LIBGRX

A 16/256 color graphics library for DJGPP

User's Manual

Written by:
Csaba Biegl

August 10, 1992

Abstract

LIBGRX is a graphics library for DJGPP or Turbo C programs. (The Turbo C version was tested using TC++ 1.01) Currently it supports VGA (32768, 256 or 16 colors), EGA (16 colors), 8514/A (256 colors) and S3-based (256 colors) cards. Planned future improvements include support for Hercules cards as well. It is upward compatible with the graphics library in DJGPP (the C part, no C++ classes). It can use the same drivers as the graphics library in DJGPP or it can work with a new driver format supporting programmable number of colors as well.

Data types, function declarations

All public data structures and graphics primitives meant for usage by the application program are declared/prototyped in the header files (in the 'include' sub-directory):

GRDRIVER.H	graphics driver format specifications
MOUSEX.H	cursor, mouse and keyboard handling
GRX.H	drawing-related structures and functions
GRXFILE.H	file formats and file access routines
GRXFONT.H	format of a font when loaded into memory

Setting video modes

Before a program can do any graphics drawing it has to configure the display adapter of the PC for the desired graphics mode. It is done with the 'GrSetMode' function as follows:

```
#ifdef __cplusplus
  void  GrSetMode(int which,int width=0,int height=0,int colors=0);
#else
  void  GrSetMode(int which,...);
#endif
```

The 'mode' parameter can be one of the following constants, declared in "grx.h":

```
typedef enum {
    GR_80_25_text,
    GR_default_text,
    GR_width_height_text,
    GR_biggest_text,
    GR_320_200_graphics,
    GR_default_graphics,
    GR_width_height_graphics,
    GR_biggest_noninterlaced_graphics,
    GR_biggest_graphics,
    GR_width_height_color_graphics
} GR_graphics_modes;
```

The 'GR_width_height_text' and 'GR_width_height_graphics' modes require the two optional size arguments, and the 'GR_width_height_color_graphics' mode requires all three optional arguments. A call with any other mode does not require any of the optional arguments.

NOTE: the 'GR_width_height_color_graphics' mode is a new mode supported only if: (1) you have a new format graphics driver (see the accompanying documentation in the file "DRIVERS.DOC"), and (2) you have a recent (version 1.06, dated after the middle of April 1992) copy of GO32 which knows how to deal with the new driver format.

New format graphics drivers operating in the proper environment (see above) can provide a table of the supported text and graphics modes using:

```
void  GrGetDriverModes(GR_DRIVER_MODE_ENTRY **ttable,GR_DRIVER_MODE_ENTRY **gtable)
```

The arguments 'ttable' and 'gtable' should be addresses of pointers to the 'GR_DRIVER_MODE_ENTRY' structure defined in the include file "grdriver.h". Upon return the pointer variables will point to two arrays of these structures, one for the text modes, the other for graphics. The 'GR_DRIVER_MODE_ENTRY' structure has the following fields:

```
typedef struct {
    unsigned short  width;
    unsigned short  height;
    unsigned short  number_of_colors;
    unsigned char   BIOS_mode;
    unsigned char   special;
} GR_DRIVER_MODE_ENTRY;
```

The first four structure members should be obvious, but the 'special' field may deserve some explanation. It is non-zero if the driver does some special "tricks" to put the card into the desired mode. An example might be the 43 row EGA text mode: for this first the "standard" 80x25 text mode is set up and then the character generator is re-loaded with a smaller font. If the 'special' field is zero then the driver simply invokes the INT 10 function 0 BIOS routine with the mode in the 'BIOS_mode' slot.

A user-defined function can be invoked every time the video mode is changed (i.e. 'GrSetMode' is called). This function should not take any parameters and its return value (if any) is ignored. It can be installed (for all subsequent 'GrSetMode' calls) with the:

```
void    GrSetModeHook(void (*callback)(void));
```

function. The current graphics mode (one of the valid 'mode' argument values for 'GrSetMode') can be obtained with the:

```
int     GrCurrentMode(void);
```

function, while the type of the installed graphics adapter can be determined with the

```
int     GrAdapterType(void);
```

function. 'GrAdapterType' returns the type of the adapter as one of the following three symbolic constants (defined in "grx.h"):

```
#define GR_VGA          0               /* VGA adapter */
#define GR_EGA          1               /* EGA adapter */
#define GR_HERC         2               /* Hercules mono adapter */
#define GR_8514A        3               /* 8514/A or compatible */
#define GR_S3           4               /* S3 graphics accelerator */
```

Graphics contexts

The library supports a set of drawing regions called 'contexts' (the 'GrContext' structure). These can be in video memory or in system memory. Contexts in system memory always have the same memory organization as the video memory. When 'GrSetMode' is called, a default context is created which maps to the whole graphics screen. Contexts are described by the 'GrContext' data structure:

```
typedef struct _GR_context_ {
    char  far *gc_baseaddr;           /* base address of display memory */
    long  gc_frameaddr;               /* upper left corner coordinate */
    long  gc_planeoffset;             /* offset to next color plane */
    int   gc_lineoffset;              /* offset to next scan line in bytes */
    char  gc_onscreen;                /* is it in video memory ? */
    char  gc_memflags;                /* memory allocation flags */
    int   gc_xmax;                    /* max X coord (width  - 1) */
    int   gc_ymax;                    /* max Y coord (height - 1) */
    int   gc_xcliplo;                 /* low X clipping limit */
    int   gc_ycliplo;                 /* low Y clipping limit */
    int   gc_xcliphi;                 /* high X clipping limit */
    int   gc_ycliphi;                 /* high Y clipping limit */
    int   gc_usrxbase;                /* user window min X coordinate */
    int   gc_usrybase;                /* user window min Y coordinate */
    int   gc_usrwidth;                /* user window width */
    int   gc_usrheight;               /* user window height */
    int   gc_xoffset;                 /* X offset from root's base */
    int   gc_yoffset;                 /* Y offset from root's base */
    struct _GR_context_ *gc_root;     /* context which owns frame buf */
} GrContext;
```

There is a subtype of the 'GrContext' structure. The structure 'GrVidRAM' contains only the first slots of a context describing the memory layout of a context. This structure is used as a component of other more complex data structures in the library.

```
typedef struct {
    char  far *gc_baseaddr;           /* base address of display memory */
    long  gc_frameaddr;               /* upper left corner coordinate */
    long  gc_planeoffset;             /* offset to next color plane */
    int   gc_lineoffset;              /* offset to next scan line in bytes */
    char  gc_onscreen;                /* is it in video memory ? */
    char  gc_memflags;                /* memory allocation flags */
} GrVidRAM;
```

The following four functions return information about the layout of and memory occupied by a graphics context of size 'width' by 'height' in the current graphics mode (as set up by 'GrSetMode'):

```
int     GrLineOffset(int width);
int     GrNumPlanes(void);
long    GrPlaneSize(int w,int h);
long    GrContextSize(int w,int h);
```

'GrLineOffset' always returns the offset between successive pixel rows of the context in **bytes**. 'GrNumPlanes' returns the number of bitmap planes in the current graphics mode. 'GrContextSize' calculates the total amount of memory needed by a context, while 'GrPlaneSize' calculates the size of a bitplane in the context. The function:

```
GrContext *GrCreateContext(int w,int h,char far *memory,GrContext *where);
```

can be used to create a new context in system memory. The NULL pointer is also accepted as the value of the 'memory' and 'where' arguments, in this case the library allocates the necessary amount of memory internally.

