CMU-ITC-88-069

The ITC Window Manager *Programmers Manual*

James Gosling

This document is a guide for programmers to the facilities of the ITC window manager. It describes all that one needs to know to be able to write programs that produce graphical images in windows.

A *window* used by a client of the graphics package is a rectangular patch of pixels. The upper left pixel is at coordinate (0,0) and coordinates increase down and to the right. The width and height of the window may be interrogated by calling wm_GetDimensions. Performing operations at coordinates outside of the window area causes the operation to be properly clipped. One unit in the horizontal or vertical direction corresponds to one pixel. There is a *current* position which moves around as operations are performed.

This extremely simple coordinate system was explicitly chosen in preference to a more general one with full homogenous transformations. Many client programs are insensitive to which coordinate system is chosen: any one is good as any other; others have such specialized requirements that they need to do their own coordinate transformations themselves, anyway.

When the dimensions or real position of a window are changed by the user the window manager will request the client program to redraw. This appears to the client as an asyncronous call on a procedure called *FlagRedraw.* This procedure should only set a flag indicating that a redraw needs to be done, it should not actually do the redraw. This is because it is called asynchronously via the signal mechanism. Any system calls that were in progress at the time of the flagging will be aborted by the system and will return an error with errno==EINTR.

The window manager provides no other support for handling window resizing other than the notification described. This is because only the client program can know the *right* way for its image to respond.

To compile a C program, the invocation of cc that compiles a client of the window manager should have the switch "-I/usr/local/include" and the program should contain the line "#include <usergraphics.h>". When linking the program, include "-litc" in the list of libraries.

struct wm_window *wm_NewWindow (*host*)
Creates a window on the desired host. If *host* is zero and the environment variable 'WMHOST' is set then the window will be created on host WMHOST. Otherwise the window will be created on the current host. A pointer to the window is returned and that window is selected as the current one. If a window cannot be created, a null pointer will be returned. The window will be automatically be assigned space on the screen using magic: the user will not be queried.

wm_SelectWindow (struct wm_window *w)
Makes *w* be the current user window. All subsequent operations are performed on the current window. Normally, in a client program that creates

only one window no window selection need be done since *wm_NewWindow* does it automatically.

wm_MoveTo (x, y)
Move the current position to the point (x,y).

wm_DrawTo (x, y)
Draws a line from the current position to the point (x,y) and leaves the current position there.

wm_SetFunction (f)
Sets the **rasterop** combination function to *f*. This function is used whenever a graphics operation is performed. For example,

```
wm_SetFunction(f_black);
wm_MoveTo(0, 0);
wm_DrawTo(100, 100);
```

will draw a black line. The following permit functions to be expressed as Boolean combinations of the three primitive functions 'source', 'mask', and 'dest'.

f_black
f_white
f_invert
f_copy Only makes sense with wm_RasterOp (and it's probably the only opcode that does make sense with wm_RasterOp)
f_BlackOnWhite
f_WhiteOnBlack

wm_SetTitle (char *s)
Sets the title line for the current window to *s*. Each window has a title line at its top which describes the entity being viewed through the window. For example, if the client program is an editor, then it should be the name of the file being edited.

wm_SetProgramName (char *s)
Sets the program name field in the current window's title line to *s*. Normally, programs don't set this, instead they use the **program** macro to declare the name of the program. **Program(name)** is placed in the main program, somewhere where global variables may be declared. It sets up a data structure which gets used by **wm_NewWindow** when a window is created and by **getprofile** when a preference option is being looked up.

wm_GetDimensions (int *width, int *height)
Sets *width and *height to the width and height of the current window.

wm_SetDimensions (MinWidth, MaxWidth, MinHeight, MaxHeight)
Sets the *preferred* minimum and maximum height and width for the current window. This does not change the actual dimensions of the window, it only provides hints to the window manager to guide its automatic selection of window sizes.

wm_SetRawInput ()

Sets the current window into raw input mode: each time that the user types a character it will immediatly be shipped down to the client, rather than saving up a line and waiting for newline to be typed.

