use only).

Status windows

Series 3c

Status windows on the Series 3c are functionally identical to those on the Series 3a. Some minor cosmetic changes have been made, including a new analog clock design for the wide status window.



Siena

Only a narrow status window is available on the Siena. A smaller font is used and the diamond list has been reorganised, compared with that of the Series 3a, to maximise the space for displaying text. Despite these changes, only four characters of the application name and up to five characters of each diamond list item can be displayed.



Diamond list text items that contain more than five characters are automatically truncated. Since such truncation may occur at an unsuitable point in the text, you should consider supplying truncated versions of the diamond text for use in the status window, such as the four-character abbreviations for 'Normal' and 'Outline' shown in the above illustration.

The following constants are defined in *wlib.h*:

WS_WIDTH_V4C	51	Normal status window width on Series 3c, in pixels; this is not a new constant - it also exists on the Series 3a
WS_WIDTH_SMALL_V4C	32	Narrow status window width on Series 3c, in pixels; this is not a new constant - it also exists on the Series 3a
WS_WIDTH_VINE	36	Narrow status window width on Siena, in pixels

Clocks

The clock creation functions, for example, `wsCreateClock2`, on the Siena and Series 3c support additional clock styles. The new styles and their appearances and dimensions are given below. The associated symbolic constants are defined in *wlib.h*.

Siena

The Siena supports one additional clock, of type `WS_CLOCK_VINE`:

`WS_CLOCK_VINE` 0x06



<code>WS_BITMAP_VINE_CLOCK_SIZE_X</code>	70	Width of Siena analogue clock, in pixels
<code>WS_BITMAP_VINE_CLOCK_SIZE_Y</code>	69	Height of Siena analogue clock, in pixels

Series 3c

The Series 3c supports two additional clock types, `WS_CLOCK_MEDIUM3` and `WS_CLOCK_XL2_ANALOG`:

`WS_CLOCK_MEDIUM3` 0x06



<code>WS_BITMAP_MEDIUM3_CLOCK_SIZE_X</code>	58	Width of Series 3c medium analogue clock, in pixels
<code>WS_BITMAP_MEDIUM3_CLOCK_SIZE_Y</code>	51	Height of Series 3c medium analogue clock, in pixels

`WS_CLOCK_XL2_ANALOG` 0x07



<code>WS_BITMAP_XL2_CLOCK_SIZE_X</code>	111	Width of Series 3c extra large analogue clock, in pixels
<code>WS_BITMAP_XL2_CLOCK_SIZE_Y</code>	110	Height of Series 3c extra large analogue clock, in pixels

Window Server versions

The following values are contained in the `version_id` member of the `W_SERVER_INFO` struct. The constants and the struct are defined in *wlib.h*:

<code>WS_TYPE_S3C</code>	0x60	Series 3c window server
<code>WS_TYPE_VINE</code>	0x70	Siena window server

INDEX

- `$WS_FL`
 - environment variable, 1-2, 1-55, 1-56, 2-25
- `$WS_FNTS`
 - environment variable, 1-2, 1-38, 2-10
- `$WS_IF`
 - environment variable, 2-10, 3-16
- `$WS_SD`
 - environment variable, 1-32
- `$WS_SF`
 - environment variable, 1-38
- `$WS_SF2`
 - environment variable, 1-39
- `$WS_SF4`
 - environment variable, 1-39
- `.pcx` files
 - from screen capture, 1-33
 - to bitmap PIC files, 1-31
- `.ph` files
 - multi bitmap header file, 1-31
- `.pic` files
 - checksum, 1-37
 - from a PCX file, 1-31
- `.plk` files
 - multi bitmap files, 1-31
- activation
 - event, 5-10
- add files
 - embedded bitmap files, 1-37
- alert flag
 - `WS_ALERT_B`, 1-3
- alerts
 - asynchronous, 2-18
 - message display, 2-12
 - synchronous, 2-14
 - updating, 2-19
 - `wsAlertW`, 1-3
- animation
 - bitmap sequences, 1-21, 3-14
 - sprites, 1-3, 1-23, 3-15
- application keys
 - handler, 1-15
 - `W_KEY_APP1`, 6-2
 - `W_KEY_APP1` to `W_KEY_APP9`, 6-2
 - `W_KEY_APP2`, 6-2
 - `W_KEY_APP3`, 6-2
 - `W_KEY_APP4`, 6-2
 - `W_KEY_APP5`, 6-2
 - `W_KEY_APP6`, 6-2
 - `W_KEY_APP7`, 6-2
 - `W_KEY_APP8`, 6-2
 - `W_KEY_APP9`, 6-2
 - `W_KEY_APPn`, 5-5
 - `wAppKeyHandler`, 1-15

- area fill pattern
 - `gFillPattern`, 4-13
- area filling
 - modes, 4-12, 4-13
 - WLIB functions, 4-12
- arrow keys
 - rubber band mode, 1-54
- arrows
 - drawing, 4-10
- ascent
 - fonts, 4-14
- asynchronous
 - events, 5-2
- attached clients
 - WLIB functions, 2-25
- attachment
 - event, 5-8
- backed up windows
 - bitmaps, 1-19
- background
 - client, 1-14
 - client switch task order, 2-8
 - event, 5-6
- background modes
 - window, 1-3
- backlight
 - key code, 5-5
- baseline
 - fonts, 4-14
- bitmaps
 - capture screen to, 1-32
 - converting from pcx with `wspcx.exe`, 1-31
 - copying, 4-29, 4-30
 - creation of, 1-30, 4-26
 - drawables, 1-18
 - drawing from, 1-30, 4-31
 - drawing to, 1-31
 - embedded files, 1-37
 - files creating, 1-31
 - files creating with grey, 1-31
 - find in file `gSetOpenAddress`, 1-30
 - freeing, 4-28, 4-32
 - freeing sequences, 3-15
 - from PCX files, 1-31
 - `G_TRMODE_CLR`, 1-30
 - `G_TRMODE_INV`, 1-30
 - `G_TRMODE_REPL`, 1-30
 - `G_TRMODE_SET`, 1-30
 - `gCopyBit`, 1-30
 - `gDrawBit`, 1-30
 - `gFillPattern`, 1-30
 - grey, 1-22
 - `gSetOpenAddress`, 1-30
 - header file for multi bitmap files, 1-31
 - horizontal slice, 4-30
 - joining multiple from PLK file, 1-31
 - loading, 4-27, 4-31
 - multiple, 4-30
 - open multiple bitmap file `gInitBit`, 1-30
 - opening, 4-31
 - overview, 1-30
 - pixel coordinates, 1-47
 - reading, 4-30
 - redraws avoiding, 1-22

- redraws avoiding example code, 1-22
- ROM-based grey, 1-30
- saving, 4-28
- screen capture, 1-31
- sequences, 1-21, 3-14
- storage of, 1-30
- structure, 1-36
- validating, 4-30
- wFree, 1-30
- WLIB functions, 1-2, 4-26
- wSetWinBitmap, 1-30
- bitmaps copying
 - gCopyBit, 4-29
 - gCopyRect, 4-30
- bitmaps create
 - gCreateBit, 4-26
- bitmaps drawing from
 - gDrawBit, 4-31
- bitmaps freeing
 - wFree, 4-32
- bitmaps function
 - wChangeWinBitmap, 3-15
 - wFree, 3-15
 - wSetWinBitmap, 3-14
- bitmaps loading
 - gGetBit, 4-31
 - gOpenBit, 4-27
- bitmaps multi save end
 - gEndMultiSave, 4-32
- bitmaps multiple initialise
 - gInitMultiSave, 4-32
- bitmaps multiple saving
 - gSaveMultiBit, 4-32
- bitmaps open
 - embedded, 4-28
 - structure, 4-26, 4-27
- bitmaps opening
 - gInitBit, 4-31
- bitmaps partial save
 - gSaveMultiRect, 4-32
- bitmaps position
 - gSetOpenAddress, 4-15
- bitmaps reading
 - gPeekBit, 4-30
- bitmaps saving
 - gSaveBit, 4-28
 - gSaveRect, 4-28
 - multiple, 4-32
- bitmaps sequences
 - wChangeWinBitmap, 1-21
 - wFree, 1-21
 - wSetWinBitmap, 1-21
- bitmaps size get
 - gQueryBit, 4-31
- bitmaps validating
 - gCheckBitmapID, 4-30
- black
 - plane, 1-3, 1-19, 3-8, 4-1, 4-13
- blind operations
 - window server, 1-11
- body cell
 - fonts, 4-14
- border
 - WLIB function, 4-8
 - WLIB functions, 4-10
- border attribute
 - W_BORD_CORNER_1, 4-11
 - W_BORD_CORNER_2, 4-11
 - W_BORD_CORNER_4, 4-11
 - W_BORD_CUSHION, 4-11
 - W_BORD_OPEN, 4-11
 - W_BORD_SHADOW_D, 4-11
 - W_BORD_SHADOW_ON, 4-11
 - W_BORD_SHADOW_S, 4-11
- border drawing
 - gBorder, 4-10
 - gBorder2, 4-12
 - gBorder2Rect, 4-10
 - gBorderRect, 4-8
 - types, 4-10
- box
 - WLIB function, 4-8
- buffered commands
 - window server, 1-11, 1-13, 2-5
- busy
 - message cancel, 2-12
 - message display, 2-11
- busy message
 - window server option, 1-6
- button drawing
 - example code, 4-24
- button text
 - wDrawButton, 4-24
 - wDrawButton2, 4-25
- cancelled
 - event, 5-6, 5-10
- capturing
 - keyboard events, 5-15
 - keys, 1-51, 5-11
 - mouse, 1-53, 5-13
 - mouse events, 5-15
 - screen to a bitmap file, 1-32
- checksum
 - font files, 1-46
 - pic files, 1-37
 - WLIB function, 4-32
- checksum get
 - gInquireChecksum, 4-32
- child
 - window, 1-18
- clear area
 - gClrRect, 4-12
- CLIB
 - library, 1-8
 - startup module, 1-8, 2-1
 - startup module MC, 1-10
- client
 - attach caller, 2-26
 - attach caller to foreground, 2-26
 - attached, 2-25, 5-8
 - background, 1-14
 - command event, 5-7
 - command get, 5-10
 - command send, 5-10
 - de-iconised event, 5-8
 - detach caller, 2-26
 - detached, 2-26, 5-8
 - foreground, 1-14

