GRX 2.4.8 User's Manual

A 2D graphics library for DOS, Linux, X11 and Win32

Based on the original doc written by: Csaba Biegl on August 10, 1992
Updated by: Mariano Alvarez Fernandez on August 17, 2000
Last update: August 14, 2007

Abstract

GRX is a 2D graphics library originaly written by Csaba Biegl for DJ Delorie's DOS port of the GCC compiler. Now it support a big range of platforms, the main four are: DOS (DJGPPv2), Linux console, X11 and Win32 (Mingw). On DOS it supports VGA, EGA and VESA compliant cards. On Linux console it uses svgalib or the framebuffer. On X11 it must work on any X11R5 (or later). From the 2.4 version, GRX comes with a Win32 driver. The framebuffer Linux console driver was new in 2.4.2. From 2.4.7 there is a support for x86_64 bits Linux machines and a support for an SDL driver on MingW and X11. On MingW and X11 it runs on a window with the original driver, and either full screen or on a window with the SDL driver.

GRX2 User's Manual

Hello world

The next program draws a double frame around the screen and writes "Hello, GRX world" centered. Then it waits after a key is pressed.

```
#include <string.h>
#include <grx20.h>
#include <grxkeys.h>

int main()
{
  char *message = "Hello, GRX world";
  int x, y;
  GrTextOption grt;

  GrSetMode( GR_default_graphics );

  grt.txo_font = &GrDefaultFont;
  grt.txo_fgcolor.v = GrWhite();
  grt.txo_bgcolor.v = GrBlack();
  grt.txo_direct = GR_TEXT_RIGHT;
  grt.txo_xalign = GR_ALIGN_CENTER;
  grt.txo_yalign = GR_ALIGN_CENTER;
  grt.txo_chrtype = GR_BYTE_TEXT;

  GrBox( 0,0,GrMaxX(),GrMaxY(),GrWhite() );
  GrBox( 4,4,GrMaxX()-4,GrMaxY()-4,GrWhite() );

  x = GrMaxX()/2;
  y = GrMaxY()/2;
  GrDrawString( message,strlen( message ),x,y,&grt );

  GrKeyRead();

  return 0;
}
```

How to compile the hello world (assuming the GRX library was previously installed)

```
DJGPP: gcc -o hellogrx.exe hellogrx.c -lgrx20
Mingw: gcc -o hellogrx.exe hellogrx.c -lgrx20 -mwindows
X11  : gcc -o hellogrx hellogrx.c -D__XWIN__ -I/usr/X11R6/include
       -lgrx20X -L/usr/X11R6/lib -lX11
Linux: gcc -o hellogrx hellogrx.c -lgrx20 -lvga

For the SDL driver:
```

```
Mingw: gcc -o hellogrx.exe hellogrx.c -lgrx20S -lSDL
X11  : gcc -o hellogrx hellogrx.c -D__XWIN__ -I/usr/X11R6/include
       -lgrx20S -lSDL -lpthread -L/usr/X11R6/lib -lX11

For x86_64 systems add -m32 or -m64 for 32/64 bits executables
and replace /lib by /lib64 as needed
```

Data types and 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
* grfontdv.h   format of a font when loaded into memory
* grx20.h      drawing-related structures and functions
* grxkeys.h    platform independent key definitions
```

User programs normally only include **include/grx20.h** and **include/grxkeys.h**

Setting the graphics driver

The graphics driver is normally set by the final user by the environment variable GRX20DRV, but a program can set it using:

```
int GrSetDriver(char *drvspec);
```

The drvspec string has the same format as the environment variable:

```
<driver> gw <width> gh <height> nc <colors>
```

Available drivers are for:

```
* DOS => herc, stdvga, stdega, et4000, cl5426, mach64, ati28800, s3, VESA, memory
* Linux => svgalib, linuxfb, memory
* X11 => xwin, memory
* Win32 => win32, memory
* SDL (Win32 and X11) => sdl::fs, sdl::ww, memory
```

The xwin and win32 drivers are windowed. The SDL driver on the same systems can be either fullscreen (::fs) or windowed (::ww).

The optionals gw, gh and nc parameters set the desired default graphics mode. Normal values for 'nc' are 2, 16, 256, 64K and 16M. The current driver name can be obtained from:

```
GrCurrentVideoDriver()->name
```

Setting video modes

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

```
int GrSetMode(int which,...);
```

On succes it returns non-zero (TRUE). The which parameter can be one of the following constants, declared in grx20.h:

```
typedef enum _GR_graphicsModes {
  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_width_height_color_text,
  GR_custom_graphics,
  GR_width_height_bpp_graphics,
  GR_width_height_bpp_text,
  GR_custom_bpp_graphics,
  GR_NC_80_25_text,
  GR_NC_default_text,
  GR_NC_width_height_text,
  GR_NC_biggest_text,
  GR_NC_320_200_graphics,
  GR_NC_default_graphics,
  GR_NC_width_height_graphics,
  GR_NC_biggest_noninterlaced_graphics,
  GR_NC_biggest_graphics,
  GR_NC_width_height_color_graphics,
  GR_NC_width_height_color_text,
  GR_NC_custom_graphics,
  GR_NC_width_height_bpp_graphics,
  GR_NC_width_height_bpp_text,
  GR_NC_custom_bpp_graphics,
} GrGraphicsMode;
```

The GR_width_height_text and GR_width_height_graphics modes require the two size arguments: int width and int height.

The GR_width_height_color_graphics and GR_width_height_color_text modes require three arguments: int width, int height and GrColor colors.

The GR_width_height_bpp_graphics and GR_width_height_bpp_text modes require three arguments: int width, int height and int bpp (bits per plane instead number of colors).

The GR_custom_graphics and GR_custom_bpp_graphics modes require five arguments: int width, int height, GrColor colors or int bpp, int vx and int vy. Using this modes you can set a virtual screen of vx by vy size.

A call with any other mode does not require any arguments.

The GR_NC_... modes are equivalent to the GR_.. ones, but they don't clear the video memory.

Graphics drivers can provide info of the supported graphics modes, use the next code skeleton to colect the data:

