CGFX7

The NitrOS-9 C Programming Graphics Libary

From nitros9

Contents

Introduction

This is the seventh edition of my CGFX library replacement, with source and some real documentation. Version 7 adds a few new functions, which are described in detail on the following pages. Version 7 also fixes some bugs in the buffering added in version 5.

This library and its source are public domain. This means that you may use and distribute this library as much as you like, but you may not charge for it (other than cost of a disk or whatever). You may freely use this library for your own developments, commercial or otherwise. Feel free to change the source as needed for your own products, but please do not release changed version-tell me instead and I will add/change the lib as necessary. I make no warranties, express or implied, as to the reliability and/or usability of this library.

If you come across any bugs/omissions/spelling errors in the library, or if you have any suggestions for improvement, send mail to any of the following addresses:

Michael Sweet
209 Kennedy Plaza
Utica, NY 13502
sweetmr@sct60a.sunyct.edu (internet/bitnet)
DODGECOLT (Delphi)

Buffering

As previously stated, this version of the library has optional buffering for the output of certain functions. To include the buffering code, simply use the function **_Flush()**. This will inform the linker that buffered output is desired. The buffer used is 256 bytes long, and is flushed when one of the following occurs:

- The buffer is filled or you try to write more than 256 bytes at a time.
- You call the **_Flush()** function to flush the buffer. This is not the same as the **flush()** or **flushall()** functions used with FILE type I/O.
- You call one of the following functions:
  - **writeln()**, **cwriteln()**
  - **read()**, **cread()**, **readln()**, **creadln()**
  - **KilBuf()**, **GPLoad()**
  - **_ss_mgpb()**, **_ss_wset()**, **_gs_styp()**
  - **DWSet()**, **DWEnd()**, **DWSelect()**
  - **OWSet()**, **OWEnd()**, **CWArea()**
  - **Play()**
- You do I/O with a different path.

The increase in speed for some things can be phenomenal. The table below shows a few benchmarks for comparison. The 'dots' program fills the screen by setting each pixel individually. The 'hline' program does the same, but uses horizontal lines instead. The 'vline' program uses vertical lines. 'Text' writes 10,000 characters to a hardware text window one at a time.

Execution Time (mins)

Program	Unbuffered	Buffered
dots	9:05	4:54
hlines	0:03	0:03
vlines	0:04	0:04
text	0:37	0:13

Note that for the tests using lines, no measurable difference was detected. This is primarily due to the relatively few calls needed. Larger numbers of lines would probably reveal a slight speed increase with buffered output. All tests were performed using the Fast GRFDrv patch.

Many thanks to Eddie Kuns who got me thinking about adding buffered output to the lib and to Bob van der Poel for pointing out an omission I made with the **_Flush** function.

SetAlarm, GetAlarm, ClrAlarm, SigAlarm

Usage

```
#include <time.h>

int SetAlarm(timbuf)
struct sgtbuf *timbuf;

int GetAlarm(timbuf)
struct sgtbuf *timbuf;

int ClrAlarm()

int SigAlarm(timbuf,signo)
struct sgtbuf *timbuf;
int signo;
```

Description

SetAlarm() sets the system alarm to the time in **timbuf**. When the current time equals the alarm time, an alarm is sounded for sixteen seconds.

GetAlarm() gets the system alarm time and places it in the time buffer pointed to by **timbuf**. If the alarm has not been set, the time returned will be 00/00/00 00:00.

ClrAlarm() will clear the system alarm.

SigAlarm() will set the system alarm to the time in **timbuf**. When the current time equals the alarm time, a signal (**signo**) is sent to the calling process.

Only one alarm can be active at a time.

BUp, BDown, RBUp, RBDown

Usage

```
BUp(path, column, row, s, fg, bg)
int path, column, row, fg, bg;
char *s;

BDown(path, column, row, s)
int path, column, row;
char *s;

RBUp(path, column, row, fg, bg)
int path, olumn, row, fg, bg;

RBDown(path, column, row, fg, bg)
int path, column, row, fg, bg;
```

Description

BUp() and **BDown()** are two functions that handle the placement of three-dimensional rectangular buttons on the screen. The **BUp()** function is used to initially place the button on the window. **S** is a pointer to the characters that will appear inside the button. If these characters are not in the current window font, then the font must be set before calling this function, as it does not distinguish between control characters and data. The button is created with a foreground color or **fg** and a background color or **bg**. A one character border (one line above and below, one character to the left and right) must be accounted for, as the button edges reach into these areas. Therefore, each button requires 3 window rows and 2+length of **s** columns.

The **BDown()** function is called after a **BUp()** to push the button down. This function uses buffer 255 of the current process's G/P buffers to push the button down, and kills it once the operation is completed. Note that this function uses the button that was drawn using **BUp()**. You *must* call **BUp()** prior to calling **BDown()**. The button can be released by calling **BUp()** again, which redraws the original button.

RBUp() and **RBDown** are functions which place three dimensional radio buttons on the window specified by **path**. Unlike **BUp()** and **BDown()**, you do not need to call **RBUp()** prior to calling **RBDown()**. The button is drawn at column **column** and row **row** with a foreground color of **fg** and a background color of **bg**.

For these functions to work properly, you must have window scaling turned off. Otherwise, the lines that are drawn will not be placed properly (resulting in a garbled display).

BColor, Border, DefColr, FColor, LSet, Palette, PSet, ScaleSw, SetGC

Usage:

```
#include <buffs.h>

int BColor(path,prn)
int path,prn;

int Border(path,prn)
int path,prn;

int DefColr(path)
int path;

int FColor(path,prn)
int path,prn;

int LSet(path,grpnum,bufnum)
int path,grpnum,bufnum;

int Palette(path,prn,colno)
int path,prn,colno;

int PSet(path,grpnum,bufnum)
int path,grpnum,bufnum;

int ScaleSw(path,bsw)
int path,bsw;

int SetGC(path,grpnum,bufnum)
int path,grpnum,bufnum;
```

Description:

BColor() will set the background color on **path** to **prn**. **Border()** will set the border color on **path** to **prn**. **DefColr()** restores the default palette settings on **path**. **FColor()** will set the foreground color on **path** to **prn**.