- going deaf, 1-26
- iconised, 1-16
- iconised mark as, 2-9
- information get, 2-8
- list get, 2-10
- management, 1-17
- pause disable, 2-10
- pausing, 1-52
- pausing enable, 2-9
- priority, 1-16, 2-7
- system-modal, 1-17, 2-9
- task order, 2-8
- wCancelSystemModal, 2-9
- client commands
 - between window server clients, 1-6
- client window
 - background drawing, 1-18
- clients
 - window server, 1-13
- clients terminating
 - window server, 1-6
- client-side buffer
 - flushing, 1-13
- clipping
 - redrawing, 3-9
 - windows, 1-25
 - windows child, 1-18
- clock
 - creating, 3-17, 3-19
 - example code, 3-18, 3-20, 3-21
 - freeing, 3-22
 - offset, 3-22
 - Series 3c and Siena update, 6-4
 - Series 3c update, 6-4
 - Siena update, 6-4
 - structure, 3-19
 - WLIB functions, 3-16
- clock attribute
 - WS_CLOCK_AM_PM, 3-17, 3-20
 - WS_CLOCK_BOX, 3-20
 - WS_CLOCK_CENTERED, 3-17, 3-20
 - WS_CLOCK_FORCE_ANALOG, 3-17, 3-20
 - WS_CLOCK_FORCE_DIGITAL, 3-17, 3-20
 - WS_CLOCK_FORMATTED, 3-19
 - WS_CLOCK_GREY, 3-20
 - WS_CLOCK_LARGE_ANALOG, 3-17, 3-19
 - WS_CLOCK_MEDIUM, 3-17, 3-19
 - WS_CLOCK_MEDIUM2, 3-19
 - WS_CLOCK_SMALL_DIGITAL, 3-17, 3-19
 - WS_CLOCK_WITH_DATE, 3-17, 3-20
 - WS_CLOCK_WITH_SECONDS, 3-17, 3-20
 - WS_CLOCK_XL_ANALOG, 3-19
- clock drawing
 - automatic, 1-6
- clock function
 - wFree, 3-22
 - wsCreateClock, 3-17
 - wsCreateClock2, 3-19
 - wsSetClock, 3-22
- code
 - key, 1-51
 - keys, 5-3
- command data get
 - wGetCommand, 5-10
- compatibility mode
 - S3 display mode, 2-6
 - Series 3, 1-12
 - status windows, 2-21
 - W_CTBY_S3, 2-6
 - W_CTBY_S3_SCR, 2-6
 - wInquireCompatibility, 2-7
- compute mode
 - ending wEndCompute, 2-8
 - priority setting wStartCompute, 2-8
- configure
 - font group, 4-18
 - window server, 2-24
- connect to
 - window server, 2-1, 2-3
- CONNECT_INFO
 - structure, 2-3
- console device
 - channel to, 1-8
- coordinates
 - pixel, 1-47
- copying bitmaps
 - WLIB function, 4-29, 4-30
- corner type
 - border attribute, 4-8
 - border attributes, 4-11
- count
 - key repeat, 1-51
 - keys repeat, 5-3
- cursor
 - flashing, 1-29
 - grey, 1-29
 - text, 3-12
 - text window, 1-29
 - wTextCursor, 1-29
- cursor attribute
 - W_CURSOR_GREY, 3-12
- cursor function
 - wDrawTextCursor, 3-13
 - wEraseTextCursor, 3-13
 - wTextCursor, 3-12
- cursor remove
 - wEraseTextCursor, 1-29
- date changed
 - event, 5-7
- DatStatusNamePtr
 - magic static, 2-13
- deaf clients
 - multi-tasking, 1-26
 - window server flag, 1-56
- de-iconise
 - event, 5-8
- descent
 - fonts, 4-14
- destroying
 - windows, 1-27
- detachment
 - event, 5-8

- diamond key
 - key code, 5-5
 - key press, 1-15
 - mode display, 2-20
- disable
 - status window, 2-22
 - status window temporary, 2-22
- disconnect from
 - window server, 2-5
- display
 - brief message, 2-11
- display process
 - SYS\$CONS on MC, 1-7
- double pixel mode
 - window, 1-12
- doubled sized pixels
 - mode, 4-1
 - S3 display mode, 2-6
- drawable
 - graphics context, 4-1
 - windows and bitmaps, 1-18, 1-50
- drawing
 - arrows, 4-10
 - background client, 1-18
 - lines, 4-7
 - polygon, 4-7
 - shadowed text, 4-23
 - shadows, 4-10
 - text, 1-44
 - to bitmaps, 1-31
 - windows, 1-25
- drawing mode
 - G_GC_FLAG_BOTH_PLANES, 4-3
 - G_GC_FLAG_DOUBLE, 4-3
 - G_GC_FLAG_GREY_PLANE, 4-3
- drawing region
 - window, 1-19
- DYL graphics
 - WLIB function, 4-33
- E_CONFIG
 - structure, 3-16
- E_PRIORITY_BACK
 - window server flag, 2-7
- E_PRIORITY_FORE
 - window server flag, 2-7
- embedded
 - font files, 1-47
- embedded bitmap files
 - add file lists, 1-37
- enable
 - status window temporary, 2-22
- end rubber band
 - event, 5-10
- environment variable
 - \$WS_FL, 1-2, 1-55, 1-56, 2-25
 - \$WS_FNTS, 1-2, 1-38, 2-10
 - \$WS_IF, 2-10, 3-16
 - \$WS_SD, 1-32
 - \$WS_SF, 1-38
 - \$WS_SF2, 1-39
 - \$WS_SF4, 1-39
- EPOC
 - operating system, 1-1
 - PC version, 1-59
- error
 - cleaning up, 1-11
 - handling window server, 2-7
 - panic numbers window server, 1-12
- error handling
 - window server, 1-11
- escape-key
 - event, 5-8
- event
 - W_KEY_MODE, 1-2
 - W_MOUSE_DOWN, 5-9
 - W_MOUSE_OUTSIDE, 5-9
 - WM_ACTIVE, 1-53, 5-10
 - WM_ATTACHED, 2-25, 5-8
 - WM_BACKGROUND, 5-6
 - WM_CANCELLED, 1-28, 5-6, 5-10
 - WM_COMMAND, 5-7
 - WM_DATE_CHANGED, 1-3, 5-7
 - WM_DEICONISE, 1-16, 2-8, 5-8
 - WM_DETACHED, 2-25, 5-8
 - WM_ESCAPE, 5-8
 - WM_FOREGROUND, 2-25, 5-6
 - WM_KEY, 1-15, 1-51, 5-3
 - WM_KEYBOARD_STATE_CHANGE, 5-8
 - WM_MOUSE, 1-52, 5-9
 - WM_MOUSE_MOVE, 5-9
 - WM_MOUSE_PRESS, 5-9
 - WM_MOUSE_RELEASE, 1-53, 5-9
 - WM_ON, 5-7, 5-11
 - WM_REDRAW, 3-12, 5-6
 - WM_RUBBER, 1-53, 5-10, 5-14
 - WM_RUBBER_BAND_INIT, 1-54, 5-9
 - WM_TASK_KEY, 1-15, 5-7
 - WM_TASK_UPDATE, 5-7
 - WM_USER_MSG, 5-7, 5-10
 - WS_EVENT_UNION, 5-1
- event function
 - wCancelGetEvent, 5-10
 - wGetEvent, 1-51
 - wGetEvent get async, 1-14
 - wGetEventSpecial, 1-51
 - wGetEventSpecial get async, 1-14
 - wGetEventWait, 1-51
 - wGetEventWait event get, 1-14
 - wSendCommand, 5-10
 - wUserMsg, 5-10
- event selected
 - WE_ESC, 5-2
 - WE_KEY, 5-2
 - WE_MOUSE, 5-2
 - WE_NORMAL, 5-2
 - WE_OTHERS, 5-2
 - WE_REDRAW, 5-2
 - WE_STATUS, 5-2
- event selected update
 - wGetEventUpdate, 5-2
- event types
 - described, 5-3
- events
 - activation, 5-10
 - asynchronous, 1-28, 5-2
 - attachment, 5-8
 - background, 5-6

- cancelled, 5-6, 5-10
- client command, 5-7, 5-10
- date changed, 5-7
- de-iconise, 5-8
- detachment, 5-8
- end rubber band, 5-10
- escape-key, 5-8
- foreground, 5-6
- getting next, 5-1
- inform key handler, 5-7
- key, 1-14
- key presses, 1-51
- keyboard capturing, 5-15
- keyboard state change, 5-8
- large screen, 5-8
- machine on, 5-7
- mouse, 1-14, 1-52, 5-9
- non window server, 1-28
- other sources, 1-28
- process termination, 5-7
- redraw, 1-24, 5-6
- rubber, 5-15
- selected, 5-2
- start rubber band, 5-9
- synchronous, 5-1
- time-stamping, 1-14
- types, 5-3
- update selected, 5-2
- user, 5-10
- user message, 5-7
- waiting, 5-1
- wGetEvent async, 5-2
- wGetEventWait sync, 5-1
- window server clients, 1-14
- WLIB functions, 5-10
- events selected async
 - wGetEventSpecial, 5-2
- fast fonts
 - described, 1-38
- file format
 - bitmaps, 1-36
 - fonts, 1-46
- files
 - bitmaps creating, 1-31
 - font, 1-42
- fill area
 - gClrRect, 4-12
- filling areas
 - modes, 4-12, 4-13
 - WLIB functions, 4-12
- flashing
 - cursor, 1-29
- FLK
 - font source file list file, 1-44
- flushing
 - client-side buffer, 1-13
 - commands window server, 2-5
 - window server, 1-11
- flushing commands
 - window server, 2-7
- FON
 - file, 1-42
- font group configure
 - gConfigureFonts, 4-18
- font group header read
 - gReadFontGroupHeader, 4-19
- font header read
 - gReadFontHeader, 4-19
- font information
 - structure, 4-16
- font internal
 - wSetSystemFont, 4-17
- font loading
 - gOpenFont, 4-15
- font open index
 - gOpenFontIndex, 4-18
- font position
 - gSetOpenAddress, 4-15
- font style
 - G_FONT_FLAG_ASCII, 4-16
 - G_FONT_FLAG_BOLD, 4-16
 - G_FONT_FLAG_CP850, 4-16
 - G_FONT_FLAG_ITALIC, 4-16
 - G_FONT_FLAG_SERIF, 4-16
 - G_STY_BOLD, 4-2
 - G_STY_DOUBLE, 4-2
 - G_STY_INVERSE, 4-2
 - G_STY_ITALIC, 4-2
 - G_STY_MONO, 4-2
 - G_STY_NORMAL, 4-2
 - G_STY_SUBSCRIPT, 4-2
 - G_STY_SUBSCRIPT2, 4-3
 - G_STY_SUPERSCRIPT, 4-2
 - G_STY_SUPERSCRIPT2, 4-3
 - G_STY_UNDERLINE, 4-2
- font system
 - wSetSystemFont, 4-17
- font type system
 - W_SYSTEM_FONT_INTERNAL_S3, 4-17
 - W_SYSTEM_FONT_INTERNAL_S3B, 4-17
 - W_SYSTEM_FONT_S3, 4-17
 - W_SYSTEM_FONT_S3B, 4-17
- font width table
 - gGetWidthTable, 4-17
- fonts
 - ascent, 4-14
 - baseline, 4-14
 - bitmaps, 1-38
 - body cell, 4-14
 - compiler wsfcamp.exe, 1-43
 - configure group, 4-18
 - descent, 4-14
 - fast, 1-38
 - file structure, 1-46
 - files, 1-42
 - files checksum, 1-46
 - files embedded, 1-47
 - files p_crc, 1-46
 - finding in file gSetOpenAddress, 1-42
 - FON files, 1-42
 - freeing, 4-16
 - group, 4-18
 - group header, 4-19
 - HC, 1-39
 - header, 4-19
 - high character, 4-14
 - horizontal leading, 4-14