```
{
  GrFrameMode fm;
  const GrVideoMode *mp;
  for(fm =GR_firstGraphicsFrameMode; fm <= GR_lastGraphicsFrameMode; fm++) {
    mp = GrFirstVideoMode(fm);
    while( mp != NULL ) {
      ..
      .. use the mp info
      ..
      mp = GrNextVideoMode(mp))
    }
  }
}
```

Don't worry if you don't understand it, normal user programs don't need to know about FrameModes. The GrVideoMode structure has the following fields:

```
typedef struct _GR_videoMode GrVideoMode;

struct _GR_videoMode {
  char    present;                    /* is it really available? */
  char    bpp;                        /* log2 of # of colors */
  short   width,height;               /* video mode geometry */
  short   mode;                       /* BIOS mode number (if any) */
  int     lineoffset;                 /* scan line length */
  int     privdata;                   /* driver can use it for anything */
  struct _GR_videoModeExt *extinfo;   /* extra info (maybe shared) */
};
```

The width, height and bpp members are the useful information if you are interested in set modes other than the GR_default_graphics.

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 don't return any value. It can be installed (for all subsequent GrSetMode calls) with the:

```
void GrSetModeHook(void (*hookfunc)(void));
```

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

```
GrGraphicsMode GrCurrentMode(void);
```

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

```
GrVideoAdapter GrAdapterType(void);
```

function. GrAdapterType returns the type of the adapter as one of the following symbolic constants (defined in grx20.h):

```
typedef enum _GR_videoAdapters {
  GR_UNKNOWN = (-1),     /* not known (before driver set) */
  GR_VGA,                /* VGA adapter */
```

```
  GR_EGA,               /* EGA adapter */
  GR_HERC,              /* Hercules mono adapter */
  GR_8514A,             /* 8514A or compatible */
  GR_S3,                /* S3 graphics accelerator */
  GR_XWIN,              /* X11 driver */
  GR_WIN32,             /* WIN32 driver */
  GR_LNXFB,             /* Linux framebuffer */
  GR_SDL,               /* SDL driver */
  GR_MEM                /* memory only driver */
} GrVideoAdapter;
```

Note that the VESA driver return GR_VGA here.

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 GrContext;

struct _GR_context {
  struct _GR_frame    gc_frame;       /* frame buffer info */
  struct _GR_context *gc_root;        /* context which owns frame */
  int    gc_xmax;                     /* max X coord (width  - 1) */
  int    gc_ymax;                     /* max Y coord (height - 1) */
  int    gc_xoffset;                  /* X offset from root's base */
  int    gc_yoffset;                  /* Y offset from root's base */
  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 */
# define gc_baseaddr                  gc_frame.gf_baseaddr
# define gc_selector                  gc_frame.gf_selector
# define gc_onscreen                  gc_frame.gf_onscreen
# define gc_memflags                  gc_frame.gf_memflags
# define gc_lineoffset                gc_frame.gf_lineoffset
# define gc_driver                    gc_frame.gf_driver
};
```

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[4],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 GRX 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 memory argument is really a 4 pointer array, each pointer must point to space to handle GrPlaneSize(w,h) bytes, really only GrNumPlanes() pointers must be malloced, the rest can be NULL. Nevertheless the normal use (see below) is

```
gc = GrCreateContext(w,h,NULL,NULL);
```

so yo don't need to care about.

The function:

```
GrContext *GrCreateSubContext(int x1,int y1,int x2,int y2,
                              const 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(const 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.) The next two functions:

```
const GrContext *GrCurrentContext(void);
const GrContext *GrScreenContext(void);
```

return the current context and the screen context respectively. 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 current 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. There are three similar functions to sets/gets the clipping limits of any context:

```
void  GrSetClipBoxC(GrContext *c,int x1,int y1,int x2,int y2);
void  GrGetClipBoxC(const GrContext *c,int *x1p,int *y1p,int *x2p,int *y2p);
void  GrResetClipBoxC(GrContext *c);
```

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);
```

If you had set a virtual screen (using a custom graphics mode), the limits of the virtual screen can be fetched with:

```
int GrVirtualX(void);
int GrVirtualY(void);
```

The routine:

```
int GrScreenIsVirtual(void);
```

returns non zero if a virtual screen is set. The rectangle showed in the real screen can be set with:

```
int GrSetViewport(int xpos,int ypos);
```

and the current viewport position can be obtained by:

```
int GrViewportX(void);
int GrViewportY(void);
```

Context use

Here is a example of normal context use:

```
GrContext *grc;

if( (grc = GrCreateContext( w,h,NULL,NULL )) == NULL ){
  ...process the error
  }
else {
  GrSetContext( grc );
  ...do some drawing
  ...and probably bitblt to the screen context
  GrSetContext( NULL ); /* the screen context! */
  GrDestroyContext( grc );
  }
```

But if you have a GrContext variable (not a pointer) you want to use (probably because is static to some routines) you can do:

```
static GrContext grc; /* not a pointer!! */

if( GrCreateContext( w,h,NULL,&grc )) == NULL ) {
  ...process the error
 }
else {
  GrSetContext( &grc );
  ...do some drawing
  ...and probably bitblt to the screen context
  GrSetContext( NULL ); /* the screen context! */
  GrDestroyContext( &grc );
  }
```

Note that GrDestoryContext knows if grc was automatically malloced or not!!

Only if you don't want GrCreateContext use malloc at all, you must allocate the memory buffers and pass it to GrCreateContext.

Using GrCreateSubContext is the same, except it doesn't need the buffer, because it uses the parent buffer.

See the **test/winclip.c** and **test/wintest.c** examples.

Color management

GRX defines the type GrColor for color variables. GrColor it's a 32 bits integer. The 8 left bits are reserved for the write mode (see below). The 24 bits right are the color value.

The library supports two models for color management. In the 'indirect' (or color table) model, color values are indices to a color table. The color table slots will 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) model, color values map

directly into component color intensities with non-overlapping bitfields of the color index representing the component colors.