It is a general convention in the library that functions returning pointers to any LIBGRX specific data structure have a last argument (most of the time named 'where' in the prototypes) which can be used to pass the address of the data structure which should be filled with the result. If this 'where' pointer has the value of NULL, then the library allocates space for the data structure internally.

The function:

```
GrContext *GrCreateSubContext(int x1,int y1,int x2,int y2,GrContext *parent,GrContext *where);
```

creates a new sub-context which maps to a part of an existing context. The coordinate arguments ('x1' through 'y2') are interpreted relative to the parent context's limits. Pixel addressing is zero-based even in sub-contexts, i.e. the address of the top left pixel is (0,0) even in a sub-context which has been mapped onto the interior of its parent context.

Sub-contexts can be resized, but not their parents (i.e. anything returned by 'GrCreateContext' or set up by 'GrSetMode' cannot be resized -- because this could lead to irrecoverable "loss" of drawing memory. The following function can be used for this purpose:

```
void    GrResizeSubContext(GrContext *context,int x1,int y1,int x2,int y2);
```

The current context structure is stored in a static location in the library. (For efficiency reasons -- it is used quite frequently, and this way no pointer dereferencing is necessary.) The context stores all relevant information about the video organization, coordinate limits, etc... The current context can be set with the:

```
void    GrSetContext(GrContext *context);
```

function. This function will reset the current context to the full graphics screen if it is passed the NULL pointer as argument. The value of the current context can be saved into a 'GrContext' structure pointed to by 'where' using:

```
GrContext *GrSaveContext(GrContext *where);
```

(Again, if 'where' is NULL, the library allocates the space.) Contexts can be destroyed with:

```
void    GrDestroyContext(GrContext *context);
```

This function will free the memory occupied by the context only if it was allocated originally by the library. The next three functions set up and query the clipping limits associated with the context:

```
void    GrSetClipBox(int x1,int y1,int x2,int y2);
void    GrGetClipBox(int *x1p,int *y1p,int *x2p,int *y2p);
void    GrResetClipBox(void);
```

'GrResetClipBox' sets the clipping limits to the limits of context. These are the limits set up initially when a context is created. The limits of the current context can be obtained using the following functions:

```
int     GrMaxX(void);
int     GrMaxY(void);
int     GrSizeX(void);
int     GrSizeY(void);
```

The 'Max' functions return the biggest valid coordinate, while the 'Size' functions return a value one higher. The limits of the graphics screen (regardless of the current context) can be obtained with:

```
int     GrScreenX(void);
int     GrScreenY(void);
```

Color management

The library supports two models for color management. In the 'indirect' (or color table) mode colors can be allocated with the highest resolution supported by the hardware (EGA: 2 bits, VGA: 6 bits) with respect to the component color intensities. In the 'direct' or RGB mode color indices map directly into component color intensities with non-overlapping bitfields of the color index representing the component colors. The RGB mode is supported in 256 color and 32768 color VGA modes only (for 32768 colors this is the only mode because of the limitations of the VGA hardware with HiColor DAC). In RGB mode the color index maps to component color intensities as follows:

256:	**rrrgggbb**	(3 bits for red and green, 2 for blue)
32768:	**xrrrrrgggggbbbbb**	(5 bits for each component color)

The RGB mode is not supported in 16 color EGA and VGA modes as there are too few available colors to provide adequate coverage. The advantage of the RGB mode is faster color allocation (no table lookup, DAC register programming, etc...) its disadvantage is the relatively crude approximation of the component color intensities.

After the first 'GrSetMode' call two colors are always defined: **black** and **white**. The indices of these two colors are returned by the functions:

```
int     GrBlack(void);
int     GrWhite(void);
```

NOTE: unlike the original DJGPP library, LIBGRX does not guarantee that the white color has the color index value of one. (GrBlack still returns 0).

The library supports four write modes: write, XOR, logical OR and logical AND. These can be selected with OR-ing the color index with one of the following constants declared in "grx.h":

```
#ifdef __TURBOC__
# define GrXOR          0x100           /* to "XOR" any color to the screen */
# define GrOR           0x200           /* to "OR" to the screen */
# define GrAND          0x300           /* to "AND" to the screen */
#endif
#ifdef __GNUC__                         /* changed for 16 bit colors */
# define GrXOR          0x10000         /* to "XOR" any color to the screen */
# define GrOR           0x20000         /* to "OR" to the screen */
# define GrAND          0x30000         /* to "AND" to the screen */
#endif
#define GrWRITE         0               /* write color */
#define GrNOCOLOR       (GrXOR | 0)     /* GrNOCOLOR is used for "no" color */
```

NOTE: 'GrXOR' was declared to be "0x100" in the original DJGPP library, but its value had to be changed in LIBGRX in anticipation of the 32768 color VGA support.

By convention, the no-op color is obtained by combining color index 0 (black) with the XOR operation. This no-op color has been defined in "grx.h" as 'GrNOCOLOR'.

The number of colors in the current graphics mode is returned by the:

```
int     GrNumColors(void);
```

function, while the number of unused, available color can be obtained by calling:

```
int     GrNumFreeColors(void);
```

Colors can be allocated with the:

```
int     GrAllocColor(int r,int g,int b);
```

function (component intensities can range from 0 to 255), or with the:

```
int     GrAllocCell(void);
```

function. In the second case the component intensities of the returned color can be set with:

```
void    GrSetColor(int color,int r,int g,int b);
```

Both 'Alloc' functions return 'GrNOCOLOR' if there are no more free colors available. Additionally 'GrAllocCell' always returns 'GrNOCOLOR' when the color system is in RGB mode, as colors returned by 'GrAllocCell' are meant to be changed -- what is not supposed to be done in RGB mode. Also note that 'GrAllocColor' operates much more efficiently in RGB mode, and that it never returns 'GrNOCOLOR' in this case.

Color table entries can be freed (when not in RGB mode) by calling:

```
void    GrFreeColor(int color);
```

The component intensities of any color can be queried using the function:

```
void    GrQueryColor(int c,int *r,int *g,int *b);
```

Initially the color system is in color table (indirect) mode. (Except for the 32768 color VGA modes which are always in RGB mode.) 256 color VGA modes can be put into the RGB mode by calling:

```
void    GrSetRGBcolorMode(void);
```

The color system can be reset (i.e. put back into color table mode if possible, all colors freed except for black and white) by calling:

```
void    GrResetColors(void);
```

The function:

```
void    GrRefreshColors(void);
```

reloads the currently allocated color values into the video hardware. This function is not needed in typical applications, unless the display adapter is programmed directly by the application.