LSet() sets the point logic for **path**. **Logiccode** can be one of the following (defined in **<buffs.h>**):

```
/* logic modes */
#define LOG_NONE 0 /* no logic - just put it there (default) */
#define LOG_AND 1  /* AND the contents of the screen with the data */
#define LOG_OR 2   /* OR the contents of  the screen with the data */
#define LOG_XOR 3  /* XOR the contents of the screen with the data */
```

PSet() will set the drawing pattern to be used on path. A number of standard patterns are included with OS-9. The **<buffs.h>** header file defines them as follows:

```
/* buffer group numbers */
#define GRP_PAT2 203 /* Two color patterns */
#define GRP_PAT4 204 /* Four color patterns */
#define GRP_PAT6 205 /* Sixteen color patterns */

/* pattern buffers */
#define PAT_SLD 0 /* Solid (default) */
#define PAT_DOT 1 /* Dots */
#define PAT_VRT 2 /* Vertical lines */
#define PAT_HRZ 3 /* Horizontal lines */
#define PAT_XHTC 4 /* Crosshatch */
#define PAT_LSNT 5 /* Left slanted lines */
#define PAT_RSNT 6 /* Right slanted lines */
#define PAT_SDOT 7 /* Small dots */
#define PAT_BDOT 8 /* BIG dots */
```

Palette() will set palette register **prn** to **colno** on **path**. **Prn** is any legal register number from 0-15. **Colno** is the palette value, derived from the following:

		Bit			
5	4	3	2	1	0
RED	GREEN	BLUE	red	green	blue

The operating system automatically converts palette settings when using a composite monitor.

ScaleSw() will turn graphics scaling on or off. If **bsw** is 0, then scaling is turned off. Otherwise, scaling is turned on. Graphics are scaled from 640x192 to the size of the window.

SetGC() will set the current mouse pointer on **path** to **grpnum,bufnum**. The standard pointer buffers are defined in **<buffs.h>**:

```
#define GRP_PTR 202 /* Pointer group */

/* mouse pointer buffers */
#define PTR_ARR 1 /* Arrow */
#define PTR_PEN 2 /* Pencil */
#define PTR_LCH 3 /* Large Cross Hairs */
#define PTR_SLP 4 /* Hourglass (sleep) */
#define PTR_ILL 5 /* Illegal (circle with a slash) */
#define PTR_TXT 6 /* Text */
#define PTR_SCH 7 /* Small Cross Hairs */
```

On error, these functions return -1, with the error number in **errno**.

cread, creadln, cwrite, cwriteln

Usage:

```
int cread(path,s,n)
int path,n;
char *s;

int creadln(path,s,n)
int path,n;
char *s;

int cwrite(path,s,n)
int path,n;
char *s;

int cwriteln(path,s,n)
int path,n;
char *s;
```

Description:

These functions are variations of the standard **read()**, **readln()**, **write()**, and **writeln()** functions for use with C NULL-terminated strings. The **cread()** and **creadln()** functions use the **I$ReadLn** call to get a string of characters. The **cread()** function will replace the terminating carriage return with a NULL, while the **creadln()** function will append a NULL to the end of the string.

The **cwrite()** and **cwriteln()** functions will write a maximum of *n* characters to the specified path using the respective system call- **I$Write** for **cwrite()** and **I$WriteLn** for **cwriteln()**.

Dialog

Usage:

```
#include <dialog.h>

int Dialog(path,dlgptr,column,row,width,length,fg,bg)
int path,column,row,width,length,fg,bg;
DIALOG *dlgptr;
```

Description:

Dialog() is a high-level dialog management function. It creates an overlay on **path** at **column,row** with a foreground and background color of **fg** and **bg** respectively. **Dlgptr** is a pointer to an array of **DIALOG** descriptions. The **DIALOG** type is defined in **<dialog.h>** as:

```
typedef struct { /* dialog structure */
char d_type;    /* type- 0=string, 1=button */
char d_column;  /* column position
within the overlay */
char d_row;     /* row within the overlay */
char d_key;     /* key assocated with this
button (0 for none) */
char d_val;     /* value to return to caller */
char *d_string; /* pointer to actual string to
be placed in overlay *
} DIALOG;        /* call this type DIALOG */
```

The **<dialog.h>** file also provides some constants for the **d_type** field in the **DIALOG** structure:

```
#define D_TEXT 0   /* ASCII text */
#define D_KEY 1    /* key-binding */
#define D_STRING 2 /* ASCII string box (accepts text) */
#define D_BUTTON 3 /* 3-D text button */
#define D_RADIO 4  /* Radio button */
#define D_END -1   /* End marker of array */
```

The actions which the **Dialog()** functions takes depends greatly on the type of the object that is being acted on.

D_TEXT

The **D_TEXT** type of object is a simple NULL-terminated string. It is placed in the overlay at **d_column,d_row**. This type of object basically lets you place text in the overlay that you don't want to change (i.e. have the user change.) The **d_key** and **d_val** fields should be set to 0 for this type.

D_KEY

The **D_KEY** type of object binds a key to a value to be returned by the **Dialog()** function. Whenever the user presses that key (and is not editing a string-see below) the **Dialog()** function will return **d_val**. **D_string**, **d_column**, and **d_row** should all be set to 0.

D_STRING

The **D_STRING** type of object is a string-requester box. The size of the box is determined by the length of the string pointed to by **d_string**. Whenever the user selects the string box, he is put into an editing mode where each keypress is placed in the string, minus control chars which are used for editing. All editing features of Kevin Darling's SCF patch are available. When the user presses ENTER, one of two things will happen. If the **d_val** field is non-zero, the **Dialog()** function will return that value to the calling function. If not, the string box is unselected and the user can continue to use the dialog box. It is highly recommended that you dynamically allocate memory for each string that you use. One 'feature' of the Microware C compiler is that it will not copy constant strings to your process's data space. Instead, it uses the copy(s) in the program module directly. This can lead to some problems when you run the same program in different windows. See the example below for the easy way around this.

D_BUTTON

The **D_BUTTON** type of object is a three dimension rectangular button with text on the top of the button. Whenever the user presses the key associated with this object or clicks on it, **d_val** is returned.

D_RADIO

The **D_RADIO** type of object is a simple radio button. If the user presses the key associated with the object or clicks on it, then the button is toggled. The current state of the button is held in the **d_val** field. The **d_string** field holds a list of opposing radio buttons. That is, when you click on a particular button, any other associated buttons are pulled up (only if necessary.) The list MUST be a character array which contains the array indexes of the opposing buttons. The indexes start at 1, as the NULL terminator is used to end the list. The button being defined should not be included in the list (in other words, only the button numbers which this button opposes.) If the **d_string** field is NULL, then **Dialog()** assumes there are no opposing buttons.

D_END

The **D_END** type of object signals the end of the array of objects in the dialog box (pun intended!) This must be the last element in the array you send to the **Dialog()** function.