- ID field, 4-3
- ID start WS_FONT_BASE, 1-38
- ID system WS_FONT_SYSTEM, 1-38
- information, 1-44, 4-16
- internal IDs, 4-17
- loading gOpenFont, 1-42
- loading gOpenFontIndex, 1-42
- low character, 4-14
- MC, 1-41
- monospaced, 1-38
- multiple, 4-18
- opening, 4-15, 4-18
- proportional, 1-38
- roman, 1-41
- ROM-based, 1-38, 4-3
- S3, 1-40
- S3a, 1-41
- source file list FLK file, 1-44
- source FSC file, 1-43
- structure, 1-46
- style, 1-45, 4-2
- style G_STY_BOLD, 1-45
- style G_STY_DOUBLE, 1-45
- style G_STY_INVERSE, 1-45
- style G_STY_ITALIC, 1-45
- style G_STY_MONO, 1-45
- style G_STY_NORMAL, 1-45
- style G_STY_UNDERLINE, 1-45
- swiss, 1-42
- system IDs, 4-17
- text, 1-38, 4-14
- vertical leading, 4-14
- width, 4-14
- width table, 4-17
- Workabout, 1-41
- Fonts
 - ID system from W_SERVER_INFO, 1-38
 - ID system S3/HC from \$WS_SF, 1-38
 - ID system S3a from \$WS_SFNTS, 1-38
 - information gFontInfo, 1-44
- fonts.h
 - header file, 1-39, 1-41
- foreground
 - client, 1-14
 - client switch task order, 2-8
 - event, 5-6
- freeing
 - bitmap sequences, 3-15
 - bitmaps, 4-28, 4-32
 - clocks, 3-22
 - fonts, 4-16
 - mouse icon, 3-23
 - permanent graphics contexts, 4-5
 - sprites, 3-16
 - temporary graphics contexts, 4-6
- FSC file
 - font source file, 1-43
- G_FONT_CONFIG
 - structure, 4-18
- G_FONT_FLAG_ASCII
 - font style, 4-16
- G_FONT_FLAG_BOLD
 - font style, 4-16
- G_FONT_FLAG_CP850
 - font style, 4-16
- G_FONT_FLAG_ITALIC
 - font style, 4-16
- G_FONT_FLAG_SERIF
 - font style, 4-16
- G_FONT_INFO
 - font information structure, 4-16
 - structure, 1-46
- G_GC
 - plane flag, 1-3
 - structure, 3-9, 4-1
- G_GC_FLAG_BOTH_PLANES
 - drawing mode, 4-3
- G_GC_FLAG_DOUBLE
 - drawing mode, 4-3
 - graphics flag, 1-2
- G_GC_FLAG_GREY_PLANE
 - drawing mode, 4-3
- G_SHADOW
 - structure, 4-23
- G_STY_BOLD
 - font style, 1-45, 4-2
- G_STY_DOUBLE
 - font style, 1-45, 4-2
- G_STY_INVERSE
 - font style, 1-45, 4-2
- G_STY_ITALIC
 - font style, 1-45, 4-2
- G_STY_MONO
 - font style, 1-45, 4-2
- G_STY_NORMAL
 - font style, 1-45, 4-2
- G_STY_SUBSCRIPT
 - font style, 4-2
- G_STY_SUBSCRIPT2
 - font style, 4-3
- G_STY_SUPERSCRIPT
 - font style, 4-2
- G_STY_SUPERSCRIPT2
 - font style, 4-3
- G_STY_UNDERLINE
 - font style, 1-45, 4-2
- G_TRMODE_CLR
 - bitmap transfer mode, 1-30
 - text mode, 4-2
 - text transfer mode, 1-45
- G_TRMODE_INV
 - bitmap transfer mode, 1-30
 - text mode, 4-2
 - text transfer mode, 1-45
- G_TRMODE_REPL
 - bitmap transfer mode, 1-30
 - text mode, 4-2
 - text transfer mode, 1-45
- G_TRMODE_SET
 - bitmap transfer mode, 1-30
 - text mode, 4-2
 - text transfer mode, 1-45
- gBorder
 - graphics output, 1-49
 - WLIB function, 4-10
- gBorder2
 - graphics output, 1-49

- WLIB function, 4-12
- gBorder2Rect
 - graphics output, 1-49
 - WLIB function, 4-10
- gBorderRect
 - graphics output, 1-49
 - WLIB function, 4-8
- gCheckBitmapID
 - WLIB function, 4-30
- gClrRect
 - graphics output, 1-49
 - WLIB function, 4-12
- gConfigureFonts
 - WLIB function, 4-18
- gCopyBit
 - bitmap WLIB function, 1-30
 - graphics output, 1-49
 - WLIB function, 4-29
- gCopyRect
 - graphics output, 1-49
 - WLIB function, 4-30
- gCreateBit
 - bitmap WLIB function, 1-30
 - WLIB function, 4-26
- gCreateGC
 - graphics context permanent, 1-50
 - WLIB function, 4-4
- gCreateGC0
 - graphics context permanent, 1-50
 - WLIB function, 4-5
- gCreateTempGC
 - graphics context temporary, 1-51
 - WLIB function, 4-5
- gCreateTempGC0
 - graphics context temporary, 1-51
 - WLIB function, 4-6
- gDrawBit
 - bitmap WLIB function, 1-30
 - graphics output, 1-49
 - WLIB function, 4-31
- gDrawBox
 - graphics output, 1-49
 - WLIB function, 4-8
- gDrawLine
 - graphics output, 1-49
 - WLIB function, 4-7
- gDrawObject
 - graphics output, 1-49
 - WLIB function, 4-12
- gDrawPolyLine
 - graphics output, 1-49
 - WLIB function, 4-7
- gEndMultiSave
 - WLIB function, 4-32
- gFillPattern
 - bitmap WLIB function, 1-30
 - graphics output, 1-49
 - WLIB function, 4-13
- gFontInfo
 - fonts WLIB function, 1-44
 - WLIB function, 4-16
- gFreeTempGC
 - graphics context free, 1-51
 - WLIB function, 4-6
- gGetBit
 - bitmap WLIB function, 1-30
 - WLIB function, 4-31
- gGetWidthTable
 - WLIB function, 4-17
- gInitBit
 - bitmap WLIB function, 1-30
 - WLIB function, 4-31
- gInitMultiSave
 - WLIB function, 4-32
- gInquireChecksum
 - WLIB function, 4-32
- gInvObloid
 - graphics output, 1-49
 - WLIB function, 4-13
- gmode
 - graphics modes, 4-2
- gOpenBit
 - bitmap WLIB function, 1-30
 - WLIB function, 4-27
- gOpenFont
 - fonts WLIB function, 1-42
 - WLIB function, 4-15
- gOpenFontIndex
 - fonts WLIB function, 1-42
 - WLIB function, 4-18
- gOpenMouseIcon
 - WLIB function, 3-23
- gPeekBit, 1-35
 - WLIB function, 4-30
- gPrintBoxText
 - graphics output, 1-49
 - text drawing WLIB function, 1-44
 - WLIB function, 4-20
- gPrintClipText
 - graphics output, 1-49
 - text drawing WLIB function, 1-44
 - WLIB function, 4-20
- gPrintText
 - graphics output, 1-49
 - text drawing WLIB function, 1-44
 - WLIB function, 4-20
- gQueryBit
 - WLIB function, 4-31
- graphics
 - adding output functions, 4-33
 - DYL, 4-33
 - G_GC_FLAG_DOUBLE flag, 1-2
 - output, 2-10
 - structures, 1-48
 - WLIB functions, 1-47
- graphics contexts
 - current, 1-50
 - free gFreeTempGC, 1-51
 - free wEndRedraw, 1-51
 - gCreateGC, 4-4
 - gCreateGC0, 4-5
 - gCreateTempGC, 4-5
 - gCreateTempGC0, 4-6
 - gFreeTempGC, 4-6
 - gSetGC, 4-6
 - gSetGC0, 4-7
 - overview, 1-50, 4-1
 - permanent, 1-50, 4-4

- permanent freeing, 4-5
- permanent gCreateGC, 1-50
- permanent gCreateGC0, 1-50
- set specific gSetGC, 1-50
- set specific gSetGC0, 1-50
- setting, 4-6
- temporary, 1-51, 4-5
- temporary & permanent, 1-50
- temporary freeing, 4-6
- temporary gCreateTempGC, 1-51
- temporary gCreateTempGC0, 1-51
- temporary wBeginRedrawGC, 1-51
- temporary wBeginRedrawGC0, 1-51
- temporary wBeginRedrawWinGC, 1-51
- temporary wBeginRedrawWinGC0, 1-51
- wFree, 4-5
- windows, 1-18
- graphics DYL call
 - wCallDYL, 4-33
- graphics DYL enquiry call
 - wCallDYLReply, 4-33
- graphics DYL load
 - wLoadDYL, 4-33
- graphics functions
 - not to current graphics context, 1-49
 - to current graphics context, 1-49
- graphics modes
 - gmode, 4-2
- graphics objects
 - gDrawObject, 4-12
 - type attributes, 4-12
 - types, 4-12
- graphics output
 - gBorder, 1-49
 - gBorder2, 1-49
 - gBorder2Rect, 1-49
 - gBorderRect, 1-49
 - gClrRect, 1-49
 - gCopyBit, 1-49
 - gCopyRect, 1-49
 - gDrawBit, 1-49
 - gDrawBox, 1-49
 - gDrawLine, 1-49
 - gDrawObject, 1-49
 - gDrawPolyLine, 1-49
 - gFillPattern, 1-49
 - gInvObloid, 1-49
 - gPrintBoxText, 1-49
 - gPrintClipText, 1-49
 - gPrintText, 1-49
 - gShadowText, 1-49
 - gXPrintText, 1-49
 - not to current graphics context, 1-49
 - to current graphics context, 1-49
 - wCancelBusyMsg, 1-49
 - wDrawButton, 1-49
 - wDrawButton2, 1-49
 - wDrawTextCursor, 1-49
 - wEraseTextCursor, 1-49
 - wInfoMsg, 1-49
 - wInfoMsgCorner, 1-49
 - wsAlertA, 1-50
 - wsAlertCancel, 1-50
 - wsAlertUpdate, 1-50
 - wsAlertW, 1-50
 - wsCreateClock, 1-49
 - wsCreateClock2, 1-49
 - wScrollRect, 1-49
 - wScrollWin, 1-49
 - wsDisable, 1-50
 - wsDisableTemp, 1-50
 - wsEnable, 1-50
 - wsEnableTemp, 1-50
 - wSetBusyMsg, 1-49
 - wsSelectList, 1-50
 - wsSetClock, 1-49
 - wsSetList, 1-50
 - wsStatusWindow, 1-50
 - wsUpdate, 1-50
 - wTextCursor, 1-49
- gReadFontGroupHeader
 - WLIB function, 4-19
- gReadFontHeader
 - WLIB function, 4-19
- grey
 - cursor, 1-29
 - plane, 1-3, 1-19, 3-8, 4-1, 4-13
- gSaveBit
 - bitmap WLIB function, 1-31
 - WLIB function, 4-28
- gSaveMultiBit
 - bitmap WLIB function, 1-31
 - WLIB function, 4-32
- gSaveMultiRect
 - bitmap WLIB function, 1-31
 - WLIB function, 4-32
- gSaveRect
 - bitmap WLIB function, 1-31
 - WLIB function, 4-28
- gSetGC
 - graphics context specific, 1-50
 - WLIB function, 4-6
- gSetGC0
 - graphics context specific, 1-50
 - WLIB function, 4-7
- gSetOpenAddress
 - bitmap WLIB function, 1-30
 - fonts WLIB function, 1-42
 - WLIB function, 4-15
- gShadowText
 - graphics output, 1-49
 - text drawing WLIB function, 1-44
 - WLIB function, 4-23
- gTextCount
 - text layout WLIB function, 1-44
 - WLIB function, 4-17
- gTextWidth
 - text layout WLIB function, 1-44
 - WLIB function, 4-16
- gXPrintText
 - graphics output, 1-49
 - text drawing WLIB function, 1-44
 - WLIB function, 4-21
- HC
 - fonts, 1-39
 - replacing the shell, 1-56
 - shell example code, 1-56
 - task switching, 1-15

