Microwindows Architecture

Greg Haerr
CEO
Century Software, Inc. (http://www.centurysoftware.com)

greg@censoft.com

Microwindows Architecture

by Greg Haerr

Table of Contents

1. Microwindows Architecture..6
1.1. Architecture..6
1.1.1. Layered Design...6
1.1.2. Device Drivers..6
1.1.2.1. Screen Driver...6
1.1.2.2. Mouse Driver...8
1.1.2.3. Keyboard Driver...9
1.1.3. MicroGUI - Device Independent Graphics Engine...............................9
1.1.4. Applications Programmer Interfaces..10
1.1.4.1. Microwindows API...11
1.1.4.2. Nano-X API...11
1.2. Device-Independent Engine Features...12
1.2.1. Graphics Engine Features and Implementation..................................12
1.2.1.1. Regions...12
1.2.1.2. Clipping...14
1.2.1.3. Line Drawing..15
1.2.1.4. Rectangles, Circles, Ellipses...15
1.2.1.5. Polygons..16
1.2.1.6. Area Fills...16
1.2.1.7. Fonts...16
1.2.1.8. Text Drawing..17
1.2.1.9. Color model and palettes...17
1.2.1.10. Image Drawing..18
1.2.1.11. Blitting...19
1.3. Microwindows API..19
1.3.1. Message-passing architecture...20
1.3.2. Window creation and destruction...21
1.3.3. Window showing, hiding and moving..22
1.3.4. Window painting..22
1.3.4.1. Client and screen coordinates...23
1.3.4.2. Device contexts...23

1.3.4.3. Graphics Drawing Functions 24
1.3.5. Utility functions 26
1.3.5.1. Setting window focus..27
1.3.5.2. Mouse capture 27
1.3.5.3. Rectangle and Region management...27
1.4. Nano-X API 29
1.4.1. Client/Server model 29
1.4.2. Events..30
1.4.3. Window creation and destruction...31
1.4.4. Window showing, hiding and moving 31
1.4.5. Drawing to a window 32
1.4.5.1. Graphics contexts..32
1.4.5.2. Graphics drawing API...33
1.4.6. Utility functions 34

List of Tables

1-1. Microwindows Core Modules 10
1-2. Region Routines 13
1-3. Microwindows Messaging Functions 20
1-4. Microwindows Window Registration, Creation & Destruction Functions 21
1-5. Microwindows Graphics Drawing API 24
1-6. Microwindows Utility Functions 26
1-7. Microwindows Rectangle & Region Functions 27
1-8. Microwindows Region Creation & Manipulation Functions 28
1-9. Microwindows Clip Region Functions 29
1-10. Nano-X Graphics Drawing Functions 33
1-11. Nano-X Utility Functions 35

Chapter 1. Microwindows Architecture

1.1. Architecture

This is my first cut at getting the architecture and implementation spilled out. Please let me know if there's more detail needed in some areas, or whether you're confused by my explanations. This document is for educational and porting purposes, so please read on.

1.1.1. Layered Design

Microwindows is essentially a layered design that allows different layers to be used or rewritten to suite the needs of the implementation. At the lowest level, screen, mouse/touchpad and keyboard drivers provide access to the actual display and other user-input hardware. At the mid level, a portable graphics engine is implemented, providing support for line draws, area fills, polygons, clipping and color models. At the upper level, various API's are implemented providing access to the graphics applications programmer. These APIs may or may not provide desktop and/or window look and feel. Currently, Microwindows supports the ECMA APIW and Nano-X APIs. These APIs provide close compatibility with the Win32 and X Window systems, allowing programs to be ported from other systems easily.

1.1.2. Device Drivers

The device driver interfaces are defined in `device.h`. A given implementation of Microwindows will link at least one screen, mouse and keyboard driver into the system. The mid level routines in the device-independent graphics engine core then call the device driver directly to perform the hardware-specific operations. This setup allows varying hardware devices to be added to the Microwindows system without affecting the way the entire system works.

1.1.2.1. Screen Driver

There are currently screen drivers written for Linux 2.2.x framebuffer systems, as well as 16-bit ELKS and MSDOS drivers for real-mode VGA cards. The real mode drivers (`scr_bios.c`, `vgaplan4.c`, `mempl4.c`, `scr_her.c`) can be configured to initialize the VGA hardware directly, or utilize the PC BIOS to begin and end graphics mode. The framebuffer drivers (`scr_fb.c`, `fb.c`, `fblin?.c`) have routines for 1, 2, 4 and 8bpp palletized displays, as well as 8, 15, 16, and 32 bpp truecolor displays. The framebuffer system works in Linux by opening `/dev/fd0` (or `getenv("FRAMEBUFFER")`) and mmap()ing the display memory into a linear buffer in memory. Some display modes, like the VGA 4 planes mode, require that OUT instructions be issued by the screen driver, while packed pixel drivers typically can get away with just reading and writing the framebuffer only. All the graphics mode initialization and deinitialization is handled by the Linux kernel. Getting this set up can be a real pain.

The screen driver is the most complex driver in the system, but was designed so that it can be extremely easy to port new hardware to Microwindows. For this reason, there are but a few entry points that must actually talk to the hardware, while other routines are provided that allow just the small set of core routines to be used, if desired. For example, a screen driver must implement `ReadPixel`, `DrawPixel`, `DrawHorzLine`, and `DrawVertLine`. These routines read and write a pixel from display memory, as well as draw a horizontal and vertical line. Clipping is handled at the device-independent layer. Currently, all mouse movement, text drawing, and bitmap drawing run on top of these low level functions. In the future, entry points will be provided for fast text and bitmap drawing capabilities. If the display is palletized, a `SetPalette` routine must be included, unless a static palette that matches the system palette is linked into the system. The screen driver, on initialization, returns values telling the system the x,y size of the screen, along with the color model supported.