m_DisableInput ()
Disables input from the current window. If the user types in this window, it will be ignored.

wm_EnableInput ()
Enables input from the current window. Characters that the user types will be passed to the client program. This is the default state.

wm_RasterOp (sx,sy,dx,dy,w,h)
Performs a RasterOp operation using the current function. (sx,sy) is the origin of the source rectangle and (dx,dy) is the origin of the destination rectangle. (w,h) is the width and height of both rectangles.

wm_RasterSmash (dx,dy,w,h)
Performs a RasterOp without a source rectangle. The meaningful operations are f_black, f_white and v_invert.

wm_FillTrapezoid (x1, y1, w1, x2, y2, w2, f, c)
Fills the trapezoid with *x1,y1* as its upper left corner, *w1* as the width of the top, *x2,y2* as its lower left corner, and *w2* as its width. The trapezoid will be filled with character *c* from font *f*. If *f* is -1 then the default shape font (shape10) will be used. **Bogosity:** *f* is a font index, not a font pointer.

wm_ClearWindow ()
Clears (to white) the current window.

wm_SawMouse (int *action, int *x, int *y)
If you've see a *wm_MouseInputToken* on the window input stream then wm_SawMouse will return in action, x and y the event that occurred on the mouse and its coordinates at the time. *wm_MouseInputToken* is a simple character. When a client program is reading characters from its window input it should be checking for occurences of *wm_MouseInputToken* . If one is seen, then *wm_SawMouse* should be called to decode the action and coordinate information. The *action* parameter tells the client what sort of event has occurred. For an explanation of the interpretation of *action* , read the section on *wm_SetMouseInterest* .

wm_SetMouseInterest (mask)
Defines a mask which describes the mouse events in which the client is interested. The mask is constructed by **or ing** together several values from *MouseMask* .

MouseMask (event)
Constructs a constant which represents the *event* occuring on the *button.*

Values for	
button	event
LeftButton	UpMovement
MiddleButton	DownTransition
RightButton	UpTransition
	DownMovement

wm_SetMouseGrid (n)
Causes the window manager to report mouse movements only when the mouse moves at least *n* pixels from its last reported position.

wm_SetMousePrefix(char *string)
Causes the window manager to prefix all mouse position reports with *string* instead of **wm_MouseInputToken.**

wm_AddMenu (char *string)
Adds the given string to the menu for the current window. Each window has associated with it one hierarchic menu: menu entries may have submenus, which in turn may have yet more submenus. *String* consists of a series of comma separated entry names, followed by a colon, followed by the response string. The response string is the string which will be transmitted back to the client program when that menu entry is selected. For example:

```
wm_AddMenu("Directory,List:ls -l\n");
wm_AddMenu("Directory,Name:pwd\n");
```

The first call to *wm_AddMenu* will add an entry in the root menu named "Directory". This entry will have a submenu, and in that submenu will be defined an entry named "List". This entry will be a leaf of the menu hierarchy and will have associated with it the string "ls -ln". If the user selects "List" in the "Directory" submenu then this string ("ls -ln") will be transmitted back to the client program through its window input channel (winin).

The second call to *wm_AddMenu* simply adds a "Name" entry to the "Directory" menu and associates the string "pwdn" with it.

The colon and rresponse string may be omitted, in which case the menu entry will be removed.

wm_SetMenuPrefix (char *string)
Causes all following menu selections to be prefixed by *string.* The string defaults to the empty string.

wm_DisableNewlines ()
Normally when a newline character is written to a window it will cause scrolling if occurs at the bottom of the window. *wm_DisableNewlines* disables this behaviour —newlines will simply move the text pointer off the bottom of the window and subsequent text will be clipped and not displayed.

struct wm_window *CurrentUserWindow
A pointer to the currently selected window.

FILE *winout
An output file corresponding to the current window. Writing text here causes the text to appear in the window just as though it were a glass tele-

type. The text will be drawn with the upper left hand corner of the first character being placed at the current position. The current position will be left just past the upper right hand corner of the last character.