Example:

```
#include <dialog.h>
#include <buffs.h>

DIALOG temp[]={{D_TEXT,0,0,0,0,"Abort, Retry, Fail?"},
               {D_BUTTON,3,2,'A','A',"ABORT"},
               {D_BUTTON,10,2,'R','R',"RETRY"},
               {D_BUTTON,17,2,'F','F',"FAIL"},
               {D_STRING,1,4,0,0,0}, /* set up string
                                        later */
               {D_RADIO,26,4,'B',0,0},
               {D_TEXT,23,4,0,0,"On"},
               {D_END,0,0,0,0,0}};

main()
{
 char ch,s[21];

 strcpy(s,"                    "); /* put blanks in
                                      the string */

 temp[4].d_string=s; /* point to string */

 SetType(1,5,0,1); /* set screen type to 5 */

 _ss_mous(1,3,10,1); /* turn mouse on */
 SetGC(1,GRP_PTR,PTR_ARR); /* ditto for pointer */

/* now, call the dialog function... */
 ch=Dialog(1,temp,20,10,30,8,0,1);

 if (ch=='A') /* print return code */
  puts("Abort!");
 else if (ch=='R')
  puts("Retry!");
 else
  puts("Fail!");

 _ss_mous(1,0,0,0); /* turn mouse off */
 SetGC(1,0,0); /* and pointer */

 _Flush(); /* flush the buffer */
}
```

SetDPtr, RSetDPtr, Point, RPoint, Line, RLine, LineM, RLineM, Box, RBox, Bar, RBar, FFill, Circle, Ellipse, Arc, PutGC

Usage:

```
int SetDPtr(path,x,y)
int path,x,y;

int RSetDPtr(path,xo,yo)
int path,xo,yo;

int Point(path,x,y)
int path,x,y;

int RPoint(path,xo,yo)
int path,xo,yo;

int Line(path,x,y)
int path,x,y;

int RLine(path,xo,yo)
int path,xo,yo;

int LineM(path,x,y)
int path,x,y;

int RLineM(path,xo,yo)
int path,xo,yo;

int Box(path,x,y)
int path,x,y;

int RBox(path,xo,yo)
int path,xo,yo;

int Bar(path,x,y)
int path,x,y;

int RBar(path,xo,yo)
int path,xo,yo;

int FFill(path)
int path;

int Circle(path,radius)
int path,radius;

int Ellipse(path,xrad,yrad)
int path,xrad,yrad;

int Arc(path,xrad,yrad,xo1,yo1,xo2,yo2)
int path,xrad,yrad,xo1,yo1,xo2,yo2;

int PutGC(path,x,y)
int path,x,y;
```

Description:

These functions will perform various graphics primitives on path. Except for **PutGC**(), the output is affected by the current logic mode and pattern buffer. The functions perform the following operations:

SetDPtr()	moves the current draw pointer to (x,y).
RSetDPtr()	moves the current draw pointer to (xo,yo) relative to the current position.
Point()	sets the point at (x,y) to the current foreground color.
RPoint()	sets the point (xo,yo) relative to the current position to the current foreground color.
Line()	draws a line in the current foreground color from the current position to (x,y).
RLine()	draws a line in the current foreground color from the current position to the point (xo,yo) relative to the current position.
LineM()	draws a line as **Line()**, but then moves the current position to (x,y).
RLineM()	draws a line as **RLine()**, but then moves the current position relative to (xo,yo).
Box()	draws a rectangular line frame in the current foreground color from the current position to (x,y)
RBox()	draws a rectangular line frame in the current foreground color from the current position to a point (xo,yo) relative to the current position.
Bar()	draws a filled rectangle in the current foreground color from the current position to (x,y)
RBar()	draws a filled rectangle in the current foreground color from the current position to the point (xo,yo) relative to the current position.
FFill()	fills a closed region in the current foreground color starting at the current position.
Circle()	draws a circle in the current foreground color with its center at the current position.
Ellipse()	draws an ellipse in the current foreground color with its center at the current position.
Arc()	draws an arc in the current foreground color with its center at the current position. The size and direction of the arc is controlled by **xrad**, **yrad**, and the line relative to the current position (xo1,yo1)-(xo2,yo2.)
PutGC()	Places the mouse pointer at (x,y). This function is not compatible with the auto-follow mouse.

On error, these functions return -1 with the error number in **errno**.

Bugs:

The **Circle()**, **Ellipse()**, and **Arc()** radii are not scaled when draw scaling is enabled. Presumably, this bug will be fixed in a future release of the operating system.

Draw

Usage:

```
Draw(path,draw_string {,variable args} )
int path;
char *draw_string;
```

Description:

Draw() imitates the **DRAW** function from *Extended Color BASIC*. **Draw_string** contains a list of commands to be executed by **Draw().** It can consist of:

= **U{*n*}**	Move UP *n* units.
D{*n*}	Move DOWN *n* units.
L{*n*}	Move LEFT *n* units.
R{*n*}	Move RIGHT *n* units.
E{*n*}	Move at a 45 degree angle *n* units.
F{*n*}	Move at a 135 degree angle *n* units.
G{*n*}	Move at a 225 degree angle *n* units.
H{*n*}	Move at a 315 degree angle *n* units.
C{*n*}	Sets the foreground color to *n*.
A{*n*}	Sets the draw angle to *n*, where *n* is: 0 - 0 degrees 1 - 90 degrees 2 - 180 degrees 3 - 360 degrees
M*x*,*y*	Moves to (*x*,*y*). If *x* or *y* is preceded by a sign, the move is relative to the current position.
S{*n*}	Sets the current scale to *n*. The scale is figured by *n* divided by 4 (i.e. 1/4, 2/4, etc.)
B	This prefix disables drawing for the next command (blanks output)
N	This prefix forces the draw pointer to remain at its current

position after drawing. Normally, the draw pointer moves after each command.

The draw string can optionally contain sub-strings (indicated by **%s**) and integers (indicated by **%d**.) No clipping is done for output lines.

Bugs:

When buffering output, lines which fall outside the window will cause further writes to be ignored until the buffer is flushed.

getenv, putenv

Usage:

```
char *getenv(var)
char *var;

putenv(var,s)
char *var,*s;

extern char *_ENVFILE;
```

Description:

Getenv() will locate the variable **var** in **/dd/sys/env.file** and return the string following the equals (=) sign. If the string does not exist in the file, then a NULL pointer is returned.

Putenv() will set the environment variable **var** to **s**. If **var** does not exist, then it is appended to the file.

These functions will read the environment file into a buffer pointed to by **_ENVFILE**. This buffer is malloc'd to 1024 bytes. Future calls to **getenv** or **putenv** will reference this buffer unless it is freed and the pointer set to NULL.