Graphics primitives

The screen, the current context or the current clip box can be cleared (i.e. set to a desired background color) by using one of the following three functions:

```
void    GrClearScreen(int bg);
void    GrClearContext(int bg);
void    GrClearClipBox(int bg);
```

The following line drawing graphics primitives are supported by the library:

```
void    GrPlot(int x,int y,int c);
void    GrLine(int x1,int y1,int x2,int y2,int c);
void    GrHLine(int x1,int x2,int y,int c);
void    GrVLine(int x,int y1,int y2,int c);
void    GrBox(int x1,int y1,int x2,int y2,int c);
void    GrCircle(int xc,int yc,int r,int c);
```

```
void    GrEllipse(int xc,int yc,int xa,int ya,int c);
void    GrCircleArc(int xc,int yc,int r,int start,int end,int c);
void    GrEllipseArc(int xc,int yc,int xa,int ya,int start,int end,int c);
void    GrPolyLine(int numpts,int points[][2],int c);
void    GrPolygon(int numpts,int points[][2],int c);
```

All primitives operate on the current graphics context. The last argument of these functions is always the color index to use for the drawing. The 'HLine' and 'VLine' primitives are for drawing horizontal and vertical lines. They have been included in the library because they are more efficient than the general line drawing provided by 'GrLine'. The ellipse primitives can only draw ellipses with their major axis parallel with either the X or Y coordinate axis. They take the half X and Y axis length in the 'xa' and 'ya' arguments. The arc (circle and ellipse) drawing functions take the start and end angles in tenths of degrees (i.e. meaningful range: 0 ... 3600). The angles are interpreted counter-clockwise starting from the positive X axis. The polyline and polygon primitives take the address of an n by 2 coordinate array. The X values should be stored in the elements with 0 second index, and the Y values in the elements with a second index value of 1. Coordinate arrays passed to the polygon primitives can either contain or omit the closing edge of the polygon -- the primitive will append it to the list if it is missing.

The following filled primitives are available:

```
void    GrFilledBox(int x1,int y1,int x2,int y2,int c);
void    GrFramedBox(int x1,int y1,int x2,int y2,int wdt,GrFBoxColors *c);
void    GrFilledCircle(int xc,int yc,int r,int c);
void    GrFilledEllipse(int xc,int yc,int xa,int ya,int c);
void    GrFilledCircleArc(int xc,int yc,int r,int start,int end,int c);
void    GrFilledEllipseArc(int xc,int yc,int xa,int ya,int start,int end,int c);
void    GrFilledPolygon(int numpts,int points[][2],int c);
void    GrFilledConvexPolygon(int numpts,int points[][2],int c);
```

Similarly to the line drawing, all of the above primitives operate on the current graphics context. The 'GrFramedBox' primitive can be used to draw **motif**-like shaded boxes and "ordinary" framed boxes as well. The 'x1' through 'y2' coordinates specify the **interior** of the box, the border is outside this area. The primitive uses five different colors for the interior and four borders of the box which are specified in the 'GrFBoxColors' structure:

```
typedef struct {
    int  fbx_intcolor;
    int  fbx_topcolor;
    int  fbx_rightcolor;
    int  fbx_bottomcolor;
    int  fbx_leftcolor;
} GrFBoxColors;
```

The 'GrFilledConvexPolygon' primitive can be used to fill convex polygons. It can also be used to fill some concave polygons whose boundaries do not intersect any horizontal scan line more than twice. All other concave polygons have to be filled with the (somewhat less efficient) 'GrFilledPolygon' primitive. This primitive can also be used to fill several disjoint non-overlapping polygons in a single operation.

The current color value of any pixel in the current context can be obtained with:

```
int     GrPixel(int x,int y);
```

Rectangular areas can be transferred within a context or between contexts by calling:

```
void    GrBitBlt(GrContext *dest,int x,int y,GrContext *source,int x1,int y1,int x2,int y2,int oper);
```

The 'oper' argument should be one of supported color modes (write, XOR, OR, AND), it will control how the pixels from the source context are combined with the pixels in the destination context. If either the source or the destination context argument is the NULL pointer then the current context is used for that argument.

Non-clipping graphics primitives

There is a non-clipping version of some of the elementary primitives. These are somewhat more efficient than the regular versions. **These are to be used only in situations when it is absolutely certain that no drawing will be performed beyond the boundaries of the current context. Otherwise the program will almost certainly crash!** The reason for including these functions is that they are somewhat more efficient than the regular, clipping versions. **ALSO NOTE:** These function **do not** check for conflicts with the mouse cursor. (See the explanation about the mouse cursor handling later in this document.) The list of the supported non-clipping primitives:

```
void    GrPlotNC(int x,int y,int c);
void    GrLineNC(int x1,int y1,int x2,int y2,int c);
void    GrHLineNC(int x1,int x2,int y,int c);
void    GrVLineNC(int x,int y1,int y2,int c);
void    GrBoxNC(int x1,int y1,int x2,int y2,int c);
void    GrFilledBoxNC(int x1,int y1,int x2,int y2,int c);
void    GrFramedBoxNC(int x1,int y1,int x2,int y2,int wdt,GrFBoxColors *c);
void    GrBitBltNC(GrContext *dst,int x,int y,GrContext *src,int x1,int y1,int x2,int y2,int oper);
int     GrPixelNC(int x,int y);
```

Customized line drawing

The basic line drawing graphics primitives described previously always draw continuous lines which are one pixel wide. There is another group of line drawing functions which can be used to draw wide and/or patterned lines. These functions have similar parameter passing conventions as the basic ones with one difference: instead of the color value a pointer to a structure of type 'GrLineOption' has to be passed to them. The definition of the 'GrLineOption' structure:

```
typedef struct {
    int  lno_color;                    /* color used to draw line */
    int  lno_width;                    /* width of the line */
    int  lno_pattlen;                  /* length of the dash pattern */
    unsigned char *lno_dashpat;        /* draw/nodraw pattern */
} GrLineOption;
```

The 'lno_pattlen' structure element should be equal to the number of alternating draw -- no draw section length values in the array pointed to by the 'lno_dashpat' element. The dash pattern array is assumed to begin with a drawn section. If the pattern length is equal to zero a continuous line is drawn. The available custom line drawing primitives:

```
void    GrCustomLine(int x1,int y1,int x2,int y2,GrLineOption *o);
void    GrCustomBox(int x1,int y1,int x2,int y2,GrLineOption *o);
void    GrCustomCircle(int xc,int yc,int r,GrLineOption *o);
void    GrCustomEllipse(int xc,int yc,int xa,int ya,GrLineOption *o);
void    GrCustomCircleArc(int xc,int yc,int r,int start,int end,GrLineOption *o);
void    GrCustomEllipseArc(int xc,int yc,int xa,int ya,int start,int end,GrLineOption *o);
void    GrCustomPolyLine(int numpts,int points[][2],GrLineOption *o);
void    GrCustomPolygon(int numpts,int points[][2],GrLineOption *o);
```