FILE *winin
An input file corresponding to the current window. Reading from here returns characters typed by the user while pointing at this window.

Text

struct font *wm_DefineFont (fontname) Defines a font, given a *fontname,* and returns a pointer to be used by subsequent **wm_SelectFont's.** *Fontname* is a string which names a font. For example, "TimesRoman10i" specifies Times Roman 10 point italic; "CMR10b" specifies Computer Modern Roman 10 point boldface. If the font specified doesn't exist, one which is "close" will be used instead.

wm_SelectFont (struct font *fontpointer) Causes the font specified by the *fontpointer* to be used for subsequent character printing. *Fontpointer* is created by calling **wm_DefineFont.**

```
wm_SelectFont(wm_DefineFont("TimesRoman10i"))
```

causes all further printing to use Times Roman 10 point italic. The reason for separating the **wm_DefineFont** and **wm_SelectFont** is to avoid having to do many repeated font lookups —the client program is expected to save the value returned from **wm_DefineFont** and reuse it.

wm_StringWidth (string, int *x, int *y) Finds the width of the given string in the current font in the current window. The x-width is returned in x, and the y-width in y. For normal left-to-right fonts, y will be zero and x will be the width of the string. The width is measured starting at the origin of the leftmost character up to the origin of the character which would immediatly follow the rightmost character.

wm_DrawString (x, y, flags, string) Draws the given string relative to the given coordinates, according to the flags. The flags control alignment of the string relative to the height and width of the string.

Height alignment options	
wm_AtTop wm_AtBottom wm_AtBaseline wm_BetweenTopAndBottom	wm_AtRight wm_AtLeft wm_BetweenLeftAndRight wm_BetweenTopAndBaseline

The flags argument is constructed by oring together one height alignment option and one width alignment option. Either may be omitted and wm_AtBaseline and wm_AtLeft will be taken as defaults.

A slight confusion is possible in understanding the differences between wm_BetweenTopAndBottom and wm_BetweenTopAndBaseline. The former properly centers, taking into account descenders (Like the tail on a lowercase 'y'), where the latter ignores descenders. The latter is likely to be more aesthetically pleasing.

For example:

```
wm_DrawString (WindowWidth/2,
        WindowHeight/2,
        wm_BetweenTopAndBaseline|wm_BetweenLeftAndRight,
        "Don't Panic");
```

will print the string "Don't Panic" correctly centered in the current window using the current font.

wm_printf (x, y, flags, format, args) This procedure is similar to wm_DrawString except that instead of taking a single string as an argument, it takes a full "printf" format and arguments.

wm_SetCursor (struct font *f; char c) Sets the cursor that is used in following the mouse to character 'c' from font 'f'. The cursor will only have this shape when it is inside the current window.

wm_SetStandardCursor (c) Sets the cursor from the standard icon font (icon12). The following cursor names and shapes are defined in usergraphics.h:

```
wm_GunsightCursor
wm_CrossCursor
wm_HourglassCursor
wm_RightFingerCursor
wm_HorizontalBarsCursor
wm_LowerRightCornerCursor
wm_PaintbrushCursor
wm_UpperLeftCornerCursor
wm_VerticalBarsCursor
wm_DangerousBendCursor
wm_CaretCursor
```

You can look at the available standard cursors by typing:
samplefont icon12

wm_SetSpaceShim (n) Sets the space shim value to *n*. A *shim* is some padding that is added on to the right of a character. The space shim applies only to space characters. After calling **wm_SetSpaceShim** all following space characters will have *n* added to their width.

wm_SetCharShim (n) is similar to **wm_SetSpaceShim** except that it applies to all characters, including spaces.

Color

This section needs to be written. But first, we should figure out what we're doing...