Color table model is supported until 256 color modes. The RGB model is supported in 256 color and up color modes.

In RGB model the color index map to component color intensities depend on the video mode set, so it can't be assumed the component color bitfields (but if you are curious check the GrColorInfo global structure in grx20.h).

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

```
GrColor GrBlack(void);
GrColor GrWhite(void);
```

GrBlack() is guaranteed to be 0.

The library supports five write modes (a write mode descibes the operation between the actual bit color and the one to be set): write, XOR, logical OR, logical AND and IMAGE. These can be selected with OR-ing the color value with one of the following constants declared in grx20.h :

```
#define GrWRITE       0UL            /* write color */
#define GrXOR         0x01000000UL   /* to "XOR" any color to the screen */
#define GrOR          0x02000000UL   /* to "OR" to the screen */
#define GrAND         0x03000000UL   /* to "AND" to the screen */
#define GrIMAGE       0x04000000UL   /* blit: write, except given color */
```

The GrIMAGE write mode only works with the bitblt function. By convention, the no-op color is obtained by combining color value 0 (black) with the XOR operation. This no-op color has been defined in grx20.h as:

```
#define GrNOCOLOR     (GrXOR | 0)    /* GrNOCOLOR is used for "no" color */
```

The write mode part and the color value part of a GrColor variable can be obtained OR-ing it with one of the following constants declared in grx20.h:

```
#define GrCVALUEMASK  0x00ffffffUL   /* color value mask */
#define GrCMODEMASK   0xff000000UL   /* color operation mask */
```

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

```
GrColor GrNumColors(void);
```

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

```
GrColor GrNumFreeColors(void);
```

Colors can be allocated with the:

```
GrColor GrAllocColor(int r,int g,int b);
GrColor GrAllocColor2(long hcolor);
```

functions (component intensities can range from 0 to 255, hcolor must be in 0xRRGGBB format), or with the:

```
GrColor GrAllocCell(void);
```

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

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

In the color table model both Alloc functions return GrNOCOLOR if there are no more free colors available. In the RGB model GrNumFreeColors returns 0 and GrAllocCell always returns GrNOCOLOR, 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(GrColor color);
```

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

```
void GrQueryColor(GrColor c,int *r,int *g,int *b);
void GrQueryColor2(GrColor c,long *hcolor);
```

Initially the color system is in color table (indirect) model if there are 256 or less colors. 256 color modes can be put into the RGB model by calling:

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

The color system can be reset (i.e. put back into color table model 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.

This functions:

```
GrColor GrAllocColorID(int r,int g,int b);
GrColor GrAllocColor2ID(long hcolor);
void GrQueryColorID(GrColor c,int *r,int *g,int *b);
void GrQueryColor2ID(GrColor c,long *hcolor);
```

are inlined versions (except if you compile GRX with GRX_SKIP_INLINES defined) to be used in the RGB model (in the color table model they call the normal routines).

See the **test/rgbtest.c** and **test/colorops.c** examples.

Portable use of a few colors

People that only want to use a few colors find the GRX color handling a bit confusing, but it gives the power to manage a lot of color deeps and two color models. Here are some guidelines to easily use the famous 16 ega colors in GRX programs. We need this GRX function:

```
GrColor *GrAllocEgaColors(void);
```

it returns a 16 GrColor array with the 16 ega colors alloced (really it's a trivial function, read the source src/setup/colorega.c). We can use a construction like that:

First, in your C code make a global pointer, and init it after set the graphics mode:

```
GrColor *egacolors;
....
int your_setup_function( ... )
{
  ...
  GrSetMode( ... )
  ...
  egacolors = GrAllocEgaColors();
  ...
}
```

Next, add this to your main include file:

```
extern GrColor *egacolors;
#define BLACK        egacolors[0]
#define BLUE         egacolors[1]
#define GREEN        egacolors[2]
#define CYAN         egacolors[3]
#define RED          egacolors[4]
#define MAGENTA      egacolors[5]
#define BROWN        egacolors[6]
#define LIGHTGRAY    egacolors[7]
#define DARKGRAY     egacolors[8]
#define LIGHTBLUE    egacolors[9]
#define LIGHTGREEN   egacolors[10]
#define LIGHTCYAN    egacolors[11]
#define LIGHTRED     egacolors[12]
#define LIGHTMAGENTA egacolors[13]
#define YELLOW       egacolors[14]
#define WHITE        egacolors[15]
```

Now you can use the defined colors in your code. Note that if you are in color table model in a 16 color mode, you have exhausted the color table. Note too that this don't work to initialize static variables with a color, because egacolors is not initialized.

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(GrColor bg);
void GrClearcontext(GrColor bg);
void GrClearClipBox(GrColor bg);
```

Thanks to the special GrColor definition, you can do more than simple clear with this functions, by example with:

```
GrClearScreen( GrWhite()|GrXOR );
```

the graphics screen is negativized, do it again and the screen is restored.

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

```
void GrPlot(int x,int y,GrColor c);
void GrLine(int x1,int y1,int x2,int y2,GrColor c);
void GrHLine(int x1,int x2,int y,GrColor c);
void GrVLine(int x,int y1,int y2,GrColor c);
void GrBox(int x1,int y1,int x2,int y2,GrColor c);
void GrCircle(int xc,int yc,int r,GrColor c);
void GrEllipse(int xc,int yc,int xa,int ya,GrColor c);
void GrCircleArc(int xc,int yc,int r,int start,int end,int style,GrColor c);
void GrEllipseArc(int xc,int yc,int xa,int ya,
                  int start,int end,int style,GrColor c);
void GrPolyLine(int numpts,int points[][2],GrColor c);
void GrPolygon(int numpts,int points[][2],GrColor c);
```

All primitives operate on the current graphics context. The last argument of these functions is always the color 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 style argument can be one of this defines from grx20.h:

```
#define GR_ARC_STYLE_OPEN       0
#define GR_ARC_STYLE_CLOSE1     1
#define GR_ARC_STYLE_CLOSE2     2
```

GR_ARC_STYLE_OPEN draws only the arc, GR_ARC_STYLE_CLOSE1 closes the arc with a line between his start and end point, GR_ARC_STYLE_CLOSE1 draws the typical cake slice. This routine:

```
void GrLastArcCoords(int *xs,int *ys,int *xe,int *ye,int *xc,int *yc);
```

can be used to retrieve the start, end, and center points used by the last arc drawing functions.