Pattern filled graphics primitives

The library also supports a pattern filled version of the basic filled primitives described above. These functions have similar parameter passing conventions as the basic ones with one difference: instead of the color value a pointer to an union of type 'GrPattern' has to be passed to them. The 'GrPattern' union can contain either a bitmap or a pixmap fill pattern. The first integer slot in the union determines which type it is. Bitmap fill patterns are rectangular arrays of bits, each set bit representing the foreground color of the fill operation, and each zero bit representing the background. Both the foreground and background colors can be combined with any of the supported logical operations. Bitmap fill patterns have one restriction: their width must be eight pixels. Pixmap fill patterns are very similar to contexts, the data is transferred to the filled primitive using 'BitBlt' operations. The relevant structure declarations (from "grx.h"):

```
/*
 * BITMAP: a mode independent way to specify a fill pattern of two
 *   colors. It is always 8 pixels wide (1 byte per scan line), its
 *   height is user-defined. SET THE TYPE FLAG TO ZERO!!!
 */
typedef struct {
    int  bmp_ispixmap;                  /* type flag for pattern union */
    int  bmp_height;                    /* bitmap height */
    unsigned char *bmp_data;            /* pointer to the bit pattern */
    int  bmp_fgcolor;                   /* foreground color for fill */
    int  bmp_bgcolor;                   /* background color for fill */
    int  bmp_memflags;                  /* set if dynamically allocated */
} GrBitmap;

/*
 * PIXMAP: a fill pattern stored in a layout identical to the video RAM
 *   for filling using 'bitblt'-s. It is mode dependent, typically one
 *   of the library functions is used to build it. KEEP THE TYPE FLAG
 *   NONZERO!!!
 */
typedef struct {
    int  pxp_ispixmap;                  /* type flag for pattern union */
    int  pxp_width;                     /* pixmap width (in pixels)  */
    int  pxp_height;                    /* pixmap height (in pixels) */
    int  pxp_oper;                      /* bitblt mode (SET, OR, XOR, AND) */
    GrVidRAM pxp_source;                /* source context for fill */
} GrPixmap;

/*
 * Fill pattern union -- can either be a bitmap or a pixmap
 */
typedef union {
    int      gp_ispixmap;               /* nonzero for pixmaps */
    GrBitmap gp_bitmap;                 /* fill bitmap */
    GrPixmap gp_pixmap;                 /* fill pixmap */
} GrPattern;

#define gp_bmp_data                     gp_bitmap.bmp_data
#define gp_bmp_height                   gp_bitmap.bmp_height
#define gp_bmp_fgcolor                  gp_bitmap.bmp_fgcolor
#define gp_bmp_bgcolor                  gp_bitmap.bmp_bgcolor

#define gp_pxp_width                    gp_pixmap.pxp_width
#define gp_pxp_height                   gp_pixmap.pxp_height
#define gp_pxp_oper                     gp_pixmap.pxp_oper
#define gp_pxp_source                   gp_pixmap.pxp_source
```

Bitmap patterns can be easily built from initialized character arrays and static structures by the C compiler, thus no special support is included in the library for creating them. The only action required from the application program might be changing the foreground and background colors as needed. Pixmap patterns are more difficult to build as they replicate the layout of the video memory which changes for different video modes. For this reason the library provides three functions to create pixmap patterns in a mode-independent way:

```
GrPattern *GrBuildPixmap(char *pixels,int w,int h,GrColorTableP colors);
GrPattern *GrBuildPixmapFromBits(char *bits,int w,int h,int fgc,int bgc);
GrPattern *GrConvertToPixmap(GrContext *src);
```

'GrBuildPixmap' build a pixmap from a two dimensional ('w' by 'h') array of characters. The elements in this array are used as indices into the color table specified with the argument 'colors'. (This means that pixmaps created this way can use at most 256 colors.) The color table pointer:

```
typedef int *GrColorTableP;
```

should point to an array of integers with the first element being the number of colors in the table and the color indices themselves starting with the second element. **NOTE:** any color modifiers (GrXOR, GrOR, GrAND) OR-ed to the elements of the color table are ignored.

The 'GrBuildPixmapFromBits' function builds a pixmap fill pattern from bitmap data. It is useful if the width of the bitmap pattern is not eight as such bitmap patterns can not be used to build a 'GrBitmap' structure.

The 'GrConvertToPixmap' function converts a graphics context to a pixmap fill pattern. It is useful when the pattern can be created with graphics drawing operations. **NOTE:** the pixmap pattern and the original context share the drawing RAM, thus if the context is redrawn the fill pattern changes as well.

Fill patterns which were built by library routines can be destroyed when no longer needed (i.e. the space occupied by them can be freed) by calling:

```
void    GrDestroyPattern(GrPattern *p);
```

NOTE: when pixmap fill patterns converted from contexts are destroyed, the drawing RAM is not freed. It is freed when the original context is destroyed. Fill patterns built by the application have to be destroyed by the application as well (if this is needed).

The list of supported pattern filled graphics primitives is shown below. These functions are very similar to their solid filled counterparts, only their last argument is different:

```
void    GrPatternFilledPlot(int x,int y,GrPattern *p);
void    GrPatternFilledLine(int x1,int y1,int x2,int y2,GrPattern *p);
void    GrPatternFilledBox(int x1,int y1,int x2,int y2,GrPattern *p);
void    GrPatternFilledCircle(int xc,int yc,int r,GrPattern *p);
void    GrPatternFilledEllipse(int xc,int yc,int xa,int ya,GrPattern *p);
void    GrPatternFilledCircleArc(int xc,int yc,int r,int start,int end,GrPattern *p);
void    GrPatternFilledEllipseArc(int xc,int yc,int xa,int ya,int start,int end,GrPattern *p);
void    GrPatternFilledConvexPolygon(int numpts,int points[][2],GrPattern *p);
void    GrPatternFilledPolygon(int numpts,int points[][2],GrPattern *p);
```

Strictly speaking the plot and line functions in the above group are not filled, but they have been included here for convenience.

Patterned line drawing

The custom line drawing functions introduced above also have a version when the drawn sections can be filled with a (pixmap or bitmap) fill pattern. To achieve this these functions must be passed both a custom line drawing option ('GrLineOption' structure) and a fill pattern ('GrPattern' union). These two have been combined into the 'GrLinePattern' structure:

```
typedef struct {
    GrPattern     *lnp_pattern;         /* fill pattern */
    GrLineOption  *lnp_option;          /* width + dash pattern */
} GrLinePattern;
```

All patterned line drawing functions take a pointer to this structure as their last argument. The list of available functions:

```
void    GrPatternedLine(int x1,int y1,int x2,int y2,GrLinePattern *lp);
void    GrPatternedBox(int x1,int y1,int x2,int y2,GrLinePattern *lp);
void    GrPatternedCircle(int xc,int yc,int r,GrLinePattern *lp);
void    GrPatternedEllipse(int xc,int yc,int xa,int ya,GrLinePattern *lp);
void    GrPatternedCircleArc(int xc,int yc,int r,int start,int end,GrLinePattern *lp);
void    GrPatternedEllipseArc(int xc,int yc,int xa,int ya,int start,int end,GrLinePattern *lp);
void    GrPatternedPolyLine(int numpts,int points[][2],GrLinePattern *lp);
void    GrPatternedPolygon(int numpts,int points[][2],GrLinePattern *lp);
```

Text drawing

The library supports loadable bit-mapped (i.e. not scalable!) fonts. Some of these fonts were converted from VGA BIOS fonts and fonts on utility diskettes coming with VGA cards. These fonts have all 256 characters. Some additional fonts were converted from fonts in the MIT X11 distribution. These have a variable number of characters, some support all 256 character codes, some only the printable ASCII codes. Fonts also have family names, which are used in a font lookup procedure supported by the library (see later). The following font families are included in the distribution:

Font file name:	Family:	Description:
pc<W>x<H>[t].fnt	pc	BIOS font, fixed
xm<W>x<H>[b][i].fnt	X_misc	X11, fixed, miscellaneous group
char<H>[b][i].fnt	char	X11, proportional, charter family
cour<H>[b][i].fnt	cour	X11, fixed, courier
helve<H>[b][i].fnt	helve	X11, proportional, helvetica
lucb<H>[b][i].fnt	lucb	X11, proportional, lucida bright
lucs<H>[b][i].fnt	lucs	X11, proportional, lucida sans serif
luct<H>[b][i].fnt	luct	X11, fixed, lucida typewriter
ncen<H>[b][i].fnt	ncen	X11, proportional, new century schoolbook
symb<H>.fnt	symbol	X11, proportional, greek letters, symbols
tms<H>[b][i].fnt	times	X11, proportional, times

In the font names <W> means the font width, <H> the font height. Many font families have bold and/or italic variants. The files containing these fonts contain a 'b' and/or 'i' character in their name just before the extension. Additionally, the strings "_bold" and/or "_ital" are appended to the font family names. Some of the pc BIOS fonts come in thin formats also, these are denoted by a 't' in their file names and the string "_thin" in their family names.