Window Manipulation

wm_HideMe() Hide the current window. Has no effect if the current window is currently hidden.

wm_ExposeMe () Expose the current window. Has no effect if the current window is currently exposed.

wm_DeleteWindow () Deletes the current window from the screen. Further operations on that window will fail.

wm_AcquireInputFocus () Acquires the input focus for the current window. When the input focus is 'acquired' by a window, all subsequent characters typed by the user are send to that window. This will continue until the input focus is acquired by some other window. The acquisition of the input focus only affects characters typed by the user, it does not affect mouse hits or menu selections.

wm_GiveUpInputFocus () Gives the input focus back to the last window that had the input focus.

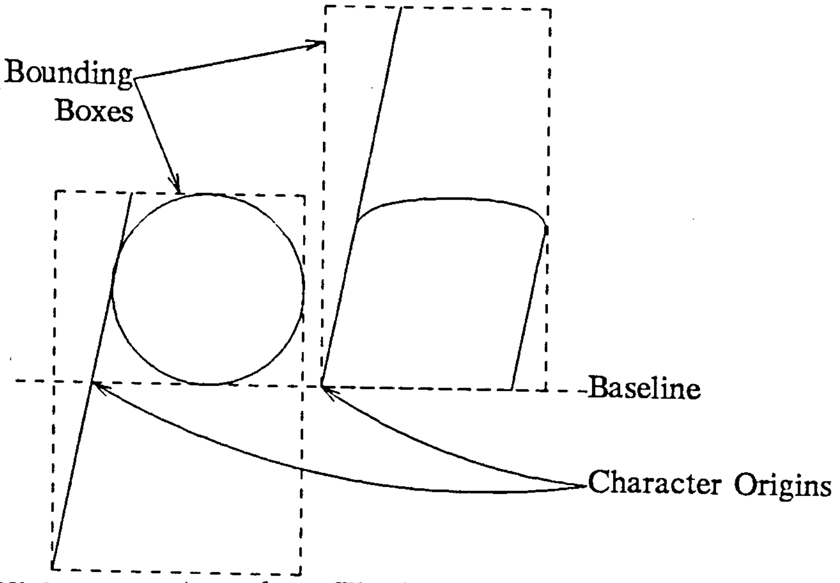
wm_IHandleAcquisition () Declares to the window manager that the client is willing and able to hadle input focus acquisition for himself. Normally, if no client is handling input focus acquisition, if the user wants to change the input focus he points the mouse at it and clicks a button. This mouse event will be thrown away and the input focus will shift. The user can then type at that window and use the mouse. If the client has executed **wm_IHandleAcquisition**, then all mouse clicks get sent to the client and the window manager will not automatically shift the input focus. That client can then (for example) acquire the input focus whenever it sees that first mouse event. The posession of the input focus will be encoded in recieved mouse events.

Fonts

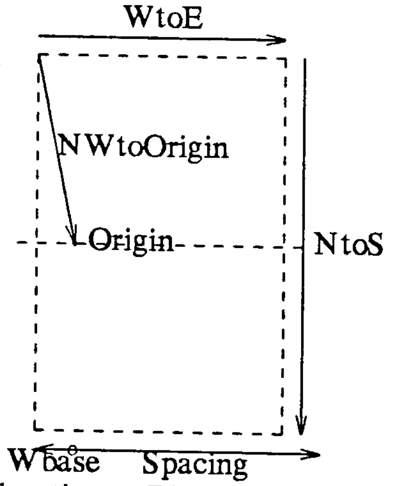
This section describes the font mechanism used by the window manager. Most writers of client programs don't need to know much of what is described here.

A *font* is a collection of *icons* . An icon is a thing which can be drawn to make a mark. The mark can have any shape. A font has two parts: a header which contains the name of the font and some summary information, and an array of icons. An icon also has two parts: One of generic information and one of specific information. The generic information describes the properties of the icon that are independant of its represenation: information about the bounding box and spacing. The specific information describes the properties that are dependant on the representation of the icon —those that are interesting only to a routine that is actually drawing the icon. At the moment, the specific information will either be a bitmap or a list of vectors.