FName, MVFName

Usage:

```
char *FName(path,title,fg,bg)
int path,fg,bg;
char *title;

char *MVFName(path,title,column,row,fg,bg)
int path,fg,bg;
char *title;
```

Description:

These functions provide an on-screen file-picking facility which allows a user to choose any file from any disk on the system. Path is the path number to use (usually 1, but it can be any window path that has read-write access.) **Fg** and **bg** are the foreground and background colors of the overlay that is created by these functions. **Title** is the NULL-terminated character string that is displayed at the top of the overlay. **Column** and **row** are the column and row where the overlay will appear. In the case of **FName()**, the overlay is centered on the current window. **MVFName()** is the *Multi-Vue* version of the **FName()** function. Centering was omitted for this function due to the way overlay coordinates are used by **GrfDrv**. **MVFName()** requires 22 columns and 11 rows for its overlay, while **FName()** and **FNameXY()** require 32 columns and 12 rows. Mouse sampling must be enabled with the **_ss_mous()** function prior to calling **MVFName()**.

The filename returned is stored in a static data area. Future calls to these functions will destroy the previous name.

Example:

```
#include <buffs.h>

char *p;

main()
{
    _ss_mous(1,3,10,1);  /* turn sampling on */
    SetGC(1,GRP_PTR,PTR_ARR); /* and mouse pointer */

    do
    {
        p=MVFName(1,"Filename?",20,10,0,2); /* get a filename */
        if (p!=NULL) /* if there was a valid name read */
            puts(p);    /* print the name */
    }
    while (p!=NULL); /* continue until user clicks on close box */

    _ss_mous(1,0,0,0); /* mouse sampling off */
    SetGC(1,0,0);      /* ditto for mousue pointer */
}
```

_Flush

Usage:

int _Flush()

Description:

_Flush() will flush the internal write buffer. It also informs the linker to include buffering code. If an error occurs during a _Flush() write, -1 is returned and the error number is placed in errno. 0 is returned on success.

BoldSw, Font, PropSw, TCharSw

Usage:

```
#include <buffs.h>

int BoldSw(path,bsw)
int path,bsw;

int Font(path,grpnum,bufnum)
int path,grpnum,bufnum;

int PropSw(path,bsw)
int path,bsw;

int TCharSw(path,bsw)
int path,bsw;
```

Description:

BoldSw() will turn boldfacing of text on or off. If **bsw** is 0, then boldfacing is turned off. Otherwise, boldfacing is turned on.

Font() will set the current font on **path** to **grpnum**, **bufnum**. The standard fonts are defined in **<buffs.h>** as:

```
#define GRP_FONT 200 /* Font group */

/* font buffers */
#define FNT_S8X8 1 /* Standard 8x8 font  (default) */
#define FNT_S6X8 2 /* Standard 6x8 font */
#define FNT_G8X8 3 /* Standard graphics character font */

/* special characters for font FNT_G8X8 */
#define CHR_RGT 0xc1 /* right arrow */
#define CHR_LFT 0xc2 /* left arrow */
#define CHR_DN 0xc3  /* down arrow */
#define CHR_UP 0xc4  /* up arrow */
#define CHR_TRI 0xc5 /* triple bar (for title bar) */
#define CHR_RSZ 0xc6 /* resize box (not used) */
#define CHR_CLZ 0xc7 /* close box */
```

```
#define CHR_MOV 0xc8 /* move box */
#define CHR_VBX 0xc9 /* vertical box  (for scroll markers) */

#define CHR_HBX 0xca /* horizontal box (for scroll markers) */
#define CHR_HRG 0xcb /* hourglass (Tandy menu) */
#define CHR_TRR 0xcc /* triple bar with open right side */
#define CHR_TRL 0xcd /* triple bar with open left side */
```

PropSw() will turn proportional spacing on or off, depending on the value of **bsw**.

TCharSw() will turn transparent characters on or off, depending on the value of **bsw**.

None of these functions has an effect on hardware text windows. -1 is returned on error, with the error number in **errno**.

DfnGPBuf, GPLoad, GetBlk, KilBuf, PutBlk, _ss_mgpb

Usage:

```
int DfnGPBuf(path,grpnum,bufnum,length)
int path,grpnum,bufnum,length;

int GPLoad(path,grpnum,bufnum,sty,sizex,sizey,length)
int path,grpnum,bufnum,sty,sizex,sizey,length;

int GetBlk(path,grpnum,bufnum,x,y,sizex,sizey)
int path,grpnum,bufnum,x,y,sizex,sizey;

int KilBuf(path,grpnum,bufnum)
int path,grpnum,bufnum;

int PutBlk(path,grpnum,bufnum,x,y)
int path,grpnum,bufnum,x,y;

char *_ss_mgpb(path,grpnum,bufnum,mapflag,size)
int path,grpnum,bufnum,mapflag,*size;
```

Description:

DfnGPBuf() will create a get/put buffer of size **length**. **Path** must be a path to a window device. **Grpnum** and **bufnum** are the group and buffer to define. If the buffer already exists or not enough memory is free to grant the request, -1 is returned and the error number is placed in **errno**. Otherwise, 0 is returned.

GetBlk() will copy a portion of the window specified by **x**, **y**, **sizex**, and **sizey**. **Path** must be a path to a window device. **Grpnum** and **bufnum** are the group and buffer that will be copied to. On error, -1 is returned and the error number is placed in **errno**.

GPLoad() will generate the necessary loading header to preload a get/put buffer. **Path** does not have to point to a window device, although no load is done unless it is a window device. **Sty** is the window type of the data (basically for color information.) A write

should directly follow this call to insure the buffer is not loaded with garbage. On error, -1 is return and the error number is placed in **errno**.

PutBlk() places a copy of a previously loaded get/put buffer at **x**,**y**. The logic used to place the buffer on the window is controlled by the current window pset logic (see Configuration Functions.) On error, -1 is returned and the error number is placed in **errno**.

KilBuf() will remove a get/put buffer from the system. Care should be taken when using this call as no 'link' count is maintained by the system. Deletion of standard get/put buffers is discouraged, and can cause major problems forcing a reboot. If buffer is 0, then all buffers in group **grpnum** are removed.

_ss_mgpb() will map a specified get/put buffer into/out of the calling process's address space. The direction of mapping is controlled by the **mapflag** variable. If it is 0, then the buffer is mapped out; otherwise the buffer is mapped in. On error, NULL (0) is returned and the error number is placed in **errno**.