- header file
 - for multi bitmap files, 1-31
 - wlib.h, 1-8
- header files
 - fonts.h, 1-39, 1-41
- high character
 - fonts, 4-14
- hook notifier
 - process, 1-55
 - window server option, 1-5
- horizontal leading
 - fonts, 4-14
- hotkeys
 - capturing, 5-12
 - implementing, 5-12
- hot-spot
 - mouse icon, 1-52
- hung-up
 - redrawing delays, 1-26
 - window server flag, 1-56
- icon ID
 - WS_DEFAULT_ICON, 1-3
- icon ID grey
 - WS_DEFAULT_ICON+1, 1-3
- iconised
 - client, 1-16
 - client mark as, 2-9
- icons
 - mouse, 1-52, 3-22
- include file
 - key code wskeys.h, 6-1
- inform key handler
 - event, 5-7
- input
 - rubber band mode, 1-53
- internal fonts
 - IDs, 4-17
- invalidate function
 - wInvalidateRect, 3-12
 - wInvalidateWin, 3-12
- invalidating
 - windows, 1-24
 - WLIB function, 3-12
- invert obloid
 - gInvObloid, 4-13
- invisible window
 - wMakeInvisible, 1-28
- key
 - pause disable wDisablePauseKey, 1-52
 - pause enable wEnablePauseKey, 1-52
 - specific cancel wCancelCaptureKey, 1-51
 - specific wCaptureKey, 1-51
 - tasks setting, 1-52
- key capture
 - wCaptureKey, 5-11
- key capture off
 - wCancelCaptureKey, 5-12
- key code
 - diamond, 5-5
 - Series 3c and Siena update, 6-1
 - W_FUNC_MODIFIER, 6-2
 - W_KEY_BACKLIGHT, 5-5
 - W_KEY_CALC_CHNG_SIGN, 6-2
 - W_KEY_CALC_CLEAR, 6-2
 - W_KEY_CALC_DECIMAL, 6-2
 - W_KEY_CALC_MEM_CLEAR, 6-2
 - W_KEY_CALC_MEM_INPUT, 6-2
 - W_KEY_CALC_MEM_MINUS, 6-2
 - W_KEY_CALC_MEM_PLUS, 6-2
 - W_KEY_CALC_MEM_RECALL, 6-2
 - W_KEY_CALC_PERCENT, 6-2
 - W_KEY_CAPS_LOCK, 5-5
 - W_KEY_DELETE_LEFT, 5-4
 - W_KEY_DELETE_RIGHT, 5-4
 - W_KEY_DIAMOND, 5-5
 - W_KEY_DOWN, 5-4
 - W_KEY_END, 5-4
 - W_KEY_ESCAPE, 5-4
 - W_KEY_HELP, 5-5
 - W_KEY_HOME, 5-4
 - W_KEY_INFO, 5-5
 - W_KEY_IR_BRING, 6-2
 - W_KEY_IR_LINK, 6-2
 - W_KEY_IR_SEND, 6-2
 - W_KEY_LCD, 5-5
 - W_KEY_LCD_MINUS, 5-5
 - W_KEY_LEFT, 5-4
 - W_KEY_MENU, 5-5
 - W_KEY_MODE, 1-15, 5-5
 - W_KEY_OFF, 5-6
 - W_KEY_ON, 5-5
 - W_KEY_PAGE_DOWN, 5-4
 - W_KEY_PAGE_UP, 5-4
 - W_KEY_RETURN, 5-4
 - W_KEY_RIGHT, 5-4
 - W_KEY_TAB, 5-4, 6-1
 - W_KEY_TASK, 5-4
 - W_KEY_UP, 5-4
 - W_KEY_VOICE, 5-4
 - W_RUSSIAN_MODIFIER, 6-2
 - wskeys.h, 6-1
- key code applications
 - W_KEY_APPn, 5-5
- key modifier
 - W_CAPS_MODIFIER, 5-3, 5-9
 - W_CTRL_MODIFIER, 5-3, 5-9
 - W_NUM_LOCK_MODIFIER, 5-3, 5-9
 - W_PSION_MODIFIER, 5-3, 5-9
 - W_SHIFT_MODIFIER, 5-3, 5-9
- key press
 - diamond, 1-15
- keyboard
 - events capturing, 5-15
 - input, 1-14, 1-51
 - key press events, 1-51
 - rubber band, 5-15
 - Series 3c and Siena update, 6-1
 - state change event, 5-8
 - update for Series 3c and Siena, 6-1
- key-click
 - disable, 2-26
 - enable, 2-26
- keys
 - application handler, 1-15
 - cancel back task switch keys, 5-13
 - cancel task switch keys, 5-12
 - capturing, 1-51, 5-11
 - code, 1-51, 5-3

- code - Series 3c and Siena update, 6-1
- code wskeys.h, 6-1
- count, 1-51, 5-3
- events, 1-14
- key press events, 1-51
- modifier, 5-3
- repeat count, 1-51, 5-3
- set back task switch keys, 5-13
- set task switch keys, 5-12
- large screen
 - event, 5-8
- LCD display resolution
 - by machine, 2-5
- leading horizontal
 - fonts, 4-14
- leading vertical
 - fonts, 4-14
- leaving
 - disable, 2-7
 - enable, 2-7
- libraries
 - WLIB using, 1-8
- line drawing
 - gDrawBox, 4-8
 - gDrawLine, 4-7
 - gDrawPolyLine, 4-7
 - WLIB functions, 4-7
- link paste
 - window server services, 1-6
- low character
 - fonts, 4-14
- machine on
 - event, 5-7
- machine type
 - from wConnect, 1-2
- magic static
 - DatStatusNamePtr, 2-13
- MC
 - fonts, 1-41
 - task switching, 1-16
 - wAttachToClient, 1-16
 - wAttachToForegroundClient, 1-16
- MC200
 - Fonts system ID, 1-39
 - mouse, 5-9
- MC400
 - Fonts system ID, 1-39
 - mouse, 5-9
- message
 - cancel busy, 2-12
 - display alerts, 2-12
 - display brief, 2-11
 - display busy, 2-11
- message constant
 - W_INFO_MSG_MAX_LEN, 2-11
- message flag
 - W_CORNER_BOTTOM_LEFT, 2-11
 - W_CORNER_BOTTOM_RIGHT, 2-11
 - W_CORNER_TOP_LEFT, 2-11
 - W_CORNER_TOP_RIGHT, 2-11
- mode display
 - diamond key, 2-20
- mode list
 - status window, 2-23
- modifier
 - keys, 5-3
- monospaced fonts
 - sets, 1-38
- mouse
 - capturing, 1-53, 5-13
 - event, 5-9
 - events, 1-14, 1-52, 5-9
 - freeing icon, 3-23
 - grabbing, 1-53
 - icons, 1-52, 3-22
 - input, 1-52
 - releasing, 5-13
 - rubber band, 5-15
- mouse capture
 - wCaptureMouse, 5-13
- mouse icon
 - gOpenMouseIcon, 3-23
 - hot-spot, 1-52
 - W_WIN_MI_STANDARD, 1-52
 - wFree, 3-23
- mouse icon attribute
 - W_WIN_MI_CROSS, 3-22
 - W_WIN_MI_HSLIDE, 3-23
 - W_WIN_MI_LEFT, 3-23
 - W_WIN_MI_MARGIN, 3-23
 - W_WIN_MI_MOVE, 3-23
 - W_WIN_MI_NULL, 1-52, 3-22
 - W_WIN_MI_PG_DOWN, 3-23
 - W_WIN_MI_PG_UP, 3-23
 - W_WIN_MI_PUSHER, 3-22
 - W_WIN_MI_RESIZE, 3-23
 - W_WIN_MI_RIGHT, 3-23
 - W_WIN_MI_STANDARD, 3-22
 - W_WIN_MI_TEXT, 3-22
 - W_WIN_MI_TO_BIG, 3-23
 - W_WIN_MI_TO_SMALL, 3-23
 - W_WIN_MI_VSLIDE, 3-23
- mouse icon position
 - gSetOpenAddress, 4-15
- mouse release
 - wReleaseMouse, 5-13
- multiple
 - bitmaps, 4-30
 - fonts, 4-18
- multi-tasking
 - redraw response, 1-26
 - window destroying and, 3-6
- normal
 - plane, 1-3, 1-19, 3-8, 4-1, 4-13
- notifier
 - hook the process, 1-55
 - window server option, 1-5
- notify
 - process SYS\$NTFY, 1-54
- obloid invert
 - gInvObloid, 4-13
- on event enable
 - wInformOn, 5-11
 - wInformOnAll, 5-11
- opening
 - bitmaps, 4-31
 - fonts, 4-15, 4-18