An icon represents some shape enclosed within a bounding box. Typically the icons are characters like those in this illustration. The bounding box for each character is at least large enough to completely enclose the character, and it is usually a tight fit. Within that bounding box is a distinguished point called the Origin. The origin is a point which is on the character's baseline and which is at the optical left edge of the character. The optical left edge of a character is often at the left edge of its bounding box. For italic characters with descenders, like the 'p' here it is inset from the left edge. The bounding boxes of successivly drawn characters may or may not overlap. The bounding box of an italic 'y' or 'p' will typically overlap the bounding box of the preceeding character so that their decender can sweep under it. Similarly, the ascender of an italic 'f' usually extends over the following character.



Associated with each character are five vectors:
NWtoOrigin is a vector which goes from the north west corner of the bounding box to the origin.
WtoE goes from the west edge to the east edge.
NtoS goes from the north edge to the south edge.
Wbasegoes from the origin due west to the west edge of the character.
Spacing goes from the origin of the character to the place where the origin of the next character should be placed when this character is drawn with one following it.
These five paramaters could have been scalars, but were done as vectors instead in order to make manipulating rotated fonts easier.



A font file is laid out according to the following declarations. The file will start with a single instance of **struct font** . This contains the vector of icons. Each icon contains two offset pointers: one to the generic part and one to the specific part for the icon. All of the generic parts occur in the file before the specific parts. Charracters with matching generic or specific parts will share the corresponding space.

```
struct SVector {                /* Short Vector */
        short x, y;
};
```

/ Given a pointer to an icon, GenericPart returns a pointer to its IconGenericPart */*

```
#define GenericPart(icon) ((struct IconGenericPart *) (((int) (icon)) + (icon) ->
                        OffsetToGeneric))

/* Given a character and a font, GenericPartOfChar returns the corresponding
                        IconGenericPart */
#define GenericPartOfChar(f,c) GenericPart(& ((f)->chars[c]))

struct IconGenericPart { /* information relating to this icon that is of general in-
                        terest */
        struct SVector Spacing;  /* The vector which when added to the origin of
                                 this character will give the origin of the next char-
                                 acter to follow it */
        struct SVector NWtoOrigin;     /* Vector from the origin to the North
                                 West corner of the characters bounding box */
        struct SVector NtoS;     /* North to south vector */
        struct SVector WtoE;     /* West to East vector */
        struct SVector Wbase;    /* Vector from the origin to the West edge paral-
                                 lel to the baseline */
};
struct BitmapIconSpecificPart {  /* information relating to an icon that is necessary
                        only if you intend to draw it */
        char type;                       /* The type of representation used for this
                                 icon. (= BitmapIcon) */
        unsigned char rows,      /* rows and columns in this bitmap */
        unsigned char cols;
        char orow,                       /* row and column of the origin */
        char ocol;                       /* Note that these are signed */
        unsigned short bits[1];  /* The bitmap associated with this icon */
};

struct icon {                            /* An icon is made up of a generic and a
                        specific part. The address of each is "Offset" bytes from
                        the "icon" structure */
        short OffsetToGeneric;
        short OffsetToSpecific;
};


/* A font name description block. These are used in font definitions and in font
                        directories */
struct FontName {
        char FamilyName[16];  /* eg. "TimesRoman" */
        short rotation;                  /* The rotation of this font (degrees;
                                 + ve=> clockwise) */
        char height;                     /* font height in points */
        char FaceCode;                   /* eg. "Italic" or "Bold" or "Bold Italic"
                                 */
};

/* Possible icon types: */
#define AssortedIcon 0  /* Not used in icons, only in fonts: the icons have an as-
                        sortment of types */
#define BitmapIcon 1    /* The icon is represented by a bitmap */
#define VectorIcon 2    /* The icon is represented as vectors */
```

```
/* A font. This structure is at the front of every font file. The icon generic and
                        specific parts follow. They are pointed to by offsets in the
                        icon structures */
struct font {
        short   magic;                  /* used to detect invalid font files */
        short   NonSpecificLength;      /* number of bytes in the font and generic
                                parts */
        struct FontName fn;     /* The name of this font */
        struct SVector NWtoOrigin;      /* These are "maximal" versions of the
                                variables by the same names in each constituent
                                icon */
        struct SVector NtoS;    /* North to South */
        struct SVector WtoE;    /* West to East */
        struct SVector Wbase;   /* From the origin along the baseline to the West
                                edge */
        struct SVector newline; /* The appropriate "newline" vector, its just NtoS
                                with an appropriate fudge factor added */
        char type;                      /* The type of representation used for the
                                icons within this font. If all icons within the font
                                share the same type, then type is that type, other-
                                wise it is "AssortedIcon" */
        short NIcons;                   /* The number of icons actually in this
                                font. The constant "CharsPerFont" doesn't actually
                                restrict the size of the following array; it's just
                                used to specify the local common case */
        struct icon     chars[CharsPerFont];
         /* at the end of the font structure come the bits for each character */
};

/* The value of font-> magic is set to FONTMAGIC. This is used to check that a
                        file does indeed contain a font */
#define FONTMAGIC 0x1fd

/* FaceCode flags */
#define BoldFace 1
#define ItalicFace 2
#define ShadowFace 4
#define FixedWidthFace 010

struct font *getpfont();  /* get font given name to parse */
struct font *getfont();   /* get font given parsed name */
struct icon *geticon();   /* get an icon given a font and a slot */
int LastX, LastY;               /* coordinates following the end of the last string
                        drawn */
```