Bugs:

There are several known **WindInt** bugs which affect the operation of these functions:

- Mapping of buffers larger than 8k is unpredictable.
- Mapping of multiple buffers smaller than 8k will confuse **WindInt**. The address returned seldom is correct.
- If you attempt to kill a non-existent buffer, WindInt will trash the buffer list forcing you to reboot. Killing all buffers with **bufnum** equal to 0 does work correctly, however.
- As stated above, things like killing font buffers while other processes are using them do nasty things forcing a reboot.
- **GetBlk()** can only get 639 dots across.

getstr

Usage:

```
int getstr(path,prompt,s,n,column,row,fg,bg)
int path,n,column,row,fg,bg;
char *prompt,*s;
```

Description:

Getstr will create an overlay window at **column** and **row**, and read a string from **path**. The carriage return is retained at the end of the string (**s**). **Getstr** will only read up to *n* characters. Care should be taken to insure that **s** is at least (*n* + 1) characters long.

_ss_ksns, _gs_ksns

Usage:

```
#include <keysense.h>

int _ss_ksns(path,keybits)
int path,keybits;

int _gs_ksns(path)
int path;
```

Description:

These functions provide a C interface to the Color Computer OS-9 get/setstat calls to get and set the keyboard status. The **_ss_ksns()** function sets key sensing for the keys specified by **keybits**. Values for **keybits** are combined from the definitions in the **<keysense.h>** header file:

```
#define SHIFTBIT 1
#define CTRLBIT 2
#define ALTBIT 4
#define UPBIT 8
#define DOWNBIT 16
#define LEFTBIT 32
#define RIGHTBIT 64
#define SPACEBIT 128
```

When the appropriate bit is set, sensing is enabled for the respective key. Any **keycode** that may have been generated by the key is no longer generated. The status of sensed keys can be obtained using the **_gs_ksns()** function. The return value from this function will show the status of any key that is currently being sensed. A set bit indicates the respective key is down.

Both functions return -1 on error. In the event of an error, the error number will be stored in the global variable **errno**.

Menu, MenuXY, MVMenu, MVMenuXY

Usage:

```
#include <menu.h>

int Menu(path,title,itemptr,fg,bg)
int path,fg,bg;
char *title;
ITEM *itemptr;

int MenuXY(path,title,itemptr,column,row,fg,bg)
int path,column,row,fg,bg;
char *title;
ITEM *itemptr;

int MVMenu(path,title,itemptr,fg,bg)
int path,fg,bg;
char *title;
ITEM *itemptr;

int MVMenuXY(path,title,itemptr,column,row,fg,bg)
int path,column,row,fg,bg;
char *title;
ITEM *itemptr;
```

Description:

These functions will create an overlay window on path and get a menu choice from the user. The overlay will have a foreground color of **fg** and a background color of **bg**. **Title** is a pointer to a NULL-terminated string of characters which is used as the title for the menu. If it is NULL, no title will be displayed. **Itemptr** is a pointer to an array of **ITEM** structures, which also must be NULL terminated. Special care should be taken not to pass a NULL **itemptr** or an **itemptr** array with zero active items, as these functions do not protect against it. The **MVMenu()** and **MVMenuXY()** functions work exactly as the **Menu()** and **MenuXY()** functions, and add mouse support for item selection. The calling program must initialize mouse sampling with the **_ss_mous()** function before calling any of the MV-series functions.

The **ITEM** structure is defined in **<menu.h>** as:

```
typedef struct {
char *itemname;  /* name of the item */
char enabled;    /* TRUE=enabled */
char (*itemfunc)(); /* function to call */
} ITEM;
```

When enabled equals 1, **itemfunc** is a pointer to a function to call when that item is selected. If this pointer is NULL, then no function is called. When **enabled** equals 2, **itemfunc** is a pointer to an array of **ITEM** structures. When the user selects this item, a sub-menu will appear containing the items in the **itemfunc** array. Sub-menus may call

functions or reference other sub-menus. **Submenus** return 16 times the item number, plus the main item that was elected. Do to 16 bit integers, sub-menus are limited to three levels deep. For example, if the user selects the second sub-item under the third main item, a value of 35 (16 * 2 + 3 = 35) would be returned. If the user should exit the sub-menu without choosing an item, only the item numbers gathered before the sub-menu will be returned. That is, the item number for that sub-menu will be 0 (so the previous example would then only return 3- 16 * 0 + 3.)

All of these functions return the item number selected or 0 if no selection was made (user pressed BREAK or moved mouse pointer off window.)

Example:

```
#include <menu.h>

ITEM mainmenu[]={{"Load a file",1,loadfunc},
                 {"Save a file",1,savefunc},
                 {"Quit",1,quitfunc},
                 {NULL,NULL,NULL}};

main()
{
    int num;

    do
        num=Menu(1,"Main Menu",mainmenu,5,2)
    while (num==0);
}
```

_ss_mous, _gs_mous

Usage:

```
#include <mouse.h>

int _ss_mous(path,sample_rate,timeout,follow)
int path,sample_rate,timeout,follow;

int _ss_msig(path,signo)
int path,signo;

int _gs_mous(path,mspacket)
int path;
MSRET *mspacket;
```

Description:

_ss_mous() sets the mouse parameters for the window on **path**. **Sample_rate** is the number of ticks to wait between samples. A **sample_rate** of 0 indicates that no sampling should be done. **Timeout** is the number of ticks between button click timeouts. If **timeout** is 0, then mouse signals are disabled. Otherwise, mouse signals are processed,

regardless of the **sample_rate**. **Follow** is a flag that enables/disables the auto-follow mouse cursor. When **follow** is 1, any mouse movements also move the mouse pointer. This value is ignored if **sample_rate** is 0.

_ss_msig() tells the window manager to send a signal **signo** to the current process when the user clicks one of the mouse buttons. Unlike **_ss_ssig()**, this function does not automatically send the signal if a button has already been clicked.

_gs_mous() will get an information packet containing the current state of the mouse. The packet is defined in **<mouse.h>** as:

```
typedef struct mousin {
char pt_valid,  /* is info valid? */
pt_actv,   /* active side */
pt_totm,   /* timeout initial value */
pt_rsrv0[2],/* reserved */
pt_tto,    /* time till timeout */
pt_tsst[2],/* time since start counter */
pt_cbsa,   /* current button state button A */
pt_cbsb,   /* current button state button B */
pt_ccta,   /* click count button A */
pt_cctb,   /* click count button B */
pt_ttsa,   /* time this state button A */
pt_ttsb,   /* time this state button B */
pt_tlsa,   /* time last state button A */
pt_tlsb,   /* time last state button B */
pt_rsrv1[6],/* reserved */
pt_stat,   /* window pointer location type */
pt_res;    /* resolution */
int  pt_acx,    /* actual x value */
pt_acy,    /* actual y value */
pt_wrx,    /* window relative x value */
pt_wry;    /* window relative y value */
} MSRET;

/* window regions for mouse */

#define WR_CNTNT 0 /* content region */
#define WR_CNTRL 1 /* control region */
#define WR_OFWIN 2 /* off window */
```

MouseKey

Usage:

```
int MouseKey(path)
int path;
```

Description:

MouseKey() checks the current mouse button and keyboard status and returns a value based on its findings. If the left mouse button has been pressed, -1 is returned. If the right mouse button has been pressed, -2 is returned. If a key has been pressed, the key code is returned. **MouseKey()** waits until a key or mouse button has been pressed.