- p_crc
 - font files, 1-46
 - function, 1-37
- p_enter
 - window server, 1-11
- p_execc
 - sub-process create, 1-26
- P_EXTENT
 - structure, 1-48, 3-1
- P_FSIG
 - structure, 1-37, 1-46
- p_iowait
 - and window server events, 1-28
- p_leave
 - window server, 1-11
- p_panic
 - window server, 1-12
- P_POINT
 - structure, 1-48, 3-1, 4-30
- P_RECT
 - structure, 1-48, 4-29
- p_resume
 - function, 1-14
- panic
 - W_PANIC_SPRITE, 3-16
 - W_PANIC_SPRITE_EXISTS, 3-16
- panic numbers
 - window server, 1-12
- parent
 - window, 1-18
- password support
 - window server option, 1-6
- pause
 - client disable, 2-10
 - client enable, 2-9
- pausing
 - client, 1-52
- PC
 - EPOC, 1-59
- PCX file
 - from screen capture, 1-33
 - to bitmap file, 1-31
- pcxsave.c
 - example program screen capture, 1-33
- pcxScreenSave
 - screen capture, 1-33
- permanent
 - status window, 2-19
- PH file
 - multi bitmap header file, 1-31
- PIC file
 - from a PCX file, 1-31
- PIC_HEAD
 - structure, 1-36
- pixel
 - coordinates, 1-47
 - double sized mode, 4-1
 - screen resolutions, 2-5
- pixel coordinates
 - bitmaps, 1-47
- plane
 - black, 1-3, 1-19, 4-1
 - grey, 1-3, 1-19, 3-8, 4-1, 4-13
 - normal, 1-3, 1-19, 4-1, 4-13
- plane flag
 - in G_GC, 1-3
- PLIB
 - library, 1-8
 - startup module, 1-9, 2-1
 - startup module MC, 1-10
- PLK file
 - multi bitmap files, 1-31
- polygon drawing
 - WLIB function, 4-7
- priority
 - changing clients, 2-7
 - client, 1-16
 - compute mode end wEndCompute, 2-8
 - compute mode setting wStartCompute, 2-8
 - redraw events, 1-27
 - wEndCompute, 1-17
 - wSetPriorityControl, 2-8
 - wStartCompute, 1-17
- process
 - SYSSFSRV, 1-54
 - SYSSMANG, 1-54
 - SYSSNTFY, 1-54
 - SYSSNULL, 1-54
 - SYSSHLL, 1-15, 1-54
 - SYSSWSRV, 1-54
 - wSystem, 1-55
- process termination
 - event, 5-7
- proportional fonts
 - sets, 1-38
- reading bitmaps
 - WLIB function, 4-30
- redraw
 - event, 5-6
 - events, 1-24
 - priority, 1-27
 - priority bit W_WIN_PRIORITY, 1-27
 - responsively, 1-26
 - update region, 1-24
 - validating before, 1-25
- redraw function
 - wBeginRedraw, 3-9
 - wBeginRedrawGC, 3-9
 - wBeginRedrawGC0, 3-10
 - wBeginRedrawWin, 3-9
 - wBeginRedrawWinGC, 3-10
 - wBeginRedrawWinGC0, 3-10
 - wEndRedraw, 3-11
- redrawing
 - wBeginRedraw WLIB functions, 1-26
 - windows, 1-24, 1-25
 - windows variants, 3-8
- redraws
 - avoiding using bitmaps, 1-22
 - avoiding using bitmaps example code, 1-22
- releasing mouse
 - WLIB function, 5-13
- repeat count
 - key, 1-51
 - keys, 5-3
- reserved static
 - see magic static, 2-4

- ROM
 - built in grey bitmap, 1-30
- roman
 - fonts, 1-41
- ROM-based fonts
 - IDs, 4-3
 - sets, 1-38
- root
 - window, 1-18
- rubber band
 - arrow keys, 1-54
 - enter mode `wRubberBand`, 1-54
 - input mode, 1-53
 - keyboard events, 5-15
 - mouse events, 5-15
 - WLIB functions, 5-14
- rubber band attribute
 - `W_BAND_COMPLETE_ON_UP`, 5-15
 - `W_BAND_GRID_SNAP`, 5-15
 - `W_BAND_GRID_SNAP_SIZE`, 5-15
 - `W_BAND_INNER`, 5-14
 - `W_BAND_OUTER`, 5-14
 - `W_BAND_RESIZE`, 5-14
 - `W_BAND_START`, 5-14
- rubber band flag
 - `WM_BAND_CANCEL`, 5-15
 - `WM_BAND_ERROR`, 5-15
 - `WM_BAND_MOVE`, 5-15
 - `WM_BAND_NOMOVE`, 5-15
 - `WM_BAND_RESIZE`, 5-15
- rubber band start
 - `wRubberBand`, 5-14
- S3
 - fonts, 1-40
- S3a
 - fonts, 1-41
- saving bitmaps
 - WLIB function, 4-28
- saving multiple bitmaps
 - WLIB function, 4-32
- scapt
 - screen capture example program, 1-36
- screen
 - coordinates, 1-18
 - double pixel mode, 1-2
 - event, 5-8
- screen capture
 - disabling, 1-32
 - example code, 1-33
 - HC, 1-33
 - `pcxScreenSave` function, 1-33
 - scapt example program, 1-36
 - to a bitmap, 1-31
 - to bitmap file, 1-32
 - to PCX file `pcxsave.c`, 1-33
- screen resolution
 - by machine, 2-5
 - by machine update, 6-1
 - Series 3c and Siena update, 6-1
- screen sizes
 - by machine update, 6-1
- scrolling
 - rectangle, 3-7
 - window, 3-7, 3-8
- selected events
 - WLIB function, 5-2
- Series 3
 - compatibility mode, 1-12
 - task switching, 1-15
- Series 3a
 - task switching, 1-15
- server
 - window server connecting to, 1-8
- setting graphics contexts
 - WLIB functions, 4-6
- shadowed text
 - WLIB functions, 4-23
- shadows
 - drawing, 4-10
- shell
 - event, 5-7
 - HC replacing the shell, 1-56
 - HC shell example code, 1-56
 - process, 1-15, 1-54
- sibling
 - window, 1-18, 3-7
- sprite
 - structure, 3-15
- sprite attribute
 - `W_SPRITE_CLIP_CHILDREN`, 3-15
- sprite function
 - `wCreateSprite`, 3-15
 - `wFree`, 3-16
 - `wSetSprite`, 3-16
- sprites
 - animated bitmaps, 1-3
 - animated graphics, 1-23
 - animation, 3-15
 - black plane, 1-23
 - changing, 3-16
 - creating, 3-15
 - freeing, 3-16
 - grey plane, 1-23
 - normal plane, 1-23
 - plane grey, 1-23
 - `W_SPRITE_CLIP_CHILDREN`, 1-23
 - `wCreateSprite`, 1-23
 - `wFree`, 1-23
 - `wSetSprite`, 1-23
- start rubber band
 - event, 5-9
- startup module
 - CLIB, 1-8, 2-1
 - CLIB MC, 1-10
 - PLIB, 1-9, 2-1
 - PLIB MC, 1-10
- start-up system
 - process, 1-54
- status window
 - compatibility mode, 2-21
 - disable, 2-22
 - drawing, 1-6
 - extent get, 2-21
 - mode list, 2-23
 - overview, 2-19
 - permanent, 2-19

- permanent enable, 2-21
- permanent set state, 2-21
- select position set, 2-23
- Series 3c and Siena update, 6-3
- Series 3c update, 6-3
- Siena update, 6-3
- state get, 2-23
- temporary, 2-19
- temporary disable, 2-22
- temporary enable, 2-22
- update displayed, 2-22
- wStatusWindow permanent, 2-21
- status window flag
 - W_STATUS_WIN_NO_DIAMOND, 2-23
 - W_STATUS_WINDOW_BIG, 2-21
 - W_STATUS_WINDOW_CTBY, 2-21
 - W_STATUS_WINDOW_ICON, 2-23
 - W_STATUS_WINDOW_OFF, 2-21
 - W_STATUS_WINDOW_SMALL, 2-21
- structures
 - bitmap open, 4-26, 4-27
 - clock, 3-19
 - CONNECT_INFO, 2-3
 - E_CONFIG, 3-16
 - G_FONT_CONFIG, 4-18
 - G_FONT_INFO, 1-46, 4-16
 - G_GC, 3-9, 4-1
 - G_SHADOW, 4-23
 - graphics, 1-48
 - P_EXTENT, 1-48, 3-1
 - P_FSIG, 1-37
 - P_POINT, 1-48, 3-1, 4-30
 - P_RECT, 1-48, 4-29
 - PIC_HEAD, 1-36
 - sprite, 3-15
 - W_SERVER_INFO, 1-38, 2-3
 - W_SUPPORT_INFO, 2-26
 - W_WINDATA, 3-1
 - wMainGc, 2-2
 - wMainWid, 2-2
 - WMSG_KEY, 5-3
 - WMSG_MOUSE, 5-9
 - WMSG_RUBBER, 5-15
 - WS_EV, 5-1
 - WS_PIC_HEADER, 1-37
 - WSERV_SPEC, 2-3
 - wSpec, 2-2
- Structures
 - WS_FONT_FILE_HEADER, 1-46
- style
 - fonts, 1-45, 4-2
- sub-process
 - create p_execc, 1-26
- swiss
 - fonts, 1-42
- synchronous
 - events, 5-1
- SYSS\$CONS
 - display process on MC, 1-7
- SYSS\$FSRV
 - process, 1-54
- SYSS\$MANG
 - process, 1-54
- SYSS\$NTFY
 - process, 1-54
- SYSS\$NULL
 - process, 1-54
- sys\$shll, 2-9
- SYSS\$SHLL
 - process, 1-54
- SYSS\$WSRV
 - process, 1-54
 - window server process, 1-1
- system fonts
 - IDs, 4-17
- system modal
 - wCancelSystemModal, 1-17
 - wSystemModal, 1-17
- system start-up
 - process, 1-54
- system type
 - from wConnect, 1-2
- system-modal
 - client, 1-17, 2-9
- task key
 - back set wSetBackTaskKey, 1-52
 - cancel wCancelTaskKey, 1-52
 - set wSetTaskKey, 1-52
- Task key
 - S3 and 3a, 1-52
- task key back cancel
 - wCancelBackTaskKey, 1-52
- task keys
 - setting, 1-52
- task order
 - client position, 2-8
- task switching
 - cancel back task switch keys, 5-13
 - cancel task switch keys, 5-12
 - HC, 1-15
 - MC, 1-16
 - Series 3, 1-15
 - Series 3a, 1-15
 - set back task switch keys, 5-13
 - set task switch keys, 5-12
 - window server, 1-14
- task switching back cancel
 - wCancelBackTaskKey, 5-13
- task switching key cancel
 - wCancelTaskKey, 5-12
- task switching key set
 - wSetTaskKey, 5-12
- task switching set back
 - wSetBackTaskKey, 5-13
- temporary
 - status window, 2-19
- text
 - button, 4-24, 4-25
 - cursor, 1-29, 3-12
 - drawing, 1-44
 - fonts, 1-38, 4-14
 - transfer mode, 1-45, 4-2
 - transfer mode G_TRMODE_CLR, 1-45
 - transfer mode G_TRMODE_INV, 1-45
 - transfer mode G_TRMODE_REPL, 1-45
 - transfer mode G_TRMODE_SET, 1-45
 - width, 4-16