Two font models are currently provided, to be linked in at your desire. The proportional font model has in-core font tables built from `.bdf` and other font conversion utilities provided. The rom-based font uses the PC BIOS to find the character generator table address and has routines to draw that fixed-pitch font format.

The screen driver can choose to implement bitblitting, by ORing in PSF_HAVEBLIT into the returned flags field. When present, bit blitting allows Microwindows to perform off-screen drawing. Microwindows allows any graphics operation that can be performed on a physical screen to be performed off-screen, and then copied (bit-blitted) to the physical screen. Implementing a blitting screen driver can be fairly complex. The first consideration in implementing a blitting driver is whether the low-level display hardware can be passed a hardware address for a framebuffer. If so, then the same routines that draw to the physical screen can be used to draw to off-screen buffers. This is the method used for the linear framebuffer drivers provided for Linux packed-pixel displays. The system replaces the mmap()'d physical framebuffer address with a malloc()'d memory address and calls the original screen driver entry point. In the case where the system doesn't use an actual physical memory address, like when running on top of X or MS Windows, then two sets of routines must be written; one to write the the underlying graphics system hardware, and another to write to memory addresses. In addition, the blit entry point must then know how to copy both ways between the two formats. In fact, all four operations, screen-to-memory, memory-to-screen, memory-to-memory, and screen-to-screen are supported by Microwindows and may need to be performed. And of course the bit blitting routine must be _fast_. See the files `fblin8.c` and `mempl4.c` for examples of supporting both types of display hardware.

If writing your first screen driver, I would recommend you start with the PC BIOS real mode driver, `scr_bios.c`, or take a look at the framebuffer driver, `scr_fb.c`, which is essentially a wrapper around all the `fblin?.c` routines to read and write various framebuffer formats. Don't set the PSF_HAVEBLIT flag at first, and you won't have to write a bitblit routine from the start.

Note that currently, all SCREENDEVICE function pointers must be filled in to at least a void function. For speed reasons, the system always assumes that the function pointers are valid. Thus, even if not implementing bitblit, a do-nothing bit-blit procedure must be provided.

1.1.2.2. Mouse Driver

There are three mouse drivers currently included in Microwindows. A GPM driver for Linux, `mou_gpm.c`, as well as a serial port mouse driver for Linux and ELKS, `mou_ser.c` . For MSDOS, an int33 driver `mou_dos.c` is provided. The provided mouse drivers decode MS, PC and Logitech mice formats. A mouse driver's basic function is to decode the mouse data and return either relative or absolute data for the mouse position and buttons.

In addition, Brad LaRonde has written a touch panel driver `mou_tp.c`, which masquerades as a mouse driver. It returns the value of x, y value of the pen on the display surface, and can be used like a mouse.

Under Linux, the main loop of Microwindows is a `select()` statement, with file descriptors for the mouse and keyboard driver always passed in. If the system that Microwindows is running on doesn't support `select()` or doesn't pass mouse data through a file descriptor, a `Poll()` entry point is provided.

1.1.2.3. Keyboard Driver

There are two keyboard drivers provided. The first, `kbd_tty.c`, is used for Linux and ELKS systems where the keyboard is opened and read as through a file descriptor. The second, `kbd_bios.c`, read the PC BIOS for keystrokes and is used in MSDOS real mode. The keyboard driver currently returns 8-bit data from the keyboard, but doesn't decode multi-character function key codes. This functionality will need to be added soon, by reading termcap files or the like.

1.1.3. MicroGUI - Device Independent Graphics Engine

The core graphics functionality of Microwindows resides in the device independent graphics engine, which calls the screen, mouse and keyboard drivers to interface with the hardware. User applications programs never all the core graphics engine routines directly, but rather through the programmer API's, discussed in the next sections. The core engine routines are separated from the applications API's is for a variety of reasons. The core routines will always reside on the server in a client/server environment. Also, the core routines use internal text font and bitmap formats that are designed for speed and may or may not be the same as the structures used in standard

API's. In addition, the core routines always use pointers, never ID's, and can then be used together to implement more complex functions without always converting handles, etc.

In Microwindows, the core routines all begin as `GdXXX()` functions, and are concerned with graphics output, not window management. In addition, all clipping and color conversion is handled within this layer. The following files comprise the core modules of Microwindows:

Table 1-1. Microwindows Core Modules

File	Description
devdraw.c	Core graphics routines for line, circle, polygon draw and fill, text and bitmap drawing, color conversion
devclip.c	Core clipping routines. (devclip2.c is the new y-x-banding algorithm, devclip1.c an older method)
devrgn.c	New dynamically allocated routines for intersect/union/subtract/xor region creation.
devmouse.c	Core routines for keeping the mouse pointer updated or clipped from the screen.
devkbd.c	Core keyboard handling routines.
devpalX.c	Linked in static palettes for 1, 2, 4 and 8bpp palletized systems.

The MicroGUI graphics engine routines are discussed in detail in Section 1.2.

1.1.4. Applications Programmer Interfaces

Microwindows currently supports two different application programming interfaces. This set of routines handles client/server activity, window manager activities like drawing title bars, close boxes, etc, as well as, of course, handling the programmer's requests for graphics output. Both the API's run on top of the core graphics engine routines and device drivers.

The basic model of any API on top of Microwindows is to hang in initialize the screen, keyboard and mouse drivers, then hang in a `select()` loop waiting for an event. When an event occurs, if it's a system event like keyboard or mouse activity, then this information is passed to the user program converted to an expose event, paint message, etc. If it's a user requesting a graphics operation, then the parameters are decoded and passed to the appropriate `GdXXX` engine routine. *Note* that the concept of a window versus raw graphics operations are handled at this API level. That is, the API defines the concepts of what a window is, what the coordinate systems are, etc, and then the coordinates are all converted to "screen coordinates" and passed to the core GdXXX engine routines to do the real work. This level also defines graphics or display contexts and passes that information, including clipping information, to the core engine routines.