NOTE: the basic libgrx distribution ("cbgrx101.zip") contains only the full "pc", "courier", "helve", "symbol" and "times" font families and selected sizes from the "X_misc" family. (Because of archive size considerations!) The full compliment of fonts can be found in the archive file named "cbgrxfnt.zip", which should be available from the same site where the basic library was obtained from.

Fonts are loaded with the 'GrLoadFont' function. If the font file name starts with any path separator character or character sequence ('<drive letter>:', '/' or '\') then it is loaded from the specified directory, otherwise the font is loaded from the default font path. The font path can be set up with the 'GrSetFontPath' function. If the font path is not set then the value of the 'GRXFONT' environment variable is used as the font path. If 'GrLoadFont' is called again with the name of an already loaded font then it will return a pointer to the result of the first loading. The special font names "@:pc8x8.fnt", "@:pc8x14.fnt" and "@:pc8x16.fnt" will cause 'GrLoadFont' to load the font from the BIOS of the graphics card of the PC (provided that it has the desired font). Alternatively, 'GrLoadBIOSFont' can also be called to load a font which resides in the BIOS of the display adapter. (The difference is that 'GrLoadFont' will look at the disk as well if the BIOS does not have the font. For example: EGA-s don't have a 16 row font in their BIOS.) Both font loading routines return NULL if the font was not found. When not needed any more, fonts can be unloaded (i.e. the storage occupied by them freed) by calling 'GrUnloadFont'. The prototype declarations for these functions:

```
GrFont *GrLoadFont(char *name);
GrFont *GrLoadBIOSFont(char *name);
void    GrUnloadFont(GrFont *font);
void    GrSetFontPath(char *path);
```

The 'GrFont' structure is actually a font header, the real font data is right next to this header in memory, but it is typically hidden from the application program. If needed, the include file "grxfont.h" can provide more details. The 'GrFont' structure:

```
typedef struct {
    short   fnt_width;                  /* width (average for proportional) */
    short   fnt_height;                 /* font height */
    short   fnt_minchar;                /* lowest character code in font */
    short   fnt_maxchar;                /* highest character code in font */
    short   fnt_isfixed;                /* nonzero if fixed font */
    short   fnt_internal;               /* nonzero if BIOS font */
    short   fnt_baseline;               /* baseline from top of font */
    short   fnt_undwidth;               /* underline width (at bottom) */
    char    fnt_name[GR_NAMEWIDTH];     /* font file name (w/o path) */
    char    fnt_family[GR_NAMEWIDTH];   /* font family name */
} GrFont;
```

There is a function: 'GrFindBestFont' which returns the font which matches best a desired size. (Best match: not bigger, but as close as possible). The application can specify whether it wants 'GrFindBestFont' to find the best match using fonts in their original sizes only, or possibly enlarged (with the value of the 'magnify' argument -- 0: no, nonzero: yes). 'GrFindBestFont' also takes a string argument which specifies the font family from which to select the font. The string can specify several family patterns separated by the ':' character. Each pattern can contain the '?' and '*' wildcard characters which work the usual way (in UNIX sense -- i.e. "X*ital" will match "X_misc_ital", but not "X_misc_bold"..).

```
GrTextOption *GrFindBestFont(int width,int height,int magnify,char *family,GrTextOption *where);
```

The 'GrTextOption' structure specifies how to draw a character string:

```
typedef struct {
    GrFont *txo_font;                   /* font to be used */
    int     txo_xmag;                   /* X magnification */
    int     txo_ymag;                   /* Y magnification */
    union {
        int v;                          /* color when no attributes */
        GrColorTableP p;                /* ptr to color table otherwise */
    } txo_fgcolor,txo_bgcolor;          /* foreground, background */
    char    txo_direct;                 /* direction */
    char    txo_xalign;                 /* X alignment */
    char    txo_yalign;                 /* Y alignment */
    char    txo_chrtype;                /* character type */
} GrTextOption;
```

The font can be enlarged independently in the X and Y directions, ('txo_xmag' and 'txo_ymag' slots -- values: 1 and up) the text can be rotated in increments of 90 degrees ('txo_direct'), alignments can be set in both directions ('txo_xalign' and 'txo_yalign'), and separate fore and background colors can be specified. The accepted text direction values:

```
#define GR_TEXT_RIGHT           0       /* normal */
#define GR_TEXT_DOWN            1       /* downward */
#define GR_TEXT_LEFT            2       /* upside down, right to left */
#define GR_TEXT_UP              3       /* upward */
#define GR_TEXT_DEFAULT         GR_TEXT_RIGHT
```

The accepted horizontal and vertical alignment option values:

```
#define GR_ALIGN_LEFT           0       /* X only */
#define GR_ALIGN_TOP            0       /* Y only */
#define GR_ALIGN_CENTER         1       /* X, Y */
#define GR_ALIGN_RIGHT          2       /* X only */
#define GR_ALIGN_BOTTOM         2       /* Y only */
#define GR_ALIGN_BASELINE       3       /* Y only */
#define GR_ALIGN_DEFAULT        GR_ALIGN_LEFT
```

Text strings can be of three different types: one character per byte (i.e. the usual C character string, this is the default), one character per 16-bit word (suitable for fonts with a large number of characters), and a PC-style character-attribute pair. In the last case the 'GrTextOption' structure must contain a pointer to a color table of size 16 (fg color bits in attrib) or 8 (bg color bits). (The color table format is explained in more detail in the previous section explaining the methods to build fill patterns.) The supported text types:

```
#define GR_BYTE_TEXT            0       /* one byte per character */
#define GR_WORD_TEXT            1       /* two bytes per character */
#define GR_ATTR_TEXT            2       /* char w/ PC style attribute byte */
```

The PC-style attribute text uses the same layout (first byte: character, second: attributes) and bitfields as the text mode screen on the PC. The only difference is that the 'blink' bit is not supported (it would be very time consuming -- the PC text mode does it with hardware support). This bit is used instead to control the underlined display of characters. For convenience the following attribute manipulation macros have been declared in "grx.h":

```
#define GR_BUILD_ATTR(fgcolor,bgcolor,underline)  \
    ((fgcolor) & 15) | (((bgcolor) & 7) << 4) | (((underline) & 1) << 7))
#define GR_ATTR_FGCOLOR(attr)   ((attr) & 15)
#define GR_ATTR_BGCOLOR(attr)   (((attr) >> 4) & 7)
#define GR_ATTR_UNDERLINE(attr) (((attr) >> 7) & 1)
```

Text strings of the types 'GR_BYTE_TEXT' and 'GR_WORD_TEXT" can also be drawn underlined. This is controlled by OR-ing the constant 'GR_UNDERLINE_TEXT' to the foreground color value:

```
#define GR_UNDERLINE_TEXT               (GrXOR << 6)
```

After the application initializes a text option structure with the desired values it can call one of the following two text drawing functions:

```
void    GrDrawChar(int chr,int x,int y,GrTextOption *opt);
void    GrDrawString(char *text,int length,int x,int y,GrTextOption *opt);
```

NOTE: text drawing is fastest when the font is not magnified, it is drawn in the 'normal' direction, and the character does not have to be clipped. It this case the library can use the appropriate low-level video RAM access routine (see "INTERNAL.DOC" for more details), while in any other case the text is drawn pixel-by-pixel (or rectangle-by-rectangle if enlarged) by the higher-level code.