MouseXY

Usage:

```
int MouseXY(path,x,y)
int path;
int *x,*y;
```

Description:

MouseXY() is a function which will return the current mouse character position to x and y. If the mouse pointer is currently off the window, or the window is not the current one, then **MouseXY()** returns -1.

NOTE: this function only returns accurate coordinates on hardware text screens and graphics screens using an 8 by 8 font.

movemem

Usage:

```
movemem(d,s,n)
char *d,*s;
int n;
```

Description:

Movemem is a memory moving function which handles overlapping source and destination areas. **N** bytes are copied from **s** to **d**.

The algorithm that is used employs 2 byte moves when possible to increase its speed.

_ss_wset, _gs_msel, _ss_umba, _ss_sbar

Usage:

```
#include <wind.h>

int _ss_wset(path,wintype,windat)
int path,wintype
WNDSCR *windat;

int _gs_msel(path,itemno)
int path,*itemno;

int _ss_umba(path)
int path;

int _ss_sbar(path,horbar,verbar)
int path,horvar,verbar;
```

Description:

_ss_wset() will set the current window type to **wintype**. **Wintype** values are defined in **<wind.h>** as:

```
/* window type defs */
#define WT_NBOX 0 /* No box- default window type */
#define WT_FWIN 1 /* Framed window with menus */
#define WT_FSWIN 2 /* Framed window with menus and scroll bars */
#define WT_SBOX 3 /* Shadowed window- form menus */
#define WT_DBOX 4 /* Double border- for dialog boxes */
#define WT_PBOX 5 /* Plain border- anything */
```

For framed windows, **windat** points to a menu structure. The menu structures are defined in **<wind.h>** as:

```
#define MNENBL 1
#define MNDSBL 0
#define WINSYNC 0xc0c0

/* default menu id's */
#define MN_MOVE 1
#define MN_CLOS 2
#define MN_GROW 3
#define MN_USCRL 4
#define MN_DSCRL 5
#define MN_RSCRL 6
#define MN_LSCRL 7
#define MN_TNDY 20
#define MN_FILE 21
#define MN_EDIT 22
#define MN_STYL 23
#define MN_FONT 24

/* window - menu data structures */
typedef struct mistr {  /* menu item descriptor */
char _mittl[15];        /* name of item */
char _mienbl;           /* is item available? */
char _mires[5];         /* reserved */
} MIDSCR;               /* item descriptor */

typedef struct mnstr {
char _mnttl[15];        /* name of menu */
char _mnid,             /* menu id number */
_mnxsiz,           /* width of menu */
_mnnits,           /* number of items */
_mnenabl;          /* is menu available? */
char _mnres[2];         /* reserved bytes */
struct mistr* _mnitems; /* pointer to items */
} MNDSCR;               /* menu descriptor */

typedef struct wnstr {  /* window descriptor */
char _wnttl[20];        /* title of window */
char _nmens;            /* number of menus on window */
char _wxmin,            /* min. window width */
_wymin;            /* min. window height */
short _wnsync;          /* synch bytes $C0C0 */
char _wnres[7];         /* reserved */
struct mnstr* _wnmen;  /* pointer to menu descriptor's array */
} WNDSCR;
```

Consult the programmer's guide in the *Multi-Vue* manual for details on framed window menus.

_gs_msel() will attempt to get a menu selection from the user. If the user makes a valid selection, then the item number is placed in the variable pointed to by **itemno** and the menu id is returned. Otherwise, 0 is returned.

_ss_umba() will update the window menus on **path**.

_ss_sbar() sets the scroll bar positions on windows using a framed window with scroll bars. The vertical scroll bar is set to **verbar** and the horizontal scroll bar is set to **horbar**.

Bugs:

There are many bugs in **WindInt** that affect these functions. The easiest to overcome is the **_ss_wset()** bug-**WindInt** forgets to erase the cursor before displaying the window borders, resulting in a garbled display. This can be overcome by turning the cursor off manually before calling **_ss_wset()**. Scroll bars also do not work properly on windows that do not lie on the lefthand side of the window. These bugs should be corrected in the next release of the operating system.

AddObj, MoveObj, DelObj

Usage:

```
#include <object.h>

OBJECT *Objects=NULL; /* global object list pointer */

OBJECT *AddObj(path,group,buffer,xcor,ycor,border)
int path,group,buffer,xcor,ycor,(*buffer)();

(void) MoveObj(path)
int path;

(void) DelObj(path,objptr)
int path;
OBJECT *objptr;
```

Description:

These functions provide simple sprite handling functions that use XOR logic. **AddObj()** will add an object to the list and place the object at its initial position. If **group** or **buffer** is equal to 0, then the **Point()** function is used in place of a **PutBlk()** call. **AddObj()** only sets the **xcor**, **ycor**, **group**, **buffer**, and **border** fields in the **OBJECT** structure (see below.) The **deltax**, **deltay**, **xaccel**, and **yaccel** are initialized to 0. The **OBJECT** structure is declared in **<object.h>** as:

```
typedef struct OBJSTR {
char group;          /* G/P group */
char buffer;         /* G/P buffer */
int xcor;            /* xcor * 32 */
int ycor;            /* ycor * 32 */
int deltax;          /* +/- xcor */
int deltay;          /* +/- ycor */
int xaccel;          /* xcor acceleration */
int yaccel;          /* ycor acceleration */
int (*border)();     /* boundary function */
struct OBJSTR *next; /* next object */
struct OBJSTR *prev; /* previous object */
} OBJECT;            /* call this type OBJECT */
```

Note that all coordinates and accelerations are fixed point numbers. That is, position 320 on the screen would be represented as 10240. This is done to provide smoother movement with higher precision at the lowest cost. The lower 5 bits of each number are the fractional portion of the number, giving an accuracy of 1/32. Higher accuracy could be achieved by changing the library code to use floats or longs.