Currently, the Microwindows API code is in `mwin/win*.c`, while the Nano-X API code is in `nanox/srv*.c`.

1.1.4.1. Microwindows API

The Microwindows API tries to be compliant with the Microsoft Win32 and WinCE GDI standard. Currently, there is support for most of the graphics drawing and clipping routines, as well as automatic window title bar drawing and dragging windows for movement. The Microwindows API is message-based, and allows programs to be written without regard to the eventual window management policies implemented by the system. The Microwindows API is not currently client/server, and will be discussed in more detail in Section 1.4.

1.1.4.2. Nano-X API

The Nano-X API is modeled after the mini-x server written initially by David Bell, which was a reimplementation of X on the MINIX operating system. It loosely follows the X Window System Xlib API, but the names all being with `GrXXX()` rather than

`X...()`. Currently, the Nano-X API is client/server, but does not have any provisions for automatic window dressings, title bars, or user window moves. There are several groups writing widget sets currently, which will provide such things. Unfortunately, the user programs must also then write only to a specific widget set API, rather than using the Nano-X API directly, which means that only the functionality provided by the widget set will be upwardly available to the applications programmer. (Although this could be considerable, in the case that, say Gdk was ported.)

1.2. Device-Independent Engine Features

This section discusses in the capabilities and implementation of Microwindows' core graphics engine in detail. It's purpose is both educational and to allow extending an API by understanding how the engine works.

1.2.1. Graphics Engine Features and Implementation

These routines concern themselves with drawing operations to off-screen or screen surfaces. Each routine starts with Gd... and is passed a pointer to the SCREENDEVICE structure (PSD) as it's first parameter. The PSD parameter specifies the low level display details, like the x, y size of the device and the color model used by the hardware, for example. In addition, the actual routines to perform drawing are function pointers in this structure. All screen coordinates are of type COORD, and specified in device coordinates, that is, offsets from the upper left corner of the screen.

Colors are always specified as an RGB COLORVAL value into the graphics engine. They are then possibly converted to palette indices and sent to the display hardware as PIXELVAL values. In the case of 32bpp truecolor displays, no conversion is required. The color model will be discussed in detail below.

1.2.1.1. Regions

Regions are used to describe arbitrary sets of pixels on the screen. A simple, square region of pixels can be described by a single rectangle. More complex sets of pixels require more complex data structures. In Microwindows, regions are described by an array of non-overlapping rectangles. Currently, there are two different implementations of regions in Microwindows, the original design for systems with limited memory, and a new design with dynamically allocated sets of rectangles. The original design used a single static array of MWCLIPRECTs to describe complex regions. Any point within any rectangle in the array was considered to be in the region. The array wasn't sorted in any particular order, but was always guaranteed to contain non-overlapping rectangles. Another global variable, clipcount, specified the number of rectangles in the array. This original design had no engine entry points for region management, the entire array was passed to the clipping functions, described below.

In the new design set with #define DYNAMICREGIONS 1, any number of regions can be created, as the regions (MWCLIPREGION *) are stored as dynamically allocated arrays of rectangles. In this implementation, the non-overlapping rectangles are always kept in "y-x" sorted bands, such that each band's y height is the same for all rectangles in the band. This means that only the x position and width of the rectangles in each band varied. Because of this, it is easier to create a set of functions for combining regions, since effectively only a single dimension had to be compared for each region operation. The new region handling routines allow for creating and destroying regions, as well as combining rectangles and regions with regions using Intersection, Union, Subtraction, and Exclusive Or. This model allows regions to be implemented apart from the clipping routines, unlike the first version. Following are the new region routines:

Table 1-2. Region Routines

Function	Description
GdAllocRegion	Create a region.
GdAllocRectRegion	Create a rectangular region from left,top,right,bottom.

Function	Description
GdAllocRectRegionIndirect	Create a rectanglular region from a MWRECT.
GdSetRectRegion	Set a region to a single rectangle.
GdDestroyRegion	Destroy a region.
GdCopyRegion	Copy a region.
GdUnionRectWithRegion	Union a rectangle with a region.
GdIntersectRegion	Create a region from the intersection of two regions.
GdSubtractRegion	Create a region from the difference of two regions.
GdUnionRegion	Create a region from the union of two regions.
GdXorRegion	Create a region from the XOR of two regions.

1.2.1.2. Clipping

Clipping in Microwindows is closely tied to the region management code. At any point in time, the graphics engine has a single clipping region, that is a set of rectangles, defined for any graphics operation. A point is drawn if it is "inside" any of the current set of clip rectangles. Two slightly modified versions of the clipping algorithm are supplied, `devclip1.c` for the original, static rectangle array, and `devclip2.c` for the newer dynamically allocated array. A single entry point GdSetClipRects, takes the passed region and specifies it's use for all subsequent graphics operations. All the drawing routines then use the two additional routines to determine whether or not to draw. `GdClipPoint` takes an x,y point in screen coordinates and returns TRUE if the point can be drawn, that is, the point is within one of the region rectangles. `GdClipArea` takes an upper left and lower right point, and returns one of the following: CLIP_VISIBLE, if the specified area is completely within the region, CLIP_INVISIBLE, if the area is completely not in the region, which means that no drawing should be performed, or CLIP_PARTIAL, if a part but not the whole area is within the region. A typical graphics primitive will call the screen driver with unmodified passed inputs if CLIP_VISIBLE is returned, or return if CLIP_INIVISIBLE is returned. In the CLIP_PARTIAL case, the primitive must break up the original request into areas that are within the clip region before calling the screen driver. This slows down the operation considerably.