The function 'GrTextXY' is provided for compatibility with the original 256 color DJGPP graphics library. It draws the text in the standard direction, unmagnified, and using the 16 row BIOS font on VGA-s or the 14 row font on EGA-s.

```
void    GrTextXY(int x,int y,char *text,int fg,int bg);
```

The size of a font, a character or a text string can be obtained by calling one of the following functions. These functions also take into consideration the magnification and text direction specified in the text option structure passed to them.

```
int     GrFontHeight(GrTextOption *opt);
int     GrFontWidth(GrTextOption *opt);
int     GrCharWidth(int chr,GrTextOption *opt);
int     GrCharHeight(int chr,GrTextOption *opt);
int     GrStringWidth(char *text,int length,GrTextOption *opt);
int     GrStringHeight(char *text,int length,GrTextOption *opt);
```

The 'GrTextRegion' structure and its associated functions can be used to implement a fast (as much as possible in graphics modes) rectangular text window using a fixed font. Clipping for such windows is done in character size increments instead of pixels (i.e. no partial characters are drawn). Only fixed fonts can be used in their natural size. 'GrDumpText' will cache the code of the drawn characters in the buffer pointed to by the 'backup' slot (if it is non-NULL) and will draw a character only if the previously drawn character in that grid element is different. This can speed up text scrolling significantly in graphics modes. The supported text types are the same as above.

```
typedef struct {
    GrFont *txr_font;                   /* font to be used */
    char   *txr_buffer;                 /* pointer to text buffer */
    char   *txr_backup;                 /* optional backup buffer */
    int     txr_xpos;                   /* upper left corner X coordinate */
```

```
    int     txr_ypos;                   /* upper left corner Y coordinate */
    int     txr_width;                  /* width of area in chars */
    int     txr_height;                 /* height of area in chars */
    int     txr_lineoffset;             /* offset in buffer(s) between lines */
    union {
        int v;                          /* color when no attributes */
        GrColorTableP p;                /* ptr to color table otherwise */
    } txr_fgcolor,txr_bgcolor;          /* foreground, background */
    char    txr_chrtype;                /* character type (see above) */
} GrTextRegion;

void    GrDumpChar(int chr,int col,int row,GrTextRegion *r);
void    GrDumpText(int col,int row,int wdt,int hgt,GrTextRegion *r);
void    GrDumpTextRegion(GrTextRegion *r);
```

The 'GrDumpTextRegion' function outputs the whole text region, while 'GrDumpText' draws only a user-specified part of it. 'GrDumpChar' updates the character in the buffer at the specified location with the new character passed to it as argument and then draws the new character on the screen as well.

Drawing in user coordinates

There is a second set of the graphics primitives which operates in user coordinates. Every context has a user to screen coordinate mapping associated with it. An application specifies the user window by calling the 'GrSetUserWindow' function.

```
void    GrSetUserWindow(int x1,int y1,int x2,int y2);
```

A call to this function it in fact specifies the virtual coordinate limits which will be mapped onto the current context regardless of the size of the context. For example, the call:

```
GrSetUserWindow(0,0,11999,8999);
```

tells the library that the program will perform its drawing operations in a coordinate system X:0...11999 (width = 12000) and Y:0...8999 (height = 9000). This coordinate range will be mapped onto the total area of the current context. The virtual coordinate system can also be shifted. For example:

```
GrSetUserWindow(5000,2000,16999,10999);
```

The user coordinates can even be used to turn the usual left-handed coordinate system (0:0 corresponds to the upper left corner) to a right handed one (0:0 corresponds to the bottom left corner) by calling:

```
GrSetUserWindow(0,8999,11999,0);
```

The library also provides three utility functions for the query of the current user coordinate limits and for converting user coordinates to screen coordinates and vice versa.

```
void    GrGetUserWindow(int *x1,int *y1,int *x2,int *y2);
void    GrGetScreenCoord(int *x,int *y);
void    GrGetUserCoord(int *x,int *y);
```