MovObj() is the function that performs the movement of each object that has been added. First, the object is erased by re-putting the object at its present position (remember, we are using XOR logic.) Then, the border function is called to move the object. The border function is passed one argument-a pointer to the object that must be moved. The border function you specify can change any of the objects fields except for the **next** and **prev** fields. If the **border** field is NULL, then a bounce algorithm is used, where objects will bounce off the sides of the screen. The border function must return a 0 on success, or a -1 if the object should be deleted. If the border function is successful, the **MoveObj()** function puts the object at its new position.

The **DelObj()** function will delete the object pointed to by **objptr**. If path is equal to -1, then the object will not be erased from the screen. This is most useful from within the **MoveObj()** function, as the object will have already been erased. **DelObj()** also frees any memory used to hold the object in memory that the **AddObj()** function has allocated.

Examples:

Look in the main archive **CGFXLib.ar** to find the example program **Balls.c**

Play

Usage:

```
Play(path,play_string {,variable args})
int path;
char *play_string;
```

Description:

Play() is a C function which imitates the **PLAY** function of *Extended Color BASIC*. **Play_string** is a string of play 'commands' which can consist of:

A-G	The corresponding note is played in the current octave. The note letter can optionally be followed by a sequence of symbols to modify the note:	
	#,+	Indicates a sharp
	-	Indicates a flat
	<	Lowers the note an octave.
	>	Raises the note an octave.

number	Sets the length of the note to 'number.' (periods can be added to indicate dotted length.)
P{n}	This will add a rest (or pause) of the current note length. It can be followed optionally by a numeric length.
L{n}	This sets the current note length to the number following it.
V{n}	This sets the current volume to the number following it. The volume can be between 0 and 63.
O{n}	This sets the current octave to the number following it. The octave can be between 0 and 7.
MS	This makes notes play staccato.
MN	This makes notes play normally.
ML	This makes notes play legato.
T{n}	This sets the tempo to the number following it.

Additionally, the play string can contain substrings (indicated by **%s**) and integers (indicated by **%d**) for variable arguments, much as can be done with **printf()**.

PolyFill, PolyLine, PolyRot, PolyScal, PolyTran

Usage:

```
#include <polygon.h>

int PolyFill(path,polygon)
int path;
VERTEX *polygon;

int PolyLine(path,polygon)
int path;
VERTEX *polygon;

int PolyRot(polygon,cx,cy,angle)
VERTEX *polygon;
int cx,cy,angle;

int PolyScal(polygon,cx,cy,xmult,ymult,div)
VERTEX *polygon;
int cx,cy,xmult,ymult,div;

int PolyTran(polygon,xoff,yoff)
VERTEX *polygon;
int xoff,yoff;
```

Description:

These functions will draw closed polygons on path. Polygon is a pointer to an array of vertices containing the endpoints of the polygon. The first and last elements of this array must be equal; otherwise, these functions will search through all of memory until they finally come back to the original pointer. There is no limit (other than memory) to the

number of sides the polygon may have. The **VERTEX** type is defined in **<polygon.h>** as:

```
typedef struct {    /* polygon endpoint structure */
int p_xcor,p_ycor; /* x and y coordinates */
} VERTEX;
```

PolyFill() will fill the polygon, while **PolyLine()** only draws the polygon's outline.

PolyRot() will rotate a polygon about (**cx**,**cy**) angle degrees.

PolyScal() will scale a polygon from (**cx**,**cy**). **Xmult** and **ymult** are the x and y axis multipliers, respectively. The result from the multiplications is divided by **div**.

PolyTran() will translate (move) a polygon. **Xoff** and **yoff** are the x and y offsets to move each vertex of the polygon.

SetType

Usage:

```
SetType(path,stype,fg,bg)
int path,stype,fg,bg;
```

Description:

SetType() is a short function which will check the current window type of path and change it to type stype if necessary.

This function is dumb enough with text windows not to take advantage of the patch to **GRFDrv** to allow 25 lines. On return, the foreground and background colors will be set to **fg** and **bg** respectively. In the case where a new window is opened, the border is set to the background color.

Shadow

Usage:

```
Shadow(path,width,length,fg,bg)
int path,width,length,fg,bg;
```

Description:

Shadow() will create an overlay centered on the current window, with a foreground color of **fg** and a background color of **bg**. The overlay created will be **width** columns wide and **length** rows long.

Note: due to the way overlay coordinates are computed, multiple overlays may not appear centered on the window. This function calls **_Flush()** to flush any pending output before creating the overlay (this does not automatically include buffering code, however!)

_gs_scsz, _gs_palt, _gs_styp, _gs_fbrg, _ss_gip, _ss_dfpl, _ss_mtyp

Usage:

```
_gs_scsz(path,horsiz,versiz)
int path,*horsiz,*versiz;

_gs_palt(path,palbuf)
int path;
char *palbuf;

_gs_styp(path,type)
int path,*type;

_gs_fbrg(path,fore,back,bord)
int path,*fore,*back,*bord;

_ss_gip(path,msres,msport,kbdstrt,kbdrpt)
int path,msres,msport,kbdstrt,kbdrpt;

_ss_dfpl(path,palbuf)
int path;
char *palbuf;

_ss_mtyp(path,montype)
int path,montype;
```

Description:

_gs_scsz() will get the width and length of **path** and place it in the variables pointed to by **horsiz** and **versiz**. If the device in question does not support the **SS.ScSiz** call, then the function returns -1 (the values of **horsiz** and **versiz** are undefined.)

_gs_palt() will get the current palette register settings for **path** and place them in a 16 character buffer pointed to by **palbuf**.

_gs_styp() will get the window type of **path** and place it in the integer pointed to by **type**. -1 is returned if **path** is not a window device.

_gs_fbrg() will get the current foreground, background, and border colors for **path** and place them into the integers pointed to by **fore**, **back**, and **bord** respectively.

_ss_gip() will set the global information parameters. **Msres** is the mouse resolution. If it is 0, then a low resolution mouse is used. Otherwise, the high resolution mouse code is used. **msport** is the mouse port to use. 1 is the left port, and 2 is the right port. **kbdstrt** is the keyboard repeat start delay. If 0, keyboard repeat is turned off. Otherwise, the delay is set to **kbdstrt** ticks (60ths of a second.) **kbdrpt** is the keyboard repeat speed. If 0, keyboard repeat is turned off. Otherwise, the speed is set to one repeat every **kbdrpt** ticks. This function should not generally be called from user programs as it affects system-wide resources.

_ss_dfpl() sets the default palette settings for windows. **Palbuf** is a pointer to a 16 character palette buffer. This function should not generally be called from user programs as it affects system-wide resources. If a program needs different palette settings for its own window, then it should use the **Palette()** function.