Because the clipping code is called constantly before drawing operations, Microwindows keeps a global cache rectangle of the last rectangle checked with `GdClipArea`, for speed and also to allow the mid level to quickly calculate how partial drawing lengths.

1.2.1.3. Line Drawing

Line drawing is the simplest of graphics operations. Microwindows supports `GdPoint` to draw a point, and `GdLine` to draw a horizontal, vertical or diagonal (using Bresenham algorithm) line. Just before any call to the screen driver, a call to `GdCheckCursor` assures that the software cursor is removed prior to drawing. `GdFixCursor` restores the cursor if previously visible.

There is a tricky part to line drawing that had to be added during the support for multiple API's. This has to do with whether or not the last point in specified line segment is drawn or not. There are two schools of thought on this, and to make it short, Microwindows supports both of them. The last parameter to GdLine specifies whether or not to draw the last point. The Microwindows API doesn't draw the last point, but the Nano-X API does.

Most drawing functions, including line drawing draw using the "current" foreground color, specified using `GdSetForeground`. In addition a drawing mode, currently either MODE_SET or MODE_XOR can be specified using `GdSetMode`.

1.2.1.4. Rectangles, Circles, Ellipses

Rectangles, circles and ellipses are drawn using the `GdRect` and `GdEllipse` routines.

A circle is an ellipse with the same x and y radius. As with lines, rectangles and ellipses are drawn using the current foreground color and mode.

1.2.1.5. Polygons

Microwindows supports polygon drawing by specifying an array of x,y points. The points are then connected using the `GdLine` function. The current foreground color, drawing mode, and clip region is used during output.

1.2.1.6. Area Fills

Microwindows supports filled rectangular areas using the `GdFillRect` function. The rectangle's outline and contents are filled using the current foreground color. Filled circles and ellipses are performed with `GdFillEllipse`, and polygon fills with `GdFillPoly`. Area filling is implemented through successive calls to the `DrawHorzLine` in the screen driver, and are much faster if fully not clipped.

1.2.1.7. Fonts

Both fixed pitch and proportional fonts are supported in Microwindows. Because of potentially large differences in display hardware, the actual font format is known only to the screen driver, although a set of standard functions are supplied for dealing with converted `.bdf` fonts and Microsoft Windows fonts, if you have a license. The engine function `GdSetFont` specifies a font number that is passed to the driver and used to index a static array of linked in fonts. Screen driver entry points GetTextSize return the font height and width for a passed string, and `GetTextBits` returns an individual character bitmap. The engine layer uses these values to calculate a clipping region for text drawing, as well as to draw the character as a monochrome bitmap.

The screen drivers currently supplied implement both fixed pitch PC ROM based fonts, as well as a proportional font format that is linked into the screen driver. A few conversion programs allow conversion of fonts from different formats to the driver format. `Bdftobogl` converts X Window System `.bdf` files to Microwindows format.

`Convfnt32` converts raster and truetype Microsoft Windows fonts, if you have a license, to Microwindows format. Convrom converts PC ROM bios fonts.

A number of free fonts are supplied with the system, a heavier proportional 14x16 system font, and a sans-serif 11x13 font for title bar and edit box displays. Any number of fonts can be linked into the system, and it's fairly easy to dynamically load fonts if one writes the routines for it.

1.2.1.8. Text Drawing

Text output is performed by first selecting the desired font with `GdSetFont`, and then calling the `GdText` function. Full text clipping is performed, although currently there is no "fast" text output entry point in the screen driver, so each character bitmap is grabbed using the `GetTextBits` entrypoint and then drawn using `Drawpixel`. While this will have to remain the same for partially clipped text, a screen driver entry point to draw fast text will probably be required shortly.

Text is drawn using the current foreground color. The background is drawn if the current "use background" mode set via `GdUseBackground` is TRUE. In this case the background is drawn using the current background color set via `GdSetBackground`. The `GdText` function also takes a bottomAlign parameter that specifies whether the text is to be bottom or top aligned, to help with differing API's.

1.2.1.9. Color model and palettes

The Microwindows graphics engine requires all colors to be specified as either 24-bit RGB color values, or in rare cases, as palette indices for speed. The palette index method will only work on systems that have hardware palettes, so it's not recommended. All of the upper-level color parameters are specified to the engine routines using a COLORVAL value, which is a long containing the desired RGB color, created using the `RGB()` macro. The engine then converts the COLORVAL to a PIXELVAL value, which is normally a long also, but on some smaller systems can be compiled as an unsigned char. The PIXELVAL value is the actual value passed to any screen driver entry point requiring a color. So the mid level routines all work with RGB

COLORVALs, while the device driver routines all work with PIXELVALs. The graphics engine converts these values using two routines, `GdFindColor` and `GdFindNearestColor`, described below.

`GdFindColor` takes a hardware independent RGB value and converts it to a hardware dependent PIXELVAL pixel value. In the case of 32bpp display drivers, no conversion is required. Otherwise for truecolor systems, Microwindows converts the RGB value to a 5/5/5 15-bit or 5/6/5 16 bit truecolor value. For 8bpp truecolor displays, the RGB value is converted to 3/3/2. For palletized displays, the `GdFindNearestColor` function is called to convert the RGB color to the nearest palette index in the current system palette. `GdFindNearestColor` uses a weighted distance-cubed method to find the palette value nearest to the requested color, and returns it. Standard palettes for 1, 2, 4 and 8bpp are included in the files `devpal1.c`, `devpal2.c`, `devpal4.c` and `devpal8.c`. These palettes associate an RGB value with an index, but may be overwritten.