- text count
 - gTextCount, 4-17
- text drawing
 - gPrintBoxText, 1-44
 - gPrintClipText, 1-44
 - gPrintText, 1-44
 - gShadowText, 1-44
 - gXPrintText, 1-44
- text mode
 - G_TRMODE_CLR, 4-2
 - G_TRMODE_INV, 4-2
 - G_TRMODE_REPL, 4-2
 - G_TRMODE_SET, 4-2
- text output
 - WLIB functions, 4-19
- text print
 - boxed, 4-20
 - clipped, 4-20
 - embellishment, 4-21
 - gPrintText, 4-20
 - shadowed, 4-23
- text print boxed
 - gPrintBoxText, 4-20
- text print clipped
 - gPrintClipText, 4-20
- text print embellished
 - gXPrintText, 4-21
- text print shadowed
 - gShadowText, 4-23
- text width
 - gTextCount, 1-44
 - gTextWidth, 1-44
 - wGetWidthTable, 1-44
- text width get
 - gTextWidth, 4-16
- textmode
 - transfer mode, 1-45, 4-2
- time-stamp
 - events, 1-14
- top-level
 - window, 1-18
- transfer mode
 - text, 1-45, 4-2
- update event selection
 - WLIB function, 5-2
- update region
 - redrawing, 1-24
- user
 - event, 5-10
- user message
 - event, 5-7
- validate function
 - wValidateRect, 3-11
 - wValidateWin, 3-11, 3-12
- validating
 - before drawing, 1-25
 - WLIB function, 3-11
- version_id
 - window server, 2-4
- versions
 - window server, 1-1
- vertical leading
 - fonts, 4-14
- visibility
 - window, 3-6
- visibility of
 - windows, 1-28
- visible window
 - wMakeVisible, 1-28
- W_BAND_COMPLETE_ON_UP
 - rubber band attribute, 5-15
- W_BAND_GRID_SNAP
 - rubber band attribute, 5-15
- W_BAND_GRID_SNAP_SIZE
 - rubber band attribute, 5-15
- W_BAND_INNER
 - rubber band attribute, 5-14
- W_BAND_OUTER
 - rubber band attribute, 5-14
- W_BAND_RESIZE
 - rubber band attribute, 5-14
- W_BAND_START
 - rubber band attribute, 5-14
- W_BORD_CORNER_1
 - border attribute, 4-11
- W_BORD_CORNER_2
 - border attribute, 4-11
- W_BORD_CORNER_4
 - border attribute, 4-11
- W_BORD_CUSHION
 - border attribute, 4-11
- W_BORD_OPEN
 - border attribute, 4-11
- W_BORD_SHADOW_D
 - border attribute, 4-11
- W_BORD_SHADOW_ON
 - border attribute, 4-11
- W_BORD_SHADOW_S
 - border attribute, 4-11
- W_CAPS_MODIFIER
 - key modifier, 5-3, 5-9
- W_CONNECT_AT_BACK
 - window server flag, 2-3
- W_CONNECT_CONNECTED
 - window server flag, 2-8
- W_CONNECT_DISABLE_LEAVES
 - window server flag, 2-3
 - window server option, 1-11
- W_CONNECT_PRIORITY
 - window server flag, 2-3, 2-7, 2-8
- W_CONNECT_SYSTEM_MODAL
 - window server flag, 2-3, 2-8
- W_CONNECT_USER_FLAG
 - window server flag, 2-3, 2-8
- W_CORNER_BOTTOM_LEFT
 - message flag, 2-11
- W_CORNER_BOTTOM_RIGHT
 - message flag, 2-11
- W_CORNER_TOP_LEFT
 - message flag, 2-11
- W_CORNER_TOP_RIGHT
 - message flag, 2-11
- W_CTBY_S3
 - compatibility mode, 2-6
- W_CTBY_S3_SCR
 - compatibility mode, 2-6

- W_CTRL_MODIFIER
 - key modifier, 5-3, 5-9
- W_CURSOR_GREY
 - cursor attribute, 3-12
- W_FUNC_MODIFIER
 - key code, 6-2
- W_INFO_MSG_MAX_LEN
 - message constant, 2-11
- W_KEY_APP1
 - application keys, 5-5, 6-2
- W_KEY_APP2
 - application keys, 5-5, 6-2
- W_KEY_APP3
 - application keys, 5-5, 6-2
- W_KEY_APP4
 - application keys, 5-5, 6-2
- W_KEY_APP5
 - application keys, 5-5, 6-2
- W_KEY_APP6
 - application keys, 5-5, 6-2
- W_KEY_APP7
 - application keys, 5-5, 6-2
- W_KEY_APP8
 - application keys, 5-5, 6-2
- W_KEY_APP9
 - application keys, 6-2
- W_KEY_APPn
 - application keys, 5-5
- W_KEY_BACKLIGHT
 - key code, 5-5
- W_KEY_CALC_CHNG_SIGN
 - key code, 6-2
- W_KEY_CALC_CLEAR
 - key code, 6-2
- W_KEY_CALC_DECIMAL
 - key code, 6-2
- W_KEY_CALC_MEM_CLEAR
 - key code, 6-2
- W_KEY_CALC_MEM_INPUT
 - key code, 6-2
- W_KEY_CALC_MEM_MINUS
 - key code, 6-2
- W_KEY_CALC_MEM_PLUS
 - key code, 6-2
- W_KEY_CALC_MEM_RECALL
 - key code, 6-2
- W_KEY_CALC_PERCENT
 - key code, 6-2
- W_KEY_CAPS_LOCK
 - key code, 5-5
- W_KEY_DELETE_LEFT
 - key code, 5-4
- W_KEY_DELETE_RIGHT
 - key code, 5-4
- W_KEY_DIAMOND
 - key code, 5-5
- W_KEY_DOWN
 - key code, 5-4
- W_KEY_END
 - key code, 5-4
- W_KEY_ESCAPE
 - key code, 5-4
- W_KEY_HELP
 - key code, 5-5
- W_KEY_HOME
 - key code, 5-4
- W_KEY_INFO
 - key code, 5-5
- W_KEY_IR_BRING
 - key code, 6-2
- W_KEY_IR_LINK
 - key code, 6-2
- W_KEY_IR_SEND
 - key code, 6-2
- W_KEY_LCD
 - key code, 5-5
- W_KEY_LCD_MINUS
 - key code, 5-5
- W_KEY_LEFT
 - key code, 5-4
- W_KEY_MENU
 - key code, 5-5
- W_KEY_MODE
 - event, 1-2
 - key code, 1-15, 5-5
- W_KEY_OFF
 - key code, 5-6
- W_KEY_ON
 - key code, 5-5
- W_KEY_PAGE_DOWN
 - key code, 5-4
- W_KEY_PAGE_UP
 - key code, 5-4
- W_KEY_RETURN
 - key code, 5-4
- W_KEY_RIGHT
 - key code, 5-4
- W_KEY_TAB
 - key code, 5-4, 6-1
- W_KEY_TASK
 - key code, 5-4
- W_KEY_UP
 - key code, 5-4
- W_KEY_VOICE
 - key code, 5-4
- W_MOUSE_DOWN
 - events, 5-9
- W_MOUSE_OUTSIDE
 - events, 5-9
- W_NUM_LOCK_MODIFIER
 - key modifier, 5-3, 5-9
- W_PANIC_SPRITE
 - panic, 3-16
- W_PANIC_SPRITE_EXISTS
 - panic, 3-16
- W_PSION_MODIFIER
 - key modifier, 5-3, 5-9
- W_RUSSIAN_MODIFIER
 - key code, 6-2
- W_SERVER_INFO
 - for system font ID, 1-38
 - structure, 1-38, 2-3
- W_SHIFT_MODIFIER
 - key modifier, 5-3, 5-9
- W_SPRITE_CLIP_CHILDREN
 - sprite attribute, 3-15
 - sprites, 1-23

- W_STATUS_WIN_NO_DIAMOND
 - status window flag, 2-23
 - window status flag, 2-23
- W_STATUS_WINDOW_BIG
 - status window flag, 2-21
- W_STATUS_WINDOW_CTBY
 - status window flag, 2-21
- W_STATUS_WINDOW_ICON
 - status window flag, 2-23
- W_STATUS_WINDOW_OFF
 - status window flag, 2-21
- W_STATUS_WINDOW_SMALL
 - status window flag, 2-21
- W_SUPPORT_CTBY_S3
 - window server flag, 2-26
- W_SUPPORT_GREY
 - window server flag, 2-26
- W_SUPPORT_INFO
 - structure, 2-26
- W_SYSTEM_FONT_INTERNAL_S3
 - font type system, 4-17
- W_SYSTEM_FONT_INTERNAL_S3B
 - font type system, 4-17
- W_SYSTEM_FONT_S3
 - font type system, 4-17
- W_SYSTEM_FONT_S3B
 - font type system, 4-17
- W_WIN_BACK_BITMAP
 - window attribute, 1-19, 3-2
- W_WIN_BACK_CLR
 - window attribute, 1-20, 1-25, 3-2
- W_WIN_BACK_CLR_NO_REDRAW
 - window attribute, 1-21, 3-2
- W_WIN_BACK_GREY_BITMAP
 - window attribute, 1-19, 3-2
- W_WIN_BACK_GREY_CLR
 - window attribute, 1-20, 1-25, 3-2
- W_WIN_BACK_GREY_CLR_NO_REDRAW
 - window attribute, 1-21
- W_WIN_BACK_GREY_NONE
 - window attribute, 1-25, 3-2
- W_WIN_BACK_GREY_NONE_NO_REDRAW
 - window attribute, 1-21
- W_WIN_BACK_GREY_SET
 - window attribute, 1-20, 1-25, 3-2
- W_WIN_BACK_GREY_SET_NO_REDRAW
 - window attribute, 1-21
- W_WIN_BACK_NONE
 - window attribute, 1-25, 3-2
- W_WIN_BACK_NONE_NO_REDRAW
 - window attribute, 1-21
- W_WIN_BACK_SET
 - window attribute, 1-20, 1-25, 3-2
- W_WIN_BACK_SET_NO_REDRAW
 - window attribute, 1-21, 3-2
- W_WIN_DOUBLE_PIXEL
 - window attribute, 1-2, 3-3
- W_WIN_FOREGROUND_ONLY
 - window attribute, 1-28, 3-3
- W_WIN_INACTIVE
 - window attribute, 1-53, 3-3
- W_WIN_INPUT_ONLY
 - window attribute, 1-53, 3-3
- W_WIN_MI_CROSS
 - mouse icon attribute, 3-22
- W_WIN_MI_HSLIDE
 - mouse icon attribute, 3-23
- W_WIN_MI_LEFT
 - mouse icon attribute, 3-23
- W_WIN_MI_MARGIN
 - mouse icon attribute, 3-23
- W_WIN_MI_MOVE
 - mouse icon attribute, 3-23
- W_WIN_MI_NULL
 - mouse icon attribute, 1-52, 3-22
- W_WIN_MI_PG_DOWN
 - mouse icon attribute, 3-23
- W_WIN_MI_PG_UP
 - mouse icon attribute, 3-23
- W_WIN_MI_PUSHER
 - mouse icon attribute, 3-22
- W_WIN_MI_RESIZE
 - mouse icon attribute, 3-23
- W_WIN_MI_RIGHT
 - mouse icon attribute, 3-23
- W_WIN_MI_STANDARD
 - mouse icon, 1-52
 - mouse icon attribute, 3-22
- W_WIN_MI_TEXT
 - mouse icon attribute, 3-22
- W_WIN_MI_TO_BIG
 - mouse icon attribute, 3-23
- W_WIN_MI_TO_SMALL
 - mouse icon attribute, 3-23
- W_WIN_MI_VSLIDE
 - mouse icon attribute, 3-23
- W_WIN_MOUSE_DRAG
 - window attribute, 1-52, 3-3
- W_WIN_MOUSE_GRAB, 1-53
 - window attribute, 3-4
- W_WIN_MOUSE_MOVE
 - window attribute, 1-52, 3-3
- W_WIN_NO_MOUSE
 - window attribute, 1-52, 3-3
- W_WIN_NO_REDRAW
 - window attribute, 1-20, 3-3
- W_WIN_PRIORITY
 - redraw priority, 1-27
 - window attribute, 3-3
- W_WIN_RUBBER_BAND
 - window attribute, 3-4
 - window attribute_, 3-4
- W_WIN_RUBBER_BAND_CAPTURE
 - window attribute, 1-54
- W_WIN_RUBBER_BAND_COMPLETE_ON_RELEASE
 - window attribute, 1-54
- W_WINDATA
 - structure, 3-1
- wAppKeyHandler
 - application keys, 1-15
- wAttachToClient
 - MC, 1-16
 - WLIB function, 2-26
- wAttachToForegroundClient
 - MC, 1-16
 - WLIB function, 2-26