If an application wants to take advantage of the user to screen coordinate mapping it has to use the user coordinate version of the graphics primitives. These have exactly the same parameter passing conventions as their screen coordinate counterparts. **NOTE: the user coordinate system is not initialized by the library!** The application has to set up its coordinate mapping before calling any of the use coordinate drawing functions -- otherwise the program will almost certainly exit (in a quite ungraceful fashion) with a 'division by zero' error. The list of supported user coordinate drawing functions:

```
void    GrUsrPlot(int x,int y,int c);
void    GrUsrLine(int x1,int y1,int x2,int y2,int c);
void    GrUsrHLine(int x1,int x2,int y,int c);
void    GrUsrVLine(int x,int y1,int y2,int c);
void    GrUsrBox(int x1,int y1,int x2,int y2,int c);
void    GrUsrFilledBox(int x1,int y1,int x2,int y2,int c);
void    GrUsrFramedBox(int x1,int y1,int x2,int y2,int wdt,GrFBoxColors *c);
void    GrUsrCircle(int xc,int yc,int r,int c);
void    GrUsrEllipse(int xc,int yc,int xa,int ya,int c);
void    GrUsrCircleArc(int xc,int yc,int r,int start,int end,int c);
void    GrUsrEllipseArc(int xc,int yc,int xa,int ya,int start,int end,int c);
void    GrUsrFilledCircle(int xc,int yc,int r,int c);
void    GrUsrFilledEllipse(int xc,int yc,int xa,int ya,int c);
void    GrUsrFilledCircleArc(int xc,int yc,int r,int start,int end,int c);
void    GrUsrFilledEllipseArc(int xc,int yc,int xa,int ya,int start,int end,int c);
void    GrUsrPolyLine(int numpts,int points[][2],int c);
void    GrUsrPolygon(int numpts,int points[][2],int c);
void    GrUsrFilledConvexPolygon(int numpts,int points[][2],int c);
void    GrUsrFilledPolygon(int numpts,int points[][2],int c);
int     GrUsrPixel(int x,int y);

void    GrUsrCustomLine(int x1,int y1,int x2,int y2,GrLineOption *o);
void    GrUsrCustomBox(int x1,int y1,int x2,int y2,GrLineOption *o);
void    GrUsrCustomCircle(int xc,int yc,int r,GrLineOption *o);
void    GrUsrCustomEllipse(int xc,int yc,int xa,int ya,GrLineOption *o);
void    GrUsrCustomCircleArc(int xc,int yc,int r,int start,int end,GrLineOption *o);
void    GrUsrCustomEllipseArc(int xc,int yc,int xa,int ya,int start,int end,GrLineOption *o);
void    GrUsrCustomPolyLine(int numpts,int points[][2],GrLineOption *o);
void    GrUsrCustomPolygon(int numpts,int points[][2],GrLineOption *o);

void    GrUsrPatternedPlot(int x,int y,GrPattern *p);
void    GrUsrPatternedLine(int x1,int y1,int x2,int y2,GrLinePattern *lp);
void    GrUsrPatternedBox(int x1,int y1,int x2,int y2,GrLinePattern *lp);
void    GrUsrPatternedCircle(int xc,int yc,int r,GrLinePattern *lp);
void    GrUsrPatternedEllipse(int xc,int yc,int xa,int ya,GrLinePattern *lp);
void    GrUsrPatternedCircleArc(int xc,int yc,int r,int start,int end,GrLinePattern *lp);
void    GrUsrPatternedEllipseArc(int xc,int yc,int xa,int ya,int start,int end,GrLinePattern *lp);
void    GrUsrPatternedPolyLine(int numpts,int points[][2],GrLinePattern *lp);
void    GrUsrPatternedPolygon(int numpts,int points[][2],GrLinePattern *lp);

void    GrUsrPatternFilledBox(int x1,int y1,int x2,int y2,GrPattern *p);
void    GrUsrPatternFilledCircle(int xc,int yc,int r,GrPattern *p);
void    GrUsrPatternFilledEllipse(int xc,int yc,int xa,int ya,GrPattern *p);
void    GrUsrPatternFilledCircleArc(int xc,int yc,int r,int start,int end,GrPattern *p);
void    GrUsrPatternFilledEllipseArc(int xc,int yc,int xa,int ya,int start,int end,GrPattern *p);
void    GrUsrPatternFilledConvexPolygon(int numpts,int points[][2],GrPattern *p);
void    GrUsrPatternFilledPolygon(int numpts,int points[][2],GrPattern *p);

GrTextOption *GrUsrFindBestFont(int width,int height,int magnify,char *family,GrTextOption *where);
void    GrUsrDrawChar(int chr,int x,int y,GrTextOption *opt);
void    GrUsrDrawString(char *text,int length,int x,int y,GrTextOption *opt);
void    GrUsrTextXY(int x,int y,char *text,int fg,int bg);
```

Graphics cursors

The library provides support for the creation and usage of an unlimited number of graphics cursors. An application can use these cursors for any purpose. Cursors always save the area they occupy before they are drawn. When moved or erased they restore this area. As a general rule of thumb, an application should erase a cursor before making changes to an area it occupies and redraw the cursor after finishing the drawing. All cursor and mouse related declaration are in the include file "mousex.h". Cursors are created with the 'GrBuildCursor' function:

```
GrCursor *GrBuildCursor(char *data,int w,int h,int xo,int yo,GrColorTableP c);
```

The 'data', 'w' (=width), 'h' (=height) and 'c' (= color table) arguments are similar to the arguments of the pixmap building library function: 'GrBuildPixmap'. (See that paragraph for a more detailed explanation.) The only difference is that the pixmap data is interpreted slightly differently: any pixel with value zero is taken as a "transparent" pixel, i.e. the background will show through the cursor pattern at that pixel. A pixmap data byte with value = 1 will refer to the first color in the table, and so on. The 'xo' (= X offset) and 'yo' (= Y offset) arguments specify the position (from the top left corner of the cursor pattern) of the cursor's "hot point". The 'GrCursor' data structure:

```
typedef struct {
    GrVidRAM  cr_andmask;            /* cursor bitmap to AND */
    GrVidRAM  cr_ormask;             /* cursor bitmap to OR */
    GrVidRAM  cr_work;               /* work area */
    GrVidRAM  cr_save;               /* screen save area */
    int   cr_xcord,cr_ycord;         /* cursor position on screen */
    int   cr_xsize,cr_ysize;         /* cursor size */
    int   cr_xoffs,cr_yoffs;         /* LU corner to hot point offset */
    int   cr_xwork,cr_ywork;         /* save/work area sizes */
    int   cr_xwpos,cr_ywpos;         /* save/work area position on screen */
    int   cr_displayed;              /* set if displayed */
} GrCursor;
```

is typically not used (i.e. read or changed) by the application program, it should just pass pointers to these structures to the appropriate library functions. Other cursor manipulation functions:

```
void GrDestroyCursor(GrCursor *cursor);
void GrDisplayCursor(GrCursor *cursor);
void GrEraseCursor(GrCursor *cursor);
void GrMoveCursor(GrCursor *cursor,int x,int y);
```

Mouse event handling

All mouse services need the presence of a mouse and an installed Microsoft compatible mouse driver. An application can test whether a mouse is available by calling the function:

```
int  MouseDetect(void);
```

which will return zero if no mouse (or mouse driver) is present, non-zero otherwise. If the mouse is present the application may decide if it wants to use the mouse in interrupt-driven or polled mode. The polled mode is compatible with previous releases of DJGPP and the 256 color graphics library, it uses the mouse driver interrupts (INT 33h) to query the mouse about its position and buttons. This method is adequate if the program can do the polling in a tight enough loop. If the program does lengthy computations in the background during which a "frozen" mouse and the loss of mouse button presses would be disturbing it has to use the interrupt driven method. For this a patched version of GO32 is needed -- a GO32 version dated after the middle of April 1992 should work. The interrupt driven mouse event mechanism uses an event queue library (described in the document "EVENTS.DOC") which stores all user interaction events (mouse presses and keyboard hits) in a queue, timestamped, in the order of occurrence. The disadvantage of the interrupt-driven mouse event mechanism is that it may be harder to debug. To select between the two modes the following function needs to be called:

```
void MouseEventMode(int use_interrupts);
```

If the 'use_interrupts' parameter is zero the mouse is put into polled mode (this is the default), otherwise it will work interrupt-driven. After selecting the mode, the mouse can be initialized by calling:

```
void MouseInit(void);
```

It is not necessary to call this function as the first call the 'MouseGetEvent' (see later) function will also perform the initialization. However, if the mouse event mode is changed after using 'MouseGetEvent', a call to 'MouseInit' is the only way to enforce the change.

If the mouse is used in interrupt-driven mode, it is a good practice to call 'MouseUnInit' before exiting the program. This will restore any interrupt vectors hooked by the program to their original values.

```
void MouseUnInit(void);
```

The mouse can be controlled with the following functions:

```
void MouseSetSpeed(int speed);
void MouseSetAccel(int thresh,int accel);
void MouseSetLimits(int x1,int y1,int x2,int y2);
void MouseGetLimits(int *x1,int *y1,int *x2,int *y2);
void MouseWarp(int x,int y);
```

The library calculates the mouse position only from the mouse mickey counters. (To avoid the limit and 'rounding to the next multiple of eight' problem with the Microsoft mouse driver when it finds itself in a graphics mode unknown to it.) The 'speed' parameter to the 'MouseSetSpeed' function is used as a divisor, i.e. coordinate changes are obtained by dividing the mickey counter changes by this value. In high resolution graphics modes the value of one just works fine, in low resolution modes (320x200 or similar) it is best set the speed to two or three. (Of course, it also depends on the sensitivity the mouse.) The 'MouseSetAccel' function is used to control the ballistic effect: if a mouse coordinate changes between two samplings by more than the 'thresh' parameter, the change is multiplied by the 'accel' parameter. NOTE: some mouse drivers perform similar calculations before reporting the coordinates in mickeys. In this case the acceleration done by the library will be additional to the one already performed by the mouse driver. The limits of the mouse movement can be set (passed limits will be clipped to the screen) with 'MouseSetLimits' (default is the whole screen) and the current limits can be obtained with 'MouseGetLimits'. 'MouseWarp' sets the mouse cursor to the specified position.