The `GdSetPalette` function determines whether there are any free entries in the system palette (discussed shortly) and if so, adds entries to the system palette, and calls the screen driver `SetPalette` entry point to set the hardware palette. There is a single global variable, gr_firstuserpalentry, that contains the index of the next available system palette entry. Initially, this is set to 24. Thus, systems with less than 24 total palette entries will never have an available palette entry to remap. On systems that do, like 256 color systems, then images requiring more color entries keep calling `GdSetPalette` until the system palette is full. To reset marker, the function `GdResetPalette` is called. This allows upper level API's to distinguish between different images and force the system palette to be rewritten.

1.2.1.10. Image Drawing

Microwindows supports two styles of images, monochrome and palettized. Monochrome images are specified with an IMAGEBITS structure, which is an array of words with 1 bits specifying the foreground color and 0 bits the background. The IMAGEBITS bits are short-word padded to the width of the bitmap. The `GdBitmap` routine draws monochrome bitmaps, similar to `GdText`, by drawing all the 1 bits in the foreground color, and the 0 bits in the background color if the "use background" set by `GdUseBackground` is TRUE.

Color bitmaps are specified using a 1, 4 or 8bpp image palette, and an array of indices into this palette, all stuffed into an IMAGEHDR structure, and drawn via `GdDrawImage`. First, the system creates a conversion palette by calling `GdMakePaletteConversionTable`, which converts the images' palette entries into system indices. At the same time, the system attempts to increase the system palette if necessary by calling the `GdSetPalette` function described above. At the end of this operation, the image has a converted palette which necessarily corresponds to the system palette. In the case of truecolor hardware, the image's palette entries are converted to hardware truecolor pixel values, and output directly.

After converting the image color entries the `GdDrawImage` determines whether the image is clipped, and outputs the image, pixel by pixel. In the future, a blitting routine could be used for faster image drawing.

1.2.1.11. Blitting

Blitting functionality is required in the screen driver for offscreen drawing capability, discussed earlier in the screen drivers section. The engine function `GdBlit` allows upper level APIs to implement copy operations from offscreen memory to the display, or vice versa. The blit format is driver specific, and generally only works for memory images created by the screen driver during runtime. The upper level APIs implement this by allocating a new SCREENDRIVER structure, copying an existing SCREENDRIVER structure into it, replacing the address field with a `malloc()`'d value, and setting the PSF_MEMORY bit, which indicates to the display driver that this is now an offscreen surface. Any subsequent calls to the engine routines then draw onto this surface. When it is desired to copy the offscreen surface back to the physical display, the GdBlit routine is called. Currently, only SRCCOPY operations are performed, but future plans will add blitting opcodes.

The function `GdCalcMemGCAlloc` calculates the byte size and line length (pitch) of an offscreen memory area given the passed bpp and planes parameters. This is before calling the screen driver to allocate an offscreen screen device.

1.3. Microwindows API

1.3.1. Message-passing architecture

The fundamental communications mechanism in the Microwindows API is the message. A message consists of a well-known message number, and two parameters, known as wParam and lParam. Messages are stored in an application's message-queue, and retrieved via the `GetMessage` function. The application blocks while waiting for a message. There are messages that correspond to hardware events, like WM_CHAR for keyboard input or WM_LBUTTONDOWN for mouse button down. In addtiion, events signaling window creation and destruction WM_CREATE and WM_DESTROY are sent. In most cases, a message is associated with a window, identified as an HWND. After retrieving the message, the application sends the message to the associated window's handling procedure using `DispatchMessage`. When a window class is created, it's associated message handling procedure is specified, so the system knows where to send the message.

The message-passing architecture allows the core API to manage many system functions by sending messages on all sorts of events, like window creation, painting needed, moving, etc. By default, the associated window handling function gets a "first pass" at the message, and then calls the `DefWindowProc` function, which handles default actions for all the messages. In this way, all windows can behave the same way when dragged, etc, unless specifically overridden by the user. Major window management policies can be redefined by merely re-implementing `DefWindowProc`, rather than making changes throughout the system.

The following functions deal with messages directly:

Table 1-3. Microwindows Messaging Functions

Function	Description
SendMessage	Send a message directly to a window.

Function	Description
PostMessage	Queue a message on the application's message queue for later dispatch.
PostQuitMessage	Queue a WM_QUIT message telling the application to terminate when read.
GetMessage	Block until a message is queued for this application.
TranslateMessage	Translate up/down keystrokes to WM_CHAR messages.
DispatchMessage	Send a messages to it's associated window procedure.

A Microwindows application's entry point is the function `WinMain`, rather than main.

1.3.2. Window creation and destruction

The basic unit of screen organization in Microwindows API is the window. Windows describe an area of the screen to draw onto, as well as an associate "window procedure" for handling messages destined for this window. Applications programmers can create windows from pre-defined classes, like buttons, edit boxes, and the like, or define their own window classes. In both cases, the method of creating and communicating with the windows remains exactly the same. The following functions deal with window registration, creation, and destruction:

Table 1-4. Microwindows Window Registration, Creation & Destruction Functions

Function	Description
RegisterClass	Define a new window class name and associated window procedure.
UnRegisterClass	Undefine a window class.

Function	Description
CreateWindowEx	Create an instance of a window of a certain class.
DestroyWindow	Destroy a window instance.
GetWindowLong	Return information about a window.
SetWindowLong	Set information about a window.
GetWindowWord	Return user information about a window.
SetWindowWord	Set user information about a window.
GetClassLong	Return information about a window class.
GetWindowText	Get a window's title or text.
SetWindowText	Set a window's title or text.

The WM_CREATE message is just after window creation, before returning from `CreateWindowEx`. The WM_DESTROY message is sent just before destroying a window with `DestroyWindow`.

When a window is registered, extra bytes can be allocated to the window structure when created. The `GetWindowLong`, `GetWindowWord`, `SetWindowLong` and `SetWindowWord` manipulate these bytes. In addition, a fixed number of extra bytes per window class can be allocated on registration and retrieved via the `GetClassLong` function.