Program Template

The following is a template for simple programs that use the window manager:

```
#include <usergraphics.h>
```

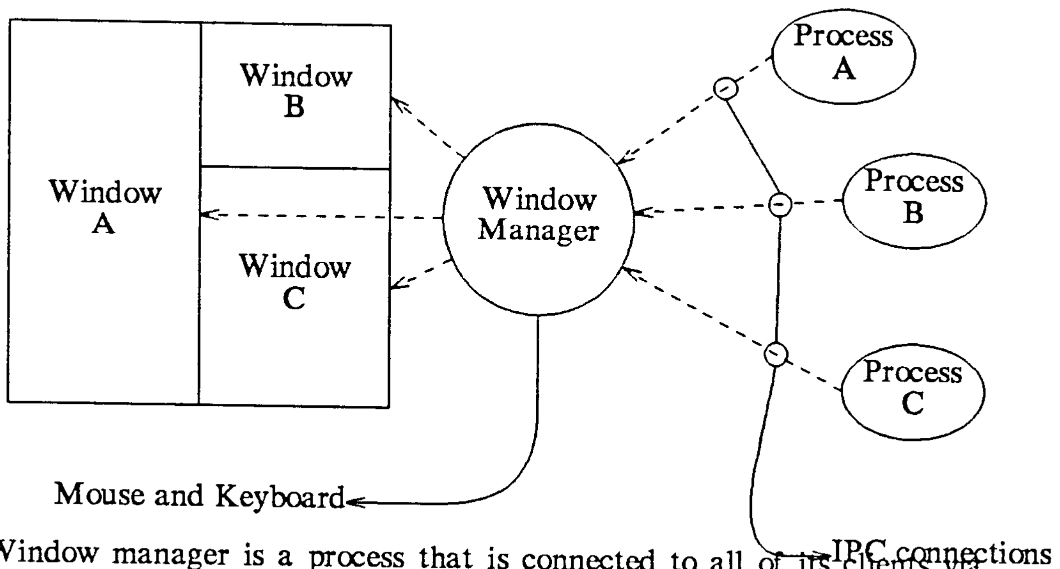
```
program(foo)

main () {
        wm_NewWindow ();
        ....
}
```