See the **test/circtest.c** and **test/arctest.c** examples.

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 primitive can either contain or omit the closing edge of the polygon – the primitive will append it to the list if it is missing.

See the **test/polytest.c** example.

Because calculating the arc points it's a very time consuming operation, there are two functions to pre-calculate the points, that can be used next with polyline and polygon primitives:

```
int  GrGenerateEllipse(int xc,int yc,int xa,int ya,
                       int points[GR_MAX_ELLIPSE_POINTS][2]);
int  GrGenerateEllipseArc(int xc,int yc,int xa,int ya,int start,int end,
                          int points[GR_MAX_ELLIPSE_POINTS][2]);
```

The following filled primitives are available:

```
void GrFilledBox(int x1,int y1,int x2,int y2,GrColor c);
void GrFramedBox(int x1,int y1,int x2,int y2,int wdt,const GrFBoxColors *c);
void GrFilledCircle(int xc,int yc,int r,GrColor c);
void GrFilledEllipse(int xc,int yc,int xa,int ya,GrColor c);
void GrFilledCircleArc(int xc,int yc,int r,
                       int start,int end,int style,GrColor c);
void GrFilledEllipseArc(int xc,int yc,int xa,int ya,
                        int start,int end,int style,GrColor c);
void GrFilledPolygon(int numpts,int points[][2],GrColor c);
void GrFilledConvexPolygon(int numpts,int points[][2],GrColor 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, wdt pixels wide. The primitive uses five different colors for the interior and four borders of the box which are specified in the GrFBoxColors structure:

```
typedef struct {
  GrColor fbx_intcolor;
  GrColor fbx_topcolor;
  GrColor fbx_rightcolor;
  GrColor fbx_bottomcolor;
  GrColor 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 nonoverlapping polygons in a single operation.

The function:

```
void GrFloodFill(int x, int y, GrColor border, GrColor c);
```

flood-fills the area bounded by the color border using x, y like the starting point.

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

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

and:

```
GrColor GrPixelC(GrContext *c,int x,int y);
```

do the same for any context.

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,GrColor op);
```

x, y is the position in the destination context, and x1, y1, x2, y2 the area from the source context to be transfered. The op argument should be one of supported color write modes (GrWRITE, GrXOR, GrOR, GrAND, GrIMAGE), it will control how the pixels from the source context are combined with the pixels in the destination context (the GrIMAGE op must be ored with the color value to be handled as transparent). If either the source or the

destination context argument is the NULL pointer then the current context is used for that argument.

See the **test/blittest.c** example.

A efficient form to get/put pixels from/to a context can be achieved using the next functions:

```
const GrColor *GrGetScanline(int x1,int x2,int yy);
const GrColor *GrGetScanlineC(GrContext *ctx,int x1,int x2,int yy);
void GrPutScanline(int x1,int x2,int yy,const GrColor *c, GrColor op);
```

The Get functions return a pointer to a static GrColor pixel array (or NULL if they fail) with the color values of a row (yy) segment (x1 to x2). GrGetScanline uses the current context. GrGestScanlineC uses the context ctx (that can be NULL to refer to the current context). Note that the output is only valid until the next GRX call.

GrPutScanline puts the GrColor pixel array c on the yy row segmet defined by x1 to x2 in the current context using the op operation. op can be any of GrWRITE, GrXOR, GrOR, GrAND or GrIMAGE. Data in c must fit GrCVALUEMASK otherwise the results are implementation dependend. So you can't supply operation code with the pixel data!.

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,GrColor c);
void GrLineNC(int x1,int y1,int x2,int y2,GrColor c);
void GrHLineNC(int x1,int x2,int y,GrColor c);
void GrVLineNC(int x,int y1,int y2,GrColor c);
void GrBoxNC(int x1,int y1,int x2,int y2,GrColor c);
void GrFilledBoxNC(int x1,int y1,int x2,int y2,GrColor c);
void GrFramedBoxNC(int x1,int y1,int x2,int y2,int wdt,const GrFBoxCol-
ors *c);
void grbitbltNC(GrContext *dst,int x,int y,GrContext *src,
                int x1,int y1,int x2,int y2,GrColor op);
GrColor GrPixelNC(int x,int y);
GrColor GrPixelCNC(GrContext *c,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 {
  GrColor 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.

Example, a white line 3 bits wide (thick) and pattern 6 bits draw, 4 bits nodraw:

```
GrLineOption mylineop;
...
mylineop.lno_color = GrWhite();
mylineop.lno_width = 3;
mylineop.lno_pattlen = 2;
mylineop.lno_dashpat = "\x06\x04";
```

The available custom line drawing primitives:

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

See the **test/linetest.c** example.

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 relevant structure declarations (from grx20.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 _GR_bitmap {
  int     bmp_ispixmap;          /* type flag for pattern union */
  int     bmp_height;            /* bitmap height */
  char   *bmp_data;              /* pointer to the bit pattern */
  GrColor bmp_fgcolor;           /* foreground color for fill */
  GrColor 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 _GR_pixmap {
  int     pxp_ispixmap;          /* type flag for pattern union */
  int     pxp_width;             /* pixmap width (in pixels)  */
  int     pxp_height;            /* pixmap height (in pixels) */
  GrColor pxp_oper;              /* bitblt mode (SET, OR, XOR, AND, IM-
AGE) */
  struct _GR_frame pxp_source;   /* source context for fill */
} GrPixmap;

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

This define group (from grx20.h) help to acces the GrPattern menbers:

```
#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(const char *pixels,int w,int h,const GrColorTableP colors);
GrPattern *GrBuildPixmapFromBits(const char *bits,int w,int h,
                                 GrColor fgc,GrColor 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 GrColor *GrColorTableP;
```

should point to an array of integers with the first element being the number of colors in the table and the color values 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,
                              int style,GrPattern *p);
void GrPatternFilledEllipseArc(int xc,int yc,int xa,int ya,int start,int end,
                               int style,GrPattern *p);
void GrPatternFilledConvexPolygon(int numpts,int points[][2],GrPattern *p);
void GrPatternFilledPolygon(int numpts,int points[][2],GrPattern *p);
void GrPatternFloodFill(int x, int y, GrColor border, 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,
                          int style,GrLinePattern *lp);
void GrPatternedEllipseArc(int xc,int yc,int xa,int ya,int start,int end,
                           int style,GrLinePattern *lp);
void GrPatternedPolyLine(int numpts,int points[][2],GrLinePattern *lp);
void GrPatternedPolygon(int numpts,int points[][2],GrLinePattern *lp);
```

Image manipulation

GRX defines the GrImage type like a GrPixmap synonym:

```
#define GrImage GrPixmap
```

nevertheless the GrImage type enforces the image character of this object, so for compatibility with future GRX versions use the next functions if you need to convert between GrImage and GrPixmap objects:

```
GrImage *GrImageFromPattern(GrPattern *p);
GrPattern *GrPatternFromImage(GrImage *p);
```

the GrImageFromPattern function returns NULL if the GrPattern given is not a GrPixmap.

Like pixmaps patterns images are dependent of the actual video mode set. So the library provides functions to create images in a mode-independent way:

```
GrImage *GrImageBuild(const char *pixels,int w,int h,const GrColorTableP colors);
GrImage *GrImageFromContext(GrContext *c);
```

these functions work like the GrBuildPixmap and GrConvertToPixmap ones. Remember: the image and the original context share the drawing RAM.

There are a number of functions to display all or part of an image in the current context:

```
void GrImageDisplay(int x,int y, GrImage *i);
void GrImageDisplayExt(int x1,int y1,int x2,int y2, GrImage *i);
```

```
void GrImageFilledBoxAlign(int xo,int yo,int x1,int y1,int x2,int y2,
                           GrImage *p);
void GrImageHLineAlign(int xo,int yo,int x,int y,int width,GrImage *p);
void GrImagePlotAlign(int xo,int yo,int x,int y,GrImage *p);
```

GrImageDisplay display the whole image using x, y like the upper left corner in the current context. GrImageDisplayExt display as much as it can (repiting the image if necesary) in the rectangle defined by x1, y1 and x2, y2.

GrImageFilledBoxAlign is a most general funtion (really the later two call it) display as much as it can in the defined rectangle using xo, yo like the align point, it is the virtual point in the destination context (it doesn't need to be into the rectangle) with that the upper left image corner is aligned.

GrImageHLineAlign and GrImagePlotAlign display a row segment or a point of the image at x y position using the xo, yo allign point.

The most usefull image funtions are these:

```
GrImage *GrImageInverse(GrImage *p,int flag);
GrImage *GrImageStretch(GrImage *p,int nwidth,int nheight);
```

GrImageInverse creates a new image object, flipping p left-right or top-down as indicated by flag that can be:

```
#define GR_IMAGE_INVERSE_LR  0x01  /* inverse left right */
#define GR_IMAGE_INVERSE_TD  0x02  /* inverse top down */
```

GrImageStretch creates a new image stretching p to nwidth by nheight.

To destroy a image objet when you don't need it any more use:

```
void GrImageDestroy(GrImage *i);
```

See the **test/imgtest.c** example.

Text drawing

The library supports loadable fonts. When in memory they are bit-mapped (i.e. not scalable!) fonts. A driver design allow GRX to load different font formats, the last GRX release come with drivers to load the GRX own font format and the BGI Borland format for all platforms supported, the X11 version can load X11 fonts too.

The GRX distribution come with a font collection in the GRX own format. Some of these fonts were converted from VGA fonts. These fonts have all 256 characters from the PC-437 codepage. Some additional fonts were converted from fonts in the MIT X11 distribution. Most of these are ISO-8859-1 coded. Fonts also have family names. The following font families are included:

```
Font file name       Family  Description
pc<W>x<H>[t].fnt     pc      VGA 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 VGA fonts come in thin formats also, these are denoted by a 't' in their file names and the string "_thin" in their family names.

The GrFont structure hold a font in memory. A number of 'pc' fonts are built-in to the library and don't need to be loaded:

```
extern  GrFont          GrFont_PC6x8;
extern  GrFont          GrFont_PC8x8;
extern  GrFont          GrFont_PC8x14;
extern  GrFont          GrFont_PC8x16;
```

Other fonts must be loaded with the GrLoadFont function. If the font file name starts with any path separator character or character sequence (':', '/' or '\') then it is loaded from the specified directory, otherwise the library try load the font first from the current directory and next 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. 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);
void GrUnloadFont(GrFont *font);
void GrSetFontPath(char *path_list);
```

Using these functions:

```
GrFont *GrLoadConvertedFont(char *name,int cvt,int w,int h,
                            int minch,int maxch);
GrFont *GrBuildConvertedFont(const GrFont *from,int cvt,int w,int h,
                             int minch,int maxch);
```

a new font can be generated from a file font or a font in memory, the 'cvt' argument direct the conversion or-ing the desired operations from these defines:

```
/*
 * Font conversion flags for 'GrLoadConvertedFont'. OR them as desired.
 */
#define GR_FONTCVT_NONE         0     /* no conversion */
#define GR_FONTCVT_SKIPCHARS    1     /* load only selected characters */
#define GR_FONTCVT_RESIZE       2     /* resize the font */
#define GR_FONTCVT_ITALICIZE    4     /* tilt font for "italic" look */
#define GR_FONTCVT_BOLDIFY      8     /* make a "bold"(er) font  */
#define GR_FONTCVT_FIXIFY       16    /* convert prop. font to fixed wdt */
```

```
#define GR_FONTCVT_PROPORTION   32    /* convert fixed font to prop. wdt */
```

GR_FONTCVT_SKIPCHARS needs 'minch' and 'maxch' arguments.