1.3.3. Window showing, hiding and moving

The ShowWindow function allows windows to be made visible or hidden. In addition, this can be specified during the creation of the window, through `CreateWindowEx`. `MoveWindow` is called to change a window's position or size. A WM_MOVE message is sent if the window's position changes, and WM_SIZE is sent on size changes.

1.3.4. Window painting

The Microwindows system determines when a window needs to be initially painted or repainted as the result of other window movement, and a WM_PAINT message is sent to the associated window procedure. At this point, it's up the the application to use the graphics primitives available to paint the window, described below. Microwindows keeps track of a windows' "update" region, and sends WM_PAINT whenever the region is non-empty. For speed reasons, the WM_PAINT message is only sent when there are no other messages in the application's queue. This allows the application to repaint in one, rather than possibly many, steps. To force a repaint rather than waiting, the UpdateWindow function can be called. The `InvalidateRect` function specifies a rectangle to add to the update region, causing a subsequent WM_PAINT.

The window title is automatically painted and is set with the `SetWindowText` function, and retrieved with the `GetWindowText` function.

1.3.4.1. Client and screen coordinates

Every window is drawn on the screen using the device global screen coordinate system for absolute reference to any pixel on the screen. The Microwindows API allows applications programmers to be concerned with only the relative coordinates from the upper left portion of their window, not including the title bar and 3d effects. This coordinate system is called *"client coordinates."* As will be explained below, the Microwindows programmer has the option of getting a device context in either screen or client coordinates. If device coordinates are specified, then the coordinate system is device-based and includes the title area and 3d areas of the window. Otherwise, the drawable region is clipped to just that area that is reserved by the system for the application's drawing. The `GetClientRect` and `GetWindowRect` functions return client or screen coordinates for the passed window. `ClientToScreen` and `ScreenToClient` can be called to translate between window coordinate systems.

1.3.4.2. Device contexts

An applications programmer must obtain a *"device context"* before calling any graphics drawing API functions. As explained above, this specifies to the system which window and what coordinate system are desired, so that these don't have to be passed to every graphics function. In addition, various other attributes like foreground and background color are also set in a device context, so that these parameters don't have to be specified for every graphics operation. The device context selects the appropriate clipping region based on the window specified and the coordinate system. When a device context is obtained, various graphics values are set to default values.

To obtain a client device context, call `GetDC`. To obtain a screen device context, required when drawing onto title bars and the like, call `GetWindowDC`. In addition, fancy clipping operations and child/sibling window clipping can be specified if `GetDCEx` is called. When finished drawing, the `ReleaseDC` function must be called to deallocate the DC.

On receipt of the WM_PAINT message, two special calls, `BeginPaint` and `EndPaint` are called, that serve as replacements to the `GetDC`/`ReleaseDC` functions. These functions essentially allocate a DC but also validate the update region so that no subsequent WM_PAINT messages are generated. `BeginPaint` also combines the update region and the clipping region so that user output will only occur where previously invalidated.

1.3.4.3. Graphics Drawing Functions

There are many graphics drawing API's in the Microwindows API. Following is a list, most of these match up to the engine GdXXX functions discussed in Section 1.2.

Table 1-5. Microwindows Graphics Drawing API

Function	Description
SetTextColor	Set the foreground text color in a DC.
SetBkColor	Set the background color in a DC.
GetSysColor	Get the system color defined for the current look and feel scheme.

Function	Description
SetSysColor	Set a system color.
SetBkMode	Set the use background flag in a DC.
SetROP2	Set the drawing mode (XOR, SET, etc) for drawing.
SetPixel	Draw a pixel in the current fg color.
MoveToEx	Prepare to draw a line.
LineTo	Draw a line from the last location to this one in the current fg color.
Rectangle	Draw a rectangle in the current pen color.
FillRect	Fill a rectangle with the current brush color.
TextOut	Draw text in the current fg/bg color.
ExtTextOut	Draw text in the current fg/bg color.
DrawText	Draw text or compute text height and width sizes.
DrawDIB	Draw a color bitmap.
SelectObject	Select a pen, brush or font to use in a DC.
GetStockObject	Get a predefined standard pen, brush or font.
CreatePen	Create a pen of a certain color.
CreateSolidBrush	Create a brush of a certain color.
CreateCompatibleBitmap	Create an offscreen area to draw onto.
DeleteObject	Delete a pen, brush or bitmap.
CreateCompatibleDC	Create an offscreen DC.
DeleteDC	Delete an offscreen DC.
BitBlit	Copy from one bitmap in a DC to another.

Function	Description
GetSystemPaletteEntries	Get the currently in-use system palette entries.

1.3.5. Utility functions

A number of routines are provided for various purposes, described below. In addition, Microwindows currently exports some helper routines, named WndXXX, that are useful but subject to change. These are detailed following:

Table 1-6. Microwindows Utility Functions

Function	Description
WndSetDesktopWallpaper	Set the desktop background image.
WndSetCursor	Set the cursor for a window.
WndRaiseWindow	Raise a window’s z-order.
WndLowerWindow	Lower a window’s z-order.
WndGetTopWindow	Return the topmost window’s handle.
WndRegisterFdInput	Register to send a message when file descriptor has read data available.
WndUnregisterFdInput	Unregister file descriptor for read data messages.
GetTickCount	Return # milliseconds elapsed since startup.
Sleep	Delay processing for specified milliseconds.
SetTimer	Create a millisecond timer.
KillTimer	Destroy a millsecond timer.

Function	**Description**
GetCursorPos	Return mouse cursor coordinates.