-
- wBeginRedraw
 - redrawing WLIB functions, 1-25, 1-26
 - WLIB function, 3-9
 - wBeginRedrawGC
 - graphics context temporary, 1-51
 - WLIB function, 3-9
 - wBeginRedrawGC0
 - graphics context temporary, 1-51
 - WLIB function, 3-10
 - wBeginRedrawWin
 - WLIB function, 3-9
 - wBeginRedrawWinGC
 - graphics context temporary, 1-51
 - WLIB function, 3-10
 - wBeginRedrawWinGC0
 - graphics context temporary, 1-51
 - WLIB function, 3-10
 - wCallDYL
 - WLIB function, 4-33
 - wCallDYLReply
 - WLIB function, 4-33
 - wCancelBackTaskKey
 - task key back cancel, 1-52
 - WLIB function, 5-13
 - wCancelBusyMsg
 - graphics output, 1-49
 - WLIB function, 2-12
 - wCancelCaptureKey
 - key specific cancel, 1-51
 - WLIB function, 5-12
 - wCancelGetEvent
 - and window server events, 1-28
 - WLIB function, 5-10
 - wCancelSystemModal
 - system modal cancel, 1-17
 - WLIB function, 2-9
 - wCancelTaskKey
 - task key cancel, 1-52
 - WLIB function, 5-12
 - wCaptureKey
 - specific key, 1-51
 - WLIB function, 5-11
 - wCaptureMouse
 - WLIB function, 5-13
 - wChangeWinBitmap
 - bitmap sequences, 1-21
 - WLIB function, 3-15
 - wCheckPoint
 - WLIB function, 2-7
 - wCleanUp
 - WLIB function, 2-7
 - wClientIconised
 - WLIB function, 2-9
 - wClientInfo
 - WLIB function, 2-8
 - wClientPosition
 - WLIB function, 2-8
 - wCloseWindowTree
 - windows destroying, 1-27
 - WLIB function, 3-6
 - wCompatibilityMode
 - S3 display mode, 2-6
 - wConnect
 - WLIB function, 1-8, 2-1, 2-3
 - wCreateSprite
 - sprites, 1-23
 - WLIB function, 3-15
 - wCreateWindow
 - window create, 1-18, 1-27
 - WLIB function, 3-1, 3-4
 - wDetachClient
 - WLIB function, 2-26
 - wDisableKeyClick
 - WLIB function, 2-26
 - wDisableLeaves
 - WLIB function, 2-7
 - wDisablePauseKey
 - pause key disable, 1-52
 - WLIB function, 2-10
 - wDisconnect
 - WLIB function, 2-5
 - wDrawButton
 - graphics output, 1-49
 - WLIB function, 4-24
 - wDrawButton2
 - graphics output, 1-49
 - WLIB function, 4-25
 - wDrawTextCursor
 - graphics output, 1-49
 - WLIB function, 3-13
 - WE_ESC
 - event selected, 5-2
 - WE_KEY
 - event selected, 5-2
 - WE_MOUSE
 - event selected, 5-2
 - WE_NORMAL
 - event selected, 5-2
 - WE_OTHERS
 - event selected, 5-2
 - WE_REDRAW
 - event selected, 5-2
 - WE_STATUS
 - event selected, 5-2
 - wEnablePauseKey
 - pause key enable, 1-52
 - WLIB function, 2-9
 - wEndCompute
 - priority, 1-17
 - WLIB function, 2-8
 - wEndRedraw
 - graphics context free, 1-51
 - WLIB function, 3-11
 - wEraseTextCursor
 - cursor remove, 1-29
 - graphics output, 1-49
 - WLIB function, 3-13
 - wFlush
 - WLIB function, 2-5
 - wFree
 - bitmap free, 1-30
 - bitmap sequences, 1-21
 - sprites, 1-23
 - WLIB function, 3-15, 3-16, 3-22, 3-23, 4-5, 4-16, 4-28, 4-32
 - wGetCommand
 - WLIB function, 5-10
-

- wGetEvent
 - event, 1-28, 1-51
 - event get async, 1-14
 - WLIB function, 5-2
- wGetEventSpecial
 - event, 1-28, 1-51
 - event get async, 1-14
 - WLIB function, 5-2
- wGetEventUpdate
 - WLIB function, 5-2
- wGetEventWait
 - event, 1-51
 - event get, 1-14
 - WLIB function, 5-1
- wGetProcessList
 - WLIB function, 2-10
- wGetWidthTable
 - text layout WLIB function, 1-44
- wGetWindowPosition
 - window position get, 1-27
 - WLIB function, 3-7
- width
 - fonts, 4-14
- window
 - attributes, 3-1
 - attributes get, 3-5
 - attributes set, 3-5
 - back up bitmaps, 1-19
 - back up bitmaps advantages, 1-19
 - back up bitmaps disadvantages, 1-19
 - back up bitmaps memory, 1-20
 - background field, 3-2
 - background modes, 1-3
 - child, 1-18
 - clipping, 1-25
 - creating, 1-27, 3-1, 3-4
 - destroying, 1-27, 3-6
 - drawing, 1-25
 - drawing region, 1-19
 - flicker free drawing, 1-25
 - flicker free redrawing, 1-25
 - inactive, 1-53
 - initialising, 1-27, 3-1
 - invisible function, 3-6
 - mouse only input, 1-53
 - no-redraw, 1-20
 - ownership, 1-18
 - parent, 1-18
 - position get wGetWindowPosition, 1-27
 - position set wWindowPosition, 1-27
 - root, 1-18
 - scrolling, 1-28, 3-7
 - scrolling optimised, 1-3
 - sibling, 1-18, 3-7
 - sibling position get, 3-7
 - sibling position set, 3-7
 - status, 2-19
 - top-level, 1-18
 - trees, 1-18, 3-5
 - visibility, 1-28, 3-6
 - visible function, 3-7
 - W_WIN_DOUBLE_PIXEL flag, 1-2
 - wInvalidateWin, 1-24

- Window
 - offset get, 3-6
 - root reassign, 3-6
- window attribute
 - W_WIN_BACK_BITMAP, 1-19, 3-2
 - W_WIN_BACK_CLR, 1-20, 1-25, 3-2
 - W_WIN_BACK_CLR_NO_REDRAW, 1-21, 3-2
 - W_WIN_BACK_GREY_BITMAP, 1-19, 3-2
 - W_WIN_BACK_GREY_CLR, 1-20, 1-25, 3-2
 - W_WIN_BACK_GREY_CLR_, 3-2
 - W_WIN_BACK_GREY_CLR_NO_REDRAW, 1-21
 - W_WIN_BACK_GREY_NONE, 1-25, 3-2
 - W_WIN_BACK_GREY_NONE_, 3-2
 - W_WIN_BACK_GREY_NONE_NO_REDRAW, 1-21
 - W_WIN_BACK_GREY_SET, 1-20, 1-25, 3-2
 - W_WIN_BACK_GREY_SET_, 3-2
 - W_WIN_BACK_GREY_SET_NO_REDRAW, 1-21
 - W_WIN_BACK_NONE, 1-25, 3-2
 - W_WIN_BACK_NONE_, 3-2
 - W_WIN_BACK_NONE_NO_REDRAW, 1-21
 - W_WIN_BACK_SET, 1-20, 1-25, 3-2
 - W_WIN_BACK_SET_NO_REDRAW, 1-21, 3-2
 - W_WIN_DOUBLE_PIXEL, 3-3
 - W_WIN_FOREGROUND_ONLY, 1-28, 3-3
 - W_WIN_INACTIVE, 1-53, 3-3
 - W_WIN_INPUT_ONLY, 1-53, 3-3
 - W_WIN_MOUSE_DRAG, 1-52, 3-3
 - W_WIN_MOUSE_GRAB, 3-4
 - W_WIN_MOUSE_MOVE, 1-52, 3-3
 - W_WIN_NO_MOUSE, 1-52, 3-3
 - W_WIN_NO_REDRAW, 1-20, 3-3
 - W_WIN_PRIORITY, 3-3
 - W_WIN_RUBBER_BAND_, 3-4
 - W_WIN_RUBBER_BAND_CAPTURE, 1-54
 - W_WIN_RUBBER_BAND_COMPLETE_ON_RELEASE, 1-54
 - WS_WIN_BITMAP_GREY, 1-22
- window create
 - wCreateWindow, 1-18, 1-27
- window destroying
 - multi-tasking and, 3-6
- window function
 - wCreateWindow, 3-1
 - wInquireWindow, 3-1
 - wSetWindow, 3-1
- window invisible
 - wMakeInvisible, 1-28
- window offset
 - wInquireWindowOffset, 1-18
- window rectangle
 - wInvalidateRect, 1-24
- window scrolling
 - continuous, 1-29