GR_FONTCVT_RESIZE needs 'w' and 'h' arguments.

The function:

```
void GrDumpFnaFont(const GrFont *f, char *fileName);
```

writes a font to an ascii font file, so it can be quickly edited with a text editor. For a description of the ascii font format, see the fna.txt file.

The function:

```
void GrDumpFont(const GrFont *f,char *CsymbolName,char *fileName);
```

writes a font to a C source code file, so it can be compiled and linked with a user program. GrDumpFont would not normally be used in a release program because its purpose is to produce source code. When the source code is compiled and linked into a program distributing the font file with the program in not necessary, avoiding the possibility of the font file being deleted or corrupted.

You can use the premade fnt2c.c program (see the source, it's so simple) to dump a selected font to source code, by example:

```
fnt2c helv15 myhelv15 myhelv15.c
```

Next, if this line is included in your main include file:

```
extern GrFont myhelv15
```

and "myhelv15.c" compiled and linked with your project, you can use 'myhelv15' in every place a GrFont is required.

This simple function:

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

draw text in the current context in the standard direction, using the GrDefaultFont (mapped in the grx20.h file to the GrFont_PC8x14 font) with x, y like the upper left corner and the foreground and background colors given (note that bg equal to GrNOCOLOR make the background transparent).

For other functions the GrTextOption structure specifies how to draw a character string:

```
typedef struct _GR_textOption {      /* text drawing option structure */
  struct _GR_font     *txo_font;      /* font to be used */
  union  _GR_textColor txo_fgcolor;   /* foreground color */
  union  _GR_textColor txo_bgcolor;   /* background color */
  char    txo_chrtype;                /* character type (see above) */
  char    txo_direct;                 /* direction (see above) */
  char    txo_xalign;                 /* X alignment (see above) */
  char    txo_yalign;                 /* Y alignment (see above) */
} GrTextOption;

typedef union _GR_textColor {        /* text color union */
  GrColor       v;                    /* color value for "direct" text */
  GrColorTableP p;                    /* color table for attribute text */
} GrTextColor;
```

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       /* chr 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 grx20.h:

```
#define GR_BUILD_ATTR(fg,bg,ul) \
        (((fg) & 15) | (((bg) & 7) << 4) | ((ul) ? 128 : 0))
#define GR_ATTR_FGCOLOR(attr)   (((attr)     ) &  15)
#define GR_ATTR_BGCOLOR(attr)   (((attr) >> 4) &   7)
#define GR_ATTR_UNDERLINE(attr) (((attr)     ) & 128)
```

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 << 4)
```

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,const GrTextOption *opt);
void GrDrawString(void *text,int length,int x,int y,const GrTextOption *opt);
```

NOTE: text drawing is fastest when 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, while in any other case the text is drawn pixel-by-pixel by the higher-level code.

There are pattern filed versions too:

```
void GrPatternDrawChar(int chr,int x,int y,const GrTextOption *opt,GrPattern *p);
void GrPatternDrawString(void *text,int length,int x,int y,const GrTex-
tOption *opt,
                         GrPattern *p);
void GrPatternDrawStringExt(void *text,int length,int x,int y,
                            const GrTextOption *opt,GrPattern *p);
```

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 text direction specified in the text option structure passed to them.

```
int  GrFontCharPresent(const GrFont *font,int chr);
int  GrFontCharWidth(const GrFont *font,int chr);
int  GrFontCharHeight(const GrFont *font,int chr);
int  GrFontCharBmpRowSize(const GrFont *font,int chr);
int  GrFontCharBitmapSize(const GrFont *font,int chr);
int  GrFontStringWidth(const GrFont *font,void *text,int len,int type);
int  GrFontStringHeight(const GrFont *font,void *text,int len,int type);
int  GrProportionalTextWidth(const GrFont *font,void *text,int len,int type);
int  GrCharWidth(int chr,const GrTextOption *opt);
int  GrCharHeight(int chr,const GrTextOption *opt);
void GrCharSize(int chr,const GrTextOption *opt,int *w,int *h);
int  GrStringWidth(void *text,int length,const GrTextOption *opt);
int  GrStringHeight(void *text,int length,const GrTextOption *opt);
void GrStringSize(void *text,int length,const GrTextOption *opt,int *w,int *h);
```

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 {                    /* fixed font text window desc. */
  struct _GR_font     *txr_font;     /* font to be used */
  union  _GR_textColor txr_fgcolor;  /* foreground color */
  union  _GR_textColor txr_bgcolor;  /* background color */
  void   *txr_buffer;                /* pointer to text buffer */
  void   *txr_backup;                /* optional backup buffer */
  int     txr_width;                 /* width of area in chars */
  int     txr_height;                /* height of area in chars */
  int     txr_lineoffset;            /* offset in buffer(s) between rows */
  int     txr_xpos;                  /* upper left corner X coordinate */
```

```
  int     txr_ypos;                   /* upper left corner Y coordinate */
  char    txr_chrtype;                /* character type (see above) */
} GrTextRegion;

void GrDumpChar(int chr,int col,int row,const GrTextRegion *r);
void GrDumpText(int col,int row,int wdt,int hgt,const GrTextRegion *r);
void GrDumpTextRegion(const 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. With these functions you can simulate a text mode window, write chars directly to the txr_buffer and call GrDumpTextRegion when you want to update the window (or GrDumpText if you know the area to update is small).

See the **test/fonttest.c** example.