1.3.5.1. Setting window focus

The `SetFocus` routine is used to pass keyboard focus from one window to another. Keystrokes are always sent to the window with focus. The WM_SETFOCUS and WM_KILLFOCUS messages are sent to windows just receiving and losing focus. The `GetActiveWindow` routines returns the first non-child ancestor of the focus window, which is the window that is currently highlighted. The `GetDesktopWindow` routine returns the window handle of the desktop window.

1.3.5.2. Mouse capture

Normally, Microwindows sends WM_MOUSEMOVE messages to the window the mouse is currently moving over. If desired, the applications programmer can "capture" the mouse and receive all mouse move messages by calling `SetCapture`. `ReleaseCapture` returns the processing to normal. In addition, the `GetCapture` function will return the window with capture, if any.

1.3.5.3. Rectangle and Region management

There are a number of functions that are used for rectangles and regions. Following is the group:

Table 1-7. Microwindows Rectangle & Region Functions

Function	**Description**
SetRect	Define a rectangle structure.
SetRectEmpty	Define an empty rectangle.
CopyRect	Copy a rectangle.

Function	Description
IsRectEmpty	Return TRUE if empty rectangle.
InflateRect	Enlarge a rectangle.
OffsetRect	Move a rectangle.
PtInRect	Determine if a point is in a rectangle.
PtInRect	Determine if a point is in a rectangle.
IntersectRect	Intersect two rectangles.
UnionRect	Union two rectangles.
SubtractRect	Difference two rectangles.
EqualRect	Determine if two rectangles are the same.

The following functions are used for region creation and manipulation:

Table 1-8. Microwindows Region Creation & Manipulation Functions

Function	Description
CreateRectRgn	Create a rectangular region.
CreateRectRgnIndirect	Create a rectangular region from a RECT.
SetRectRgn	Set a region to a single rectangle.
CreateRoundRectRgn	Create a round rectangular region.
CreateEllipticRgn	Create an elliptical or circular region.
CreateEllipticRgnIndirect	Create an elliptical or circular region from a RECT.
OffsetRgn	Offset a region by x, y values.
GetRgnBox	Get a region’s bounding rect.
GetRegionData	Get a region’s internal data structure.
PtInRgn	Determine if a point is in a region.
RectInRegion	Determine if a rectangle intersects a region.

Function	Description
EqualRgn	Determine if two regions are equal.
CombineRgn	Copy/And/Or/Xor/Subtract a region from another.

The following regions are used to set user specified clipping regions. These regions are then intersected with the visible clipping region that Microwindows maintains prior to drawing:

Table 1-9. Microwindows Clip Region Functions

Function	Description
SelectClipRegion	Assign a user specified clipping region.
ExtSelectClipRegion	And/Or/Xor/Subtract user clipping region with another region.

1.4. Nano-X API

The Nano-X API was originally designed by David Bell, with his mini-x package for the MINIX operating system. Nano-X is now running on top of the core graphics engine routines discussed in Section 1.2. Nano-X was designed for a client/server environment, as no pointers to structures are passed to the API routines, instead a call is made to the server to get an ID, which is passed to the API functions and is used to reference the data on the server. In addition, Nano-X is not message-oriented, instead modeled after the X protocol which was designed for speed on systems where the client and server machines were different.

1.4.1. Client/Server model

In Nano-X, there are two linking mechanisms that can be used for applications programs. In the client/server model, the application program is linked with a client library that forms a UNIX socket connection with the Nano-X server, a separate process. Each application then communicates all parameters over the UNIX socket. For speed and debugging, it is sometimes desirable to link the application directly with the server. In this case, a stub library is provided that just passes the client routines parameters to the server function.

The Nano-X naming convention uses GrXXX to designate client side callable routines, with a marshalling layer implemented in the files `nanox/client.c` , `nanox/nxproto.c` , and `nanox/srvnet.c` . The client/server network layer currently uses a fast approach to marshalling the data from the Gr routine into a buffer, and sent all at once to the receiving stubs in `nanox/srvnet.c`, before calling the server drawing routines in `nanox/srvfunc.c`. In the linked application scenario, the Nano-X client links directly with the functions in `nanox/srvfunc.c` , and the `nanox/client.c` and `nanox/srvnet.c` files are not required.

A Nano-X application must call `GrOpen` before calling any other Nano-X function, and call `GrClose` before exiting. These functions establish a connection with the server when running the client/server model, and return an error status if the server can't be found or isn't currently running.

The main loop in a Nano-X application is to create some windows, define the events you want with `GrSelectEvents`, and then wait for an event with `GrGetNextEvent`. If it is desired to merely check for an event, but not wait if there isn't one, `GrCheckNextEvent` can be used. `GrPeekEvent` can be used to examine the next event without removing it from the queue.

When running Nano-X programs in the client/server model, it's currently necessary to run the server first in a shell script, then wait a second, then run the application. Some rewriting is needed to fire up the server when an application requires it, I believe.

1.4.2. Events

Nano-X applications specify which events they would like to see on a per-window basis using `GrSelectEvents`. Then, in the main loop, the application calls `GrGetNextEvent` and waits for one of the event types selected for in any of the windows. Typically, a switch statement is used to determine what to do after receiving the event. This is similar to the Microwindows's API `GetMessage`/`DispatchMessage` loop, except that in Microwindows API, `DispatchMessage` is used to send the event to the window's handling procedure, typically located with the window object. In Nano-X, all the event handling code for each of the windows must be placed together in the main event loop, there is no automatic dispatching. Of course, widget sets serve to provide object-orientation, but this is in addition to the Nano-X API.

Following are the event types that Nano-X programs can recieve:

GR_EVENT_TYPE_NONE, ERROR, EXPOSURE, BUTTON_DOWN, BUTTON_UP, MOUSE_ENTER, MOUSE_EXIT, MOUSE_MOTION, MOUSE_POSITION, KEY_UP, KEY_DOWN, FOCUS_IN, FOCUS_OUT, FDINPUT, UPDATE, CHLD_UPDATE

Note that Nano-X API provides mouse enter and exit events whereas Microwindows API does not. Also, the exposure events are calculated and sent immediately by the server, and not combined and possibly delayed for better paint throughput as in the Microwindows API.