_ss_mtyp() will set the current monitor type to **montype**, where 0 = composite monitor or television, 1 = analog RGB monitor, and 2 = monochrome composite monitor. This function should not generally be called from user programs as it affects system-wide resources.

_ss_tone

Usage:

```
int _ss_tone(path,duration,volume,frequency)
int path,duration,volume,frequency;
```

Description:

The **_ss_tone()** function provides a C interface to the **SS.Tone** setstat call on the Color Computer 3 running Level II OS-9. **Path** is any path to a window or VDG screen. **Duration** is the duration of the tone, measured in 1/60ths of seconds. **Volume** is the volume of the tone, where 0 is silence and 63 is maximum volume. **Frequency** is a number from 0 to 4095 representing the actual frequency (not equal to Hz!)

Due to an error in the system code, specifying a volume of 0 will force an immediate return, so rests (silence delays) must be done using sleep calls. **_ss_tone()** returns -1 on error, where the error code is placed in the global variable **errno**.

TandyMN

Usage:

```
int TandyMN(path,inum,fg,bg)
int path,inum,fg,bg;
```

Description:

TandyMN() is a function which will run the appropriate program selected from the default Tandy menu. **Inum** is the item number chosen from the Tandy menu. **Fg** and **bg** are the foreground and background colors for the overlay window created for the appropriate program. If **path** is less than 3, then the function creates an overlay on the current window. Otherwise, the function assumes that path points to a new window.

Bell, BlinkOff, BlinkOn, Clear, CrRtn, CurDwn, CurLft, CurOff, CurOn, CurRgt, CurUp, CurXY, DelLine, ErEOLine, ErEOScrn, ErLine, InsLin, ReVOff, ReVOn, UndlnOff, UndlnOn

Usage:

```
int Bell(path)
int path;

int BlnkOff(path)
int path;

int BlnkOn(path)
int path;

int Clear(path)
int path;

int CrRtn(path)
int path;

int CurDwn(path)
int path;

int CurLft(path)
int path;

int CurOff(path)
int path;

int CurOn(path)
int path;

int CurRgt(path)
```

```
int path;


int CurUp(path)
int path;

int CurXY(path,x,y)
int path,x,y;

int DelLine(path)
int path;

int ErEOLine(path)
int path;

int ErEOScrn(path)
int path;

int ErLine(path)
int path;

int InsLin(path)
int path;

int ReVOff(path)
int path;

int ReVOn(path)
int path;

int UndlnOff(path)
int path;

int UndlnOn(path)
int path;
```

Description:

All of these functions perform an operation on the cursor or window on path. For **CurXY**(), **x** and **y** are the column and row to position the cursor to. These coordinates are zero-based (the origin is 0,0.) The remaining functions do the following:

Bell()	sounds the bell.
BlnkOff()	turns blinking off (hardware text windows only.)
BlnkOn()	turns blinking on (hardware text windows only.)
Clear()	clears the screen and homes the cursor.
CrRtn()	moves the cursor to column zero.
CurDown()	moves the cursor down 1 row.
CurLft()	moves the cursor left 1 column.
CurOff()	turns the cursor off.
CurOn()	turns the cursor on.
CurRgt()	moves the cursor right 1 column.
CurUp()	moves the cursor up 1 row.
DelLine()	deletes the current line and moves the lines below the current one upward.

The last line is blanked.

ErEOLine() erases from the current column to the end of the line.

ErEOScrn() erases from the current column to the end of the screen.

ErLine() erases the current line.

InsLin() inserts a blank line at the current row. Lines below the current one are moved downward.

ReVOff() turns reverse video off.

ReVOn() turns reverse video on.

UndlnOff() turns underlining off.

UndlnOn() turns underlining on.

On error these functions return -1 with the error number in **errno**.

CWArea, DWEnd, DWProtSw, DWSet, MVOEnd, OWEnd, OWSet, Select, UnShadow

Usage:

```
int CWArea(path,cpx,cpy,szx,szy);
int path,cpx,cpy,szx,szy;

int DWEnd(path)
int path;

int DWProtSw(path,bsw)
int path,bsw;

int DWSet(path,sty,cpx,cpy,szx,szy,fprn,bprn,bdprn)
int path,sty,cpx,cpy,szx,szy,fprn,bprn,bdprn;

int DWSet(path,sty=0,cpx,cpy,szx,szy,fprn,bprn)
int path,sty,cpx,cpy,szx,szy,fprn,bprn;

int MVOWEnd(path)
int path;

int OWEnd(path)
int path;

int OWSet(path,svs,cpx,cpy,szx,szy,fprn,bprn)
int path,svs,cpx,cpy,szx,szy,fprn,bprn;

int Select(path)
int path;

int UnShadow(path)
int path;
```

Description:

DWSet() will create a new window on **path** of type **sty**. If **sty** is 0, then the window is created on the process's current window. The valid values for sty are:

STY	Window Size	Colors	Memory	Type
	current window			
1	40x24(25*)	8&8	2000 bytes	Text
2	80x24(25*)	8&8	4000 bytes	Text
5	80x24	2	16000 bytes	Graphics
6	40x24	4	16000 bytes	Graphics
7	80x24	4	32000 bytes	Graphics
8	40x24	16	32000 bytes	Graphics

DWEnd() will remove a window on **path**.

DWProtSw() will allow new windows to be created over the window on **path** if **bsw** is 1. Otherwise, it will disallow any new windows to be created over the window.

OWSet() will create an overlay window over the window on **path**. If **svs** is 1, then the area under the overlay is saved and the overlay area is cleared. Otherwise, the overlay area is left untouched.

OWEnd() and **UnShadow()** will remove a previously created overlay. If the overlay was created with the **svs** flag set to 1, then the area under the overlay is restored.

MVOWEnd() will reset the current *Multi-Vue* window type to the 'no box' and remove a previously created overlay. If the overlay was created with the **svs** flag set to 1, then the area under the overlay is restored.

CWArea() will change the current window/overlay working area to that specified in the parameter list. The coordinates specified are relative to the window or overlay that is active.

Select() will select the window on **path** to be displayed. If the calling process does not own the keyboard (is not the current active process), then the call is ignored.

In all cases, **path** refers to the path to the window in question, **cpx** refers to the upper left-hand corner column position, **cpy** refers to the upper left-hand corner row position, **szx** refers to the character column width, **szy** refers to the character row height, **fprn** refers to the foreground color, **bprn** refers to the background color, and **bdprn** refers to the border color. These functions call **_Flush()** to flush any pending output before proceeding (this does not automatically include buffering, however!) On error, -1 is returned. The error code may be found in the **errno** variable.