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,GrColor c);
void GrUsrLine(int x1,int y1,int x2,int y2,GrColor c);
void GrUsrHLine(int x1,int x2,int y,GrColor c);
void GrUsrVLine(int x,int y1,int y2,GrColor c);
void GrUsrBox(int x1,int y1,int x2,int y2,GrColor c);
void GrUsrFilledBox(int x1,int y1,int x2,int y2,GrColor c);
void GrUsrFramedBox(int x1,int y1,int x2,int y2,int wdt,GrFBoxColors *c);
void GrUsrCircle(int xc,int yc,int r,GrColor c);
void GrUsrEllipse(int xc,int yc,int xa,int ya,GrColor c);
void GrUsrCircleArc(int xc,int yc,int r,int start,int end,
                    int style,GrColor c);
void GrUsrEllipseArc(int xc,int yc,int xa,int ya,int start,int end,
                     int style,GrColor c);
void GrUsrFilledCircle(int xc,int yc,int r,GrColor c);
void GrUsrFilledEllipse(int xc,int yc,int xa,int ya,GrColor c);
void GrUsrFilledCircleArc(int xc,int yc,int r,int start,int end,
                          int style,GrColor c);
void GrUsrFilledEllipseArc(int xc,int yc,int xa,int ya,int start,int end,
                           int style,GrColor c);
void GrUsrPolyLine(int numpts,int points[][2],GrColor c);
void GrUsrPolygon(int numpts,int points[][2],GrColor c);
void GrUsrFilledConvexPolygon(int numpts,int points[][2],GrColor c);
void GrUsrFilledPolygon(int numpts,int points[][2],GrColor c);
void GrUsrFloodFill(int x, int y, GrColor border, GrColor c);
GrColor GrUsrPixel(int x,int y);
GrColor GrUsrPixelC(GrContext *c,int x,int y);
void GrUsrCustomLine(int x1,int y1,int x2,int y2,const GrLineOption *o);
void GrUsrCustomBox(int x1,int y1,int x2,int y2,const GrLineOption *o);
void GrUsrCustomCircle(int xc,int yc,int r,const GrLineOption *o);
void GrUsrCustomEllipse(int xc,int yc,int xa,int ya,const GrLineOption *o);
void GrUsrCustomCircleArc(int xc,int yc,int r,int start,int end,
                          int style,const GrLineOption *o);
void GrUsrCustomEllipseArc(int xc,int yc,int xa,int ya,int start,int end,
                           int style,const GrLineOption *o);
void GrUsrCustomPolyLine(int numpts,int points[][2],const GrLineOption *o);
void GrUsrCustomPolygon(int numpts,int points[][2],const GrLineOption *o);
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,
                             int style,GrLinePattern *lp);
void GrUsrPatternedEllipseArc(int xc,int yc,int xa,int ya,int start,int end,
                              int style,GrLinePattern *lp);
```

```
void GrUsrPatternedPolyLine(int numpts,int points[][2],GrLinePattern *lp);
void GrUsrPatternedPolygon(int numpts,int points[][2],GrLinePattern *lp);
void GrUsrPatternFilledPlot(int x,int y,GrPattern *p);
void GrUsrPatternFilledLine(int x1,int y1,int x2,int y2,GrPattern *p);
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,int style,GrPat
void GrUsrPatternFilledEllipseArc(int xc,int yc,int xa,int ya,int start,int end,int st
void GrUsrPatternFilledConvexPolygon(int numpts,int points[][2],GrPattern *p);
void GrUsrPatternFilledPolygon(int numpts,int points[][2],GrPattern *p);
void GrUsrPatternFloodFill(int x, int y, GrColor border, GrPattern *p);
void GrUsrDrawChar(int chr,int x,int y,const GrTextOption *opt);
void GrUsrDrawString(char *text,int length,int x,int y,const GrTextOption *opt);
void GrUsrTextXY(int x,int y,char *text,GrColor fg,GrColor 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. Cursors are created with the GrBuildCursor function:

```
GrCursor *GrBuildCursor(char far *pixels,int pitch,int w,int h,
                        int xo,int yo,const GrColorTableP c);
```

The pixels, 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.), but with two differences. First, is not assumed that the pixels data is w x h sized, the pitch argument set the offset between rows. Second, 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 _GR_cursor {
  struct _GR_context work;            /* work areas (4) */
  int     xcord,ycord;                /* cursor position on screen */
  int     xsize,ysize;                /* cursor size */
  int     xoffs,yoffs;                /* LU corner to hot point offset */
  int     xwork,ywork;                /* save/work area sizes */
  int     xwpos,ywpos;                /* save/work area position on screen */
  int     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 GrDisplayCursor(GrCursor *cursor);
void GrEraseCursor(GrCursor *cursor);
void GrMoveCursor(GrCursor *cursor,int x,int y);
void GrDestroyCursor(GrCursor *cursor);
```

See the **test/curstest.c** example.

Keyboard input

GRX can handle platform independant key input. The file grxkeys.h defines the keys to be used in the user's program. This is an extract:

```
#define GrKey_Control_A            0x0001
#define GrKey_Control_B            0x0002
#define GrKey_Control_C            0x0003
...
#define GrKey_A                    0x0041
#define GrKey_B                    0x0042
#define GrKey_C                    0x0043
...
#define GrKey_F1                   0x013b
#define GrKey_F2                   0x013c
#define GrKey_F3                   0x013d
...
#define GrKey_Alt_F1               0x0168
#define GrKey_Alt_F2               0x0169
#define GrKey_Alt_F3               0x016a
```

But you can be confident that the standard ASCII is right maped. The GrKeyType type is defined to store keycodes:

```
typedef unsigned short GrKeyType;
```

This function:

```
int GrKeyPressed(void);
```

returns non zero if there are any keycode waiting, that can be read with:

```
GrKeyType GrKeyRead(void);
```

The function:

```
int GrKeyStat(void);
```

returns a keyboard status word, or-ing it with the next defines it can be known the status of some special keys:

```
#define GR_KB_RIGHTSHIFT    0x01      /* right shift key depressed */
#define GR_KB_LEFTSHIFT     0x02      /* left shift key depressed */
#define GR_KB_CTRL          0x04      /* CTRL depressed */
#define GR_KB_ALT           0x08      /* ALT depressed */
#define GR_KB_SCROLLOCK     0x10      /* SCROLL LOCK active */
```

```
#define GR_KB_NUMLOCK       0x20      /* NUM LOCK active */
#define GR_KB_CAPSLOCK      0x40      /* CAPS LOCK active */
#define GR_KB_INSERT        0x80      /* INSERT state active */
#define GR_KB_SHIFT         (GR_KB_LEFTSHIFT | GR_KB_RIGHTSHIFT)
```

See the **test/keys.c** example.

Mouse event handling

All mouse services need the presence of a mouse. An application can test whether a mouse is available by calling the function:

```
int  GrMouseDetect(void);
```

which will return zero if no mouse (or mouse driver) is present, non-zero otherwise. The mouse must be initialized by calling one (and only one) of these functions:

```
void GrMouseInit(void);
void GrMouseInitN(int queue_size);
```

GrMouseInit sets a event queue (see below) size to GR_M_QUEU_SIZE (128). A user supply event queue size can be set calling GrMouseInitN instead.

It is a good practice to call GrMouseUnInit before exiting the program. This will restore any interrupt vectors hooked by the program to their original values.