Sample Client Program

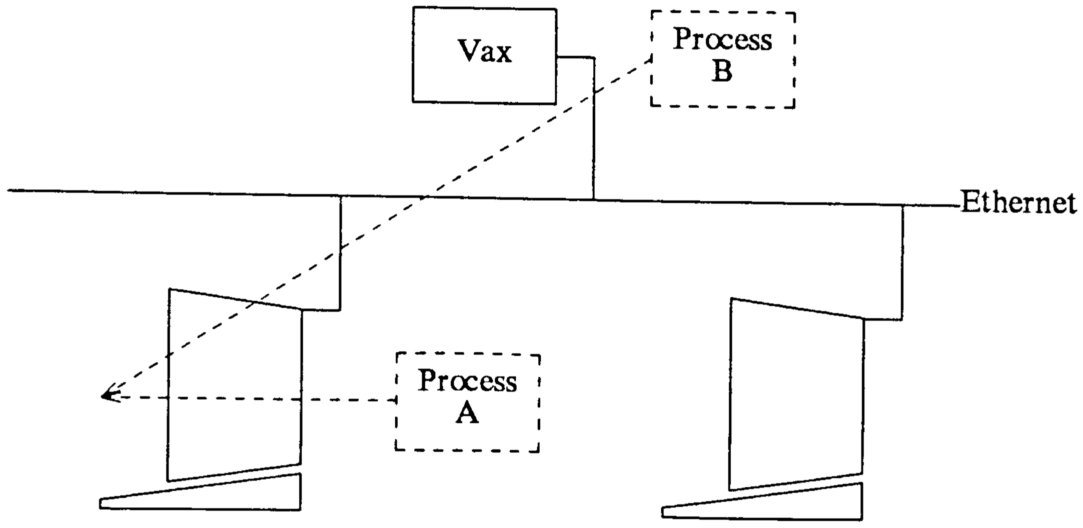
This is a simple clock which was written to test out user level graphics. It draws a face which consists of 12 tick marks arranged in a circle and the hour, minute and second hands. The clock face is updated every second.

```
#include <stdio.h>
#include "usergraphics.h"This library contains all of the definitions for the client in-
                         terface to the window manager
#include "clocktable.h"          This library contains a table of sines and cosines
#include <time.h>

struct tm LastTimeDisplayed;

int MidpointX,
int MidpointY;
int FaceRadius;
int RedrawRequested = 0;

FlagRedraw () {
        RedrawRequested++;
}

program (clock)

main () {
        int                     FaceInnerRingRadius;
        int                     n;
        int                     HourRadius, HourInner,
                                MinRadius, MinInner,
                                SecRadius, SecInner;
        register struct tm *CurrentTime;
        if (fork ())
                exit (0);
        FlagRedraw ();
        while (1) {
                long now = time (0);
                CurrentTime = (struct tm  *) localtime (&now);
                if (RedrawRequested) {
                        while (1) {
                                wm_GetDim& MidpointX, & MidpointY);
                        if (MidpointY)
                              break;
                        pause ();
                }
                wm_ClearWindow ();
                MidpointX /= 2;
                MidpointY /= 2;
                FaceRadius = MidpointX;
                if (MidpointY < FaceRadius)
                        FaceRadius = MidpointY;
                FaceInnerRingRadius = FaceRadius * 19 / 20;
                HourRadius = FaceRadius * 12 / 20;
                HourInner  = FaceRadius *  9 / 20;
                MinRadius  = FaceRadius * 16 / 20;
                MinInner   = FaceRadius *  9 / 20;
                SecRadius  = FaceRadius * 18 / 20;
                SecInner   = FaceRadius *(-6)/ 20;
                for (n = 0; n < 60; n += 5) {
                        wm_MoveTo (FaceInnerRingRadius * angtbl[n].xf
                                                    / SCALEANG + Mid-
                                                    pointX, FaceInner-
                                                    RingRadius * angtbl[n].yf
                                                    / SCALEANG + Mid-
                                                    pointY);
                        wm_DrawTo (FaceRadius * angtbl[n].xf /
                                                    SCALEANG + Mid-
                                                    pointX, FaceRadius *
                                                    angtbl[n].yf /
                                                    SCALEANG + Midpoin-
                                                    tY);
                }
        }
        UpdateHand (CurrentTime -> tm_sec,
                                & LastTimeDisplayed.tm_sec, SecRadius,
                                SecInner);
        UpdateHand (CurrentTime -> tm_min,
                                & LastTimeDisplayed.tm_min, MinRadius,
                                MinInner);
        UpdateHand (((CurrentTime -> tm_hour * 5) %60) + Current-
                                Time -> tm_min / 12,
                                & LastTimeDisplayed.tm_hour, HourRa-
                                dius, HourInner);
        wm_MoveTo(MidpointX<<1, MidpointY<<1);
        fflush (winout);
        if (RedrawRequested==0)
                sleep (1);
        else
                RedrawRequested--;
    }
}


UpdateHand (NewTime, OldTime, OuterRadius, InnerRadius)
int     *OldTime; {
        if (*OldTime != NewTime |RedrawRequested) {
                if (RedrawRequested==0 && *OldTime != NewTime) {
                        wm_SetFunction (f_white);
                        DisplayHand(*OldTime, OuterRadius, InnerRadius);
                }
                wm_SetFunction (f_black);
                DisplayHand(NewTime, OuterRadius, InnerRadius);
                *OldTime = NewTime;
        }
}

DisplayHand(Time, OuterRadius, InnerRadius) {
        int OuterX = OuterRadius * angtbl[Time].xf / SCALEANG + MidpointX;
        int OuterY = OuterRadius * angtbl[Time].yf / SCALEANG + MidpointY;
        if (InnerRadius> 0) {
                int LastTime = (Time== 0 ? 59 : Time-1);
                int NextTime = (Time==59 ?  0 : Time+1);
                wm_MoveTo (MidpointX, MidpointY);
                wm_DrawTo (InnerRadius * angtbl[NextTime].xf / SCALEANG
                                        + MidpointX, InnerRadius *
                                        angtbl[NextTime].yf / SCALEANG +
                                        MidpointY);
                wm_DrawTo (OuterX, OuterY);
                wm_MoveTo (MidpointX, MidpointY);
                wm_DrawTo (InnerRadius * angtbl[LastTime].xf / SCALEANG +
                                        MidpointX, InnerRadius *
                                        angtbl[LastTime].yf / SCALEANG +
                                        MidpointY);
        }
        else if (InnerRadius< 0)
                wm_MoveTo (InnerRadius * angtbl[Time].xf / SCALEANG +
                                        MidpointX,
                  InnerRadius * angtbl[Time].yf / SCALEANG + MidpointY);
        else
                wm_MoveTo (MidpointX, MidpointY);
        wm_DrawTo (OuterX, OuterY);
}
```