- wScrollRect, 1-28
- wScrollWin, 1-28
- window server
 - buffered commands, 2-7
 - changes, 1-1
 - client information by process id, 2-8
 - client list get, 2-10
 - clients, 1-13
 - configure, 2-24
 - connect to, 1-8, 2-1, 2-3
 - connecting to example code, 2-4
 - disconnect from, 2-5
 - error handling, 1-11, 2-7
 - events and p_iowait, 1-28
 - events and wCancelGetEvent, 1-28
 - events and WM_CANCELLED, 1-28
 - flag WSERV_FLAG_HOOK_NOTIFIER, 1-55
 - flag WSERV_FLAG_HUNG_UP, 1-56
 - flag WSERV_FLAG_LOW_BATTERY_WARNINGS, 1-56
 - flag WSERV_FLAG_NO_NOTIFIER_REBOOT, 1-55
 - flag WSERV_FLAG_NO_PANIC_NOTIFY, 1-55
 - introduction, 1-1
 - process SYS\$WSRV, 1-1
 - Series 3c and Siena update, 6-1
 - supported features get, 2-26
 - task switching, 1-14
 - version 3, 1-7
 - version 3.5, 1-5
 - version 4, 1-2
 - version from wConnect, 1-2
 - version_id, 2-4
 - versions, 1-1
 - versions Series 3c and Siena update, 6-4
 - versions Series 3c update, 6-4
 - versions Siena update, 6-4
 - W_CONNECT_DISABLE_LEAVES flag, 1-11
 - wConnect WLIB function, 1-8
 - WLIB functions, 2-1
- window server configuration
 - see WSERV_FLAG, 2-24
- window server constant
 - WS_LAST_CLIENT_POSITION, 2-8
- window server flag
 - E_PRIORITY_BACK, 2-7
 - E_PRIORITY_FORE, 2-7
 - W_CONNECT_AT_BACK, 2-3
 - W_CONNECT_CONNECTED, 2-8
 - W_CONNECT_DISABLE_LEAVES, 2-3
 - W_CONNECT_PRIORITY, 2-3, 2-7, 2-8
 - W_CONNECT_SYSTEM_MODAL, 2-3, 2-8
 - W_CONNECT_USER_FLAG, 2-3, 2-8
 - W_SUPPORT_CTBY_S3, 2-26
 - W_SUPPORT_GREY, 2-26
- window server flags
 - see also WSERV_FLAG, 2-24
- window status flag
 - W_STATUS_WIN_NO_DIAMOND, 2-23
- window tree
 - wInitialiseWindowTree, 1-27
- window visible
 - wMakeVisible, 1-28
- windows
 - clipping child, 1-18
 - drawables, 1-18
 - graphics contexts, 1-18
 - invalidating, 1-24
 - overview, 1-18
 - overview additional info, 1-27
 - redrawing, 1-24, 1-25
 - redrawing variants, 3-8
- windows and bitmaps
 - drawables, 1-18
- windows destroying
 - wCloseWindowTree, 1-27
- wInfoMsg
 - graphics output, 1-49
 - WLIB function, 2-11
- wInfoMsgCorner
 - graphics output, 1-49
 - WLIB function, 2-11
- wInformOn
 - WLIB function, 5-11
- wInformOnAll
 - WLIB function, 5-11
- wInitialiseWindowTree
 - window tree, 1-27
 - WLIB function, 3-5
- wInquireCompatibility
 - compatibility mode get, 2-7
- wInquireStatusWindow
 - WLIB function, 2-23
- wInquireWindow
 - WLIB function, 3-1, 3-5
- wInquireWindowOffset
 - window offset, 1-18
 - WLIB function, 3-6
- wInvalidateRect
 - and invisible windows, 1-28
 - window rectangle, 1-24
 - WLIB function, 3-12
- wInvalidateWin
 - and invisible windows, 1-28
 - window, 1-24
 - WLIB function, 3-12
- WLIB
 - header file, 1-8
 - library with CLIB and PLIB, 1-8
- wLoadDYL
 - WLIB function, 4-33
- WM_ACTIVE
 - event, 1-53, 5-10
- WM_ATTACHED
 - event, 2-25, 5-8
- WM_BACKGROUND
 - event, 5-6
- WM_BAND_CANCEL
 - rubber band flag, 5-15
- WM_BAND_ERROR
 - rubber band flag, 5-15
- WM_BAND_MOVE
 - rubber band flag, 5-15

WM_BAND_NOMOVE
 rubber band flag, 5-15

WM_BAND_RESIZE
 rubber band flag, 5-15

WM_CANCELLED
 and window server events, 1-28
 event, 1-28, 5-6, 5-10

WM_COMMAND
 event, 5-7

WM_DATE_CHANGED
 event, 1-3, 5-7

WM_DEICONISE
 event, 1-16, 2-8, 5-8

WM_DETACHED
 event, 2-25, 5-8

WM_ESCAPE
 event, 5-8

WM_FOREGROUND
 event, 2-25, 5-6

WM_KEY
 event, 1-15, 1-51, 5-3

WM_KEYBOARD_STATE_CHANGE
 event, 5-8

WM_MOUSE
 event, 1-52, 5-9

WM_MOUSE_MOVE
 event, 5-9

WM_MOUSE_PRESS
 event, 5-9

WM_MOUSE_RELEASE
 event, 1-53, 5-9

WM_ON
 event, 5-7, 5-11

WM_REDRAW
 event, 3-12, 5-6

WM_RUBBER
 event, 1-53, 5-10, 5-14, 5-15

WM_RUBBER_BAND_INIT
 event, 1-54, 5-9

WM_TASK_KEY
 event, 1-15, 5-7

WM_TASK_UPDATE
 event, 5-7

WM_USER_MSG
 event, 5-7, 5-10

wMainGc
 structure, 2-2

wMainWid
 structure, 2-2

wMakeInvisible
 window WLIB function, 1-28
 WLIB function, 3-6

wMakeVisible
 window WLIB function, 1-28
 WLIB function, 3-7

WMSG_KEY
 structure, 5-3

WMSG_MOUSE
 structure, 5-9

WMSG_RUBBER
 structure, 5-15

Workabout
 fonts, 1-41
 task switching, 1-15

wReassignRootWindow
 WLIB function, 3-6

wReleaseMouse
 WLIB function, 5-13

wRubberBand
 enter rubber band mode, 1-54
 WLIB function, 5-14

WS_ALERT_B
 alert flag, 1-3

WS_CLOCK_AM_PM
 clock attribute, 3-17, 3-20

WS_CLOCK_BOX
 clock attribute, 3-20

WS_CLOCK_CENTERED
 clock attribute, 3-17, 3-20

WS_CLOCK_FORCE_ANALOG
 clock attribute, 3-17, 3-20

WS_CLOCK_FORCE_DIGITAL
 clock attribute, 3-17, 3-20

WS_CLOCK_FORMATTED
 clock attribute, 3-19

WS_CLOCK_GREY
 clock attribute, 3-20

WS_CLOCK_LARGE_ANALOG
 clock attribute, 3-17, 3-19

WS_CLOCK_MEDIUM
 clock attribute, 3-17, 3-19

WS_CLOCK_MEDIUM2
 clock attribute, 3-19

WS_CLOCK_SMALL_DIGITAL
 clock attribute, 3-17, 3-19

WS_CLOCK_WITH_DATE
 clock attribute, 3-17, 3-20

WS_CLOCK_WITH_SECONDS
 clock attribute, 3-17, 3-20

WS_CLOCK_XL_ANALOG
 clock attribute, 3-19

WS_DEFAULT_ICON
 icon ID, 1-3

WS_DEFAULT_ICON+1
 icon grey ID, 1-3

WS_EV
 structure, 5-1

WS_EVENT_UNION
 events, 5-1

WS_FONT_BASE
 define, 1-38

WS_FONT_FILE_HEADERS
 structure, 1-46

WS_FONT_SYSTEM
 define, 1-38

WS_LAST_CLIENT_POSITION
 window server constant, 2-8

WS_PIC_HEADER
 structure, 1-37

WS_WIN_BITMAP_GREY
 window attribute, 1-22

wsAlertA
 graphics output, 1-50
 WLIB function, 2-18

wsAlertCancel
 graphics output, 1-50

wsAlertUpdate
 graphics output, 1-50

- WLIB function, 2-19
- wsAlertW
 - alerts, 1-3
 - graphics output, 1-50
 - WLIB function, 2-14
- wsCreateClock
 - graphics output, 1-49
 - WLIB function, 3-17
- wsCreateClock2
 - graphics output, 1-49
 - WLIB function, 3-19
- wScrollRect
 - graphics output, 1-49
 - window scrolling, 1-28
 - WLIB function, 3-7
- wScrollWin
 - graphics output, 1-49
 - window scrolling, 1-28
 - WLIB function, 3-8
- wsDisable
 - graphics output, 1-50
 - WLIB function, 2-22
- wsDisableTemp
 - graphics output, 1-50
 - WLIB function, 2-22
- wsEnable
 - graphics output, 1-50
 - WLIB function, 2-21
- wsEnableTemp
 - graphics output, 1-50
 - WLIB function, 2-22
- wSendCommand
 - WLIB function, 5-10
- WSERV_FLAG
 - HOOK_NOTIFIER, 2-24
 - HUNG_UP_SW, 2-25
 - LOW_BATTERY_WARNINGS, 2-25
 - NO_NOTIFIER_REBOOT, 2-24
 - NO_PANIC_NOTIFY, 2-24
 - NO_SHELL_REBOOT, 2-24
 - SW_NO_CAPS, 2-25
 - SW_NO_LINK, 2-25
 - SW_NO_LOW_BATTERY, 2-25
 - SW_NO_PACKS, 2-25
 - UPDATE_MSGS, 2-24
- WSERV_FLAG_HOOK_NOTIFIER
 - window server flag, 1-55
- WSERV_FLAG_HUNG_UP
 - window server flag, 1-56
- WSERV_FLAG_LOW_BATTERY_WARNINGS
 - window server flag, 1-56
- WSERV_FLAG_NO_NOTIFIER_REBOOT
 - window server flag, 1-55
- WSERV_FLAG_NO_PANIC_NOTIFY
 - window server flag, 1-55
- WSERV_SPEC
 - structure, 2-3
- wSetBackTaskKey
 - task key back set, 1-52
 - WLIB function, 5-13
- wSetBusyMsg
 - graphics output, 1-49
 - WLIB function, 2-11
- wSetPriorityControl
 - WLIB function, 2-8
- wSetSprite
 - sprites, 1-23
 - WLIB function, 3-16
- wSetSystemFont
 - WLIB function, 4-17
- wSetTaskKey
 - task key set, 1-52
 - WLIB function, 5-12
- wSetWinBitmap
 - bitmap sequences, 1-21
 - bitmap WLIB function, 1-30
 - WLIB function, 3-14
- wSetWindow
 - WLIB function, 3-1, 3-5
- wsfcomp.exe
 - font file compiler, 1-43
- wspcx.exe
 - bitmap converter, 1-31
 - bitmap file converter, 1-31
- wSpec
 - structure, 2-2
- wsScreenExt
 - WLIB function, 2-21
- wsSelectList
 - graphics output, 1-50
 - WLIB function, 2-23
- wsSetClock
 - graphics output, 1-49
 - WLIB function, 3-22
- wsSetList
 - graphics output, 1-50
 - WLIB function, 2-23
- wsStatusWindow
 - graphics output, 1-50
- wStartCompute
 - priority, 1-17
 - WLIB function, 2-8
- wStartup
 - WLIB function, 2-1
- wStatusWindow
 - status window permanent set, 2-21
- wsUpdate
 - graphics output, 1-50
 - WLIB function, 2-22
- wSupportInfo
 - WLIB function, 2-26
- wSystem
 - process, 1-55
 - WLIB function, 2-24
- wSystemModal
 - system modal, 1-17
 - WLIB function, 2-9
- wTextCursor
 - cursor WLIB function, 1-29
 - graphics output, 1-49
 - WLIB function, 3-12
- wUserMsg
 - WLIB function, 5-10
- wValidateRect
 - WLIB function, 1-25, 3-11, 3-12
- wValidateWin
 - WLIB function, 1-25, 3-11

wWindowPosition
 window position set, 1-27
 WLIB function, 3-7