As typical mouse drivers do not know how to draw mouse cursors in high resolution graphics modes, the mouse cursor is drawn by the library. The mouse cursor can be set with:

```
void MouseSetCursor(GrCursor *cursor);
void MouseSetColors(int fg,int bg);
```

'MouseSetColors' uses an internal arrow pattern, the color 'fg' will be used as the interior of it and 'bg' will be the border. The current mouse cursor can be obtained with:

```
GrCursor *MouseGetCursor(void);
```

The mouse cursor can be displayed/erased with:

```
void MouseDisplayCursor(void);
void MouseEraseCursor(void);
```

The mouse cursor can be left permanently displayed. All graphics primitives except for the few non-clipping functions check for conflicts with the mouse cursor and erase it before the drawing if necessary. Of course, it may be more efficient to erase the cursor manually before a long drawing sequence and redraw it after completion. The library provides an alternative pair of calls for this purpose which will erase the cursor only if it interferes with the drawing:

```
int  MouseBlock(GrContext *c,int x1,int y1,int x2,int y2);
void MouseUnBlock(void);
```

'MouseBlock' should be passed the context in which the drawing will take place (the usual convention of NULL meaning the current context is supported) and the limits of the affected area. It will erase the cursor only if it interferes with the drawing. If it returns a non-zero value then 'MouseUnBlock' has to be called at the end of the drawing, otherwise it should not be called.

The library supports (beside the simple cursor drawing) three types of "rubberband" attached to the mouse cursor. The 'MouseSetCursorMode' function is used to select the cursor drawing mode.

```
void MouseSetCursorMode(int mode,...);
```

The parameter 'mode' can have the following values:

```
/*
 * MOUSE CURSOR modes:
 *  M_CUR_NORMAL -- just the cursor
 *  M_CUR_RUBBER -- rectangular rubber band (XOR-d to the screen)
 *  M_CUR_LINE   -- line attached to the cursor
 *  M_CUR_BOX    -- rectangular box dragged by the cursor
 */
#define M_CUR_NORMAL    0
#define M_CUR_RUBBER    1
#define M_CUR_LINE      2
#define M_CUR_BOX       3
```

'MouseSetCursorMode' takes different parameters depending on the cursor drawing mode selected. The accepted call formats are:

```
MouseSetCursorMode(M_CUR_NORMAL);
MouseSetCursorMode(M_CUR_RUBBER,xanchor,yanchor,color);
MouseSetCursorMode(M_CUR_LINE,xanchor,yanchor,color);
MouseSetCursorMode(M_CUR_BOX,dx1,dy1,dx2,dy2,color);
```

The anchor parameters for the rubberband and rubberline modes specify a fixed screen location to which the other corner of the primitive is bound. The 'dx1' through 'dy2' parameters define the offsets of the corners of the dragged box from the hotpoint of the mouse cursor. The color value passed is always XOR-ed to the screen, i.e. if an application wants the rubberband to appear in a given color on a given background then it has to pass the XOR of these two colors to 'MouseSetCursorMode'.

The status of the mouse cursor can be obtained with calling 'MouseCursorIsDisplayed'. This function will return non-zero if the cursor is displayed, zero if it is erased.

```
int  MouseCursorIsDisplayed(void);
```

The 'MouseGetEvent' function is used to obtain the next mouse or keyboard event. It takes a flag with various bits encoding the type of event needed. It returns the event in a 'MouseEvent' structure. The relevant declarations from "mousex.h":

```
void MouseGetEvent(int flags,MouseEvent *event);

typedef struct {
    int  flags;                     /* flags (see above) */
    int  x,y;                       /* coordinates */
    int  buttons;                   /* button state */
    int  key;                       /* key code from keyboard */
    int  kbstat;                    /* keybd status (ALT, CTRL, etc..) */
    long time;                      /* time stamp of the event */
} MouseEvent;
```

The event structure has been extended with a keyboard status word (thus a program can check for combinations like ALT-<left mousebutton press>) and a time stamp (in DOS clock ticks since the start of the program) which can be used to check for double clicks, etc... The following macros have been defined in "mousex.h" to help in creating the control flag for 'MouseGetEvent' and decoding the various bits in the event structure:

```
/*
 * MOUSE event flag bits
 */
#define M_MOTION        0x001
#define M_LEFT_DOWN     0x002
#define M_LEFT_UP       0x004
#define M_RIGHT_DOWN    0x008
#define M_RIGHT_UP      0x010
#define M_MIDDLE_DOWN   0x020
#define M_MIDDLE_UP     0x040
#define M_BUTTON_DOWN   (M_LEFT_DOWN | M_MIDDLE_DOWN | M_RIGHT_DOWN)
```

```
#define M_BUTTON_UP     (M_LEFT_UP    | M_MIDDLE_UP   | M_RIGHT_UP)
#define M_BUTTON_CHANGE (M_BUTTON_UP | M_BUTTON_DOWN )

/*
 * MOUSE button status bits
 */
#define M_LEFT          1
#define M_RIGHT         2
#define M_MIDDLE        4

/*
 * Other bits and combinations
 */
#define M_KEYPRESS      0x080           /* keypress */
#define M_POLL          0x100           /* do not wait for the event */
#define M_NOPAINT       0x200
#define M_EVENT         (M_MOTION | M_KEYPRESS | M_BUTTON_DOWN | M_BUTTON_UP)

/*
 * KEYBOARD status word bits
 */
#define KB_RIGHTSHIFT   0x01            /* right shift key depressed */
#define KB_LEFTSHIFT    0x02            /* left shift key depressed */
#define KB_CTRL         0x04            /* CTRL depressed */
#define KB_ALT          0x08            /* ALT depressed */
#define KB_SCROLLOCK    0x10            /* SCROLL LOCK active */
#define KB_NUMLOCK      0x20            /* NUM LOCK active */
#define KB_CAPSLOCK     0x40            /* CAPS LOCK active */
#define KB_INSERT       0x80            /* INSERT state active */

#define KB_SHIFT        (KB_LEFTSHIFT | KB_RIGHTSHIFT)
```

'MouseGetEvent' will display the mouse cursor if it was previously erased and the 'M_NOPAINT' bit is not set in the flag passed to it. In this case it will also erase the cursor after an event has been obtained.

NOTE: some of the mouse event constants have different values than in the original DJGPP graphics library. This change was necessary to better follow the mouse driver conventions for assigning bits with similar functions.

The generation of mouse and keyboard events can be individually enabled or disabled (by passing a non-zero or zero, respectively, value in the corresponding 'enable_XX' parameter) by calling:

```
void MouseEventEnable(int enable_kb,int enable_ms);
```