Implementation notes



The Window manager is a process that is connected to all of its clients via socket-to-socket IPC channels. All graphics operations cause messages to be sent between the client and the window manager. The remote procedure call protocol has beed especially crafted for this application. Each procedure call turns into a sequence of bytes. The first is the opcode and the rest (the count is determined by looking up the opcode in a table) are the arguments. Opcodes 0-127 are reserved for the ASCII characters, the rest are in the range 128-255. Many procedure call may be packed into a packet and a procedure call may be split across packet boundaries. If the call doesn't return a value, then it isn't issued immediatly, rather it is just queued up until either the buffer is filled or an explicit fflush(winout) is executed.



Processes on any machine, whether or not they are workstations, can create windows on any display.

Still to be documented:

wm_StdioWindow()
wm_ShowBits(x,y,w,h,b)
wm_DefineRegion(id,x,y,w,h)
wm_SelectRegion(id)
wm_ZapRegions()
wm_SetClipRectangle(x,y,w,h)
wm_WriteToCutBuffer()
wm_ReadFromCutBuffer(n)
wm_RotateCutRing(n)
wm_SaveRegion(id,x,y,w,h)
wm_RestoreRegion(id,x,y)
wm_ForgetRegion(id)
wm_HereIsRegion(id,w,h)
wm_ZoomFrom(x,y,w,h)
wm_LinkRegion(newid,oldid)
wm_NameRegion(id,name)