1.4.3. Window creation and destruction

Windows are created in Nano-X with the `GrNewWindow` function. Windows can be specified to be input-only, in which case the `GrNewInputWindow` function is used. The window border and color is specified in these calls, but will have to be rewritten when fancier window dressings are required. The return value from these functions is an ID that can be used in later calls to get a graphics context or perform window manipulation.

Pixmaps, which are offscreen windows, are created with GrNewPixmap. The ID returned can be used with any drawing function. Pixmaps are copied to windows using the GrCopyArea function, and destroyed like windows with GrDestroyWindow.

1.4.4. Window showing, hiding and moving

Windows are shown by calling the `GrMapWindow` function, and hidden using `GrUnmapWindow`. Mapping a window is required for all ancestors of a window in order for it to be visible. The GrRaiseWindow call is used to raise the Z order of a window, while `GrLowerWindow` is used to lower the Z order. `GrMoveWindow` is used to change the position of a window, and `GrResizeWindow` is used to resize a window. A window can be reparented with `GrReparentWindow`.

1.4.5. Drawing to a window

Nano-X requires both a window ID and a graphics context ID in order to draw to a window. Nano-X sends expose events to the application when a window needs to be redrawn. Unlike the Microwindows API, Nano-X clients are typically required to create their drawing graphics contexts early on and keep them for the duration of the application. Like Microwindows though, the graphics contexts record information like the current background and foreground colors so they don't have to be specified in every graphics API call.

1.4.5.1. Graphics contexts

To allocate a graphics context for a window, call `GrNewGC`. On termination, call `GrDestroyGC`. `GrCopyGC` can be used to copy on GC to another. `GrGetGCInfo` is used to retrieve the settings contained in a GC. After creating a graphics context, the server returns a graphics context ID. This is then used as a parameter in all the graphics drawing API functions. In Nano-X programs, the current clipping region and window coordinate system aren't stored with the GC, as they are in Microwindows' DCs. This is because, first, Nano-X doesn't support dual coordinate systems for drawing to the "window dressing" area versus the "user" area of the window (window and client coordinates in Microwindows). User programs can't draw the border area of the window, only a single color and width can be specified. Although resembling X, this will have to change, so that widget sets can specify the look and feel of all aspects of the windows they maintain. Since the clipping region isn't maintained with the graphics context, but instead with the window data structure, Nano-X applications must specify both a window ID and a graphics context ID when calling any graphics API function. Because of this, many Nano-X applications allocate all graphics contexts in the beginning of the program, and hold them throughout execution, since the graphics contexts hold only things like foreground color, etc, and no window information. This cannot be done with Microwindows API because the DC's contain window clipping information and must be released before processing the next message.

1.4.5.2. Graphics drawing API

Following are the graphics drawing functions available with Nano-X. Like Microwindows API, these all match up eventually to the graphics engine `GdXXX` routines.

Table 1-10. Nano-X Graphics Drawing Functions

Function	Description
GrGetGCTextSize	Return text width and height information.
GrClearWindow	Clear a window to it's background color.
GrSetGCForeground	Set the foreground color in a graphics context.
GrSetGCBackground	Set the background color in a graphics context.
GrSetGCUseBackground	Set the "use background color" in a graphics context.
GrSetGCMode	Set the drawing mode.
GrSetGCFont	Set the font.

Function	Description
GrPoint	Draw a point in the passed gc's foreground color.
GrLine	Draw a line in the passed gc's foreground color.
GrRect	Draw a rectangle in passed gc's foreground color.
GrFillRect	Fill a rectangle with the passed gc's foreground color.
GrEllipse	Draw a circle or ellipse with the passed gc's foreground color.
GrFillEllipse	Fill a circle or ellipse with the passed gc's foreground color.
GrPoly	Draw a polygon using the passed gc's foreground color.
GrFillPoly	Fill a polygon using the passed gc's foreground color.
GrText	Draw a text string using the foreground and possibly background colors.
GrBitmap	Draw an image using a passed monocrhome bitmap, use fb/bg colors.
GrBMP	Draw an image from a .bmp file.
GrJPEG	Draw an image from a .jpg file.
GrArea	Draw a rectangular area using the passed device-dependent pixels.
GrReadArea	Read the pixel values from the screen and return them.
GrGetSystemPaletteEntries	Get the currently in-use system palette entries.

Function	Description
GrFindColor	Translate an RGB color value to a PIXELVAL pixel value.

1.4.6. Utility functions

Various functions serve as utility functions to manipulate windows and provide other information. These include the following:

Table 1-11. Nano-X Utility Functions

Function	Description
GrSetBorderColor	Set the border color of a window. Not suitable for 3d look and feel.
GrSetCursor	Set the cursor bitmap for the window.
GrMoveCursor	Move the cursor to absolute screen coordinates.
GrSetFocus	Set the keyboard input focus window.
GrRedrawScreen	Redraw the entire screen.
GrGetScreenInfo	Return information about the size of the physical display.
GrGetWindowInfo	Return information about the passed window.
GrGetGCInfo	Return information about the passed graphics context.
GrGetFontInfo	Return information about the passed font number.
GrRegisterInput	Register a file descriptor to return an event when read data available.

Function	Description
GrPrepareSelect	Prepare the fd_set and maxfd variables for using Nano-X as a passive library.
GrServiceSelect	Callback the passed GetNextEvent routine when Nano-X has events requiring processing.
GrMainLoop	A convenience routine for a typical Nano-X application main loop.