```
void GrMouseUnInit(void);
```

The mouse can be controlled with the following functions:

```
void GrMouseSetSpeed(int spmult,int spdiv);
void GrMouseSetAccel(int thresh,int accel);
void GrMouseSetLimits(int x1,int y1,int x2,int y2);
void GrMouseGetLimits(int *x1,int *y1,int *x2,int *y2);
void GrMouseWarp(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 some mouse driver when it finds itself in a graphics mode unknown to it.) The parameters to the GrMouseSetSpeed function specify how coordinate changes are obtained from mickey counter changes, multipling by spmult and dividing by spdiv. In high resolution graphics modes the value of one just works fine, in low resolution modes (320x200 or similar) it is best set the spdiv to two or three. (Of course, it also depends on the sensitivity the mouse.) The GrMouseSetAccel 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 GrMouseSetLimits (default is the whole screen) and the current limits can be obtained with GrMouseGetLimits. GrMouseWarp 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 GrMouseSetCursor(GrCursor *cursor);
void GrMouseSetColors(GrColor fg,GrColor bg);
```

GrMouseSetColors 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 *GrMouseGetCursor(void);
```

The mouse cursor can be displayed/erased with:

```
void GrMouseDisplayCursor(void);
void GrMouseEraseCursor(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  GrMouseBlock(GrContext *c,int x1,int y1,int x2,int y2);
void GrMouseUnBlock(int return_value_from_GrMouseBlock);
```

GrMouseBlock 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. When the drawing is finished GrMouseUnBlock must be called with the argument returned by GrMouseBlock.

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

```
int  GrMouseCursorIsDisplayed(void);
```

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

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

The parameter mode can have the following values:

```
#define GR_M_CUR_NORMAL   0    /* MOUSE CURSOR modes: just the cursor */
#define GR_M_CUR_RUBBER   1    /* rect. rubber band (XOR-d to the screen) */
#define GR_M_CUR_LINE     2    /* line attached to the cursor */
#define GR_M_CUR_BOX      3    /* rectangular box dragged by the cursor */
```

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

```
GrMouseSetCursorMode(M_CUR_NORMAL);
GrMouseSetCursorMode(M_CUR_RUBBER,xanchor,yanchor,GrColor);
GrMouseSetCursorMode(M_CUR_LINE,xanchor,yanchor,GrColor);
GrMouseSetCursorMode(M_CUR_BOX,dx1,dy1,dx2,dy2,GrColor);
```

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 GrMouseSetCursorMode.

The GrMouseGetEvent 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 GrMouseEvent structure. The relevant declarations from grx20.h:

```
void GrMouseGetEvent(int flags,GrMouseEvent *event);

typedef struct _GR_mouseEvent {    /* mouse event buffer structure */
  int  flags;                       /* event type flags (see above) */
  int  x,y;                         /* mouse coordinates */
  int  buttons;                     /* mouse button state */
  int  key;                         /* key code from keyboard */
  int  kbstat;                      /* keybd status (ALT, CTRL, etc..) */
  long dtime;                       /* time since last event (msec) */
} GrMouseEvent;
```

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 which can be used to check for double clicks, etc... The following macros have been defined in grx20.h to help in creating the control flag for GrMouseGetEvent and decoding the various bits in the event structure:

```
#define GR_M_MOTION         0x001         /* mouse event flag bits */
#define GR_M_LEFT_DOWN      0x002
#define GR_M_LEFT_UP        0x004
#define GR_M_RIGHT_DOWN     0x008
#define GR_M_RIGHT_UP       0x010
#define GR_M_MIDDLE_DOWN    0x020
#define GR_M_MIDDLE_UP      0x040
#define GR_M_BUTTON_DOWN    (GR_M_LEFT_DOWN | GR_M_MIDDLE_DOWN | \
                             GR_M_RIGHT_DOWN)
#define GR_M_BUTTON_UP      (GR_M_LEFT_UP   | GR_M_MIDDLE_UP   | \
                             GR_M_RIGHT_UP)
#define GR_M_BUTTON_CHANGE  (GR_M_BUTTON_UP | GR_M_BUTTON_DOWN )

#define GR_M_LEFT           1             /* mouse button index bits */
#define GR_M_RIGHT          2
#define GR_M_MIDDLE         4

#define GR_M_KEYPRESS       0x080        /* other event flag bits */
#define GR_M_POLL           0x100
#define GR_M_NOPAINT        0x200
#define GR_COMMAND          0x1000
#define GR_M_EVENT          (GR_M_MOTION | GR_M_KEYPRESS | \
                             GR_M_BUTTON_CHANGE | GR_COMMAND)
```

GrMouseGetEvent will display the mouse cursor if it was previously erased and the GR_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.

GrMouseGetEvent block until a event is produced, except if the GR_M_POLL bit is set in the flag passed to it.

Another version of GetEvent:

```
void GrMouseGetEventT(int flags,GrMouseEvent *event,long timout_msecs);
```

can be istructed to wait timout_msec for the presence of an event. Note that event->dtime is only valid if any event occured (event->flags != 0) otherwise it's set as -1. Additionally event timing is real world time even in X11 && Linux.

If there are one or more events waiting the function:

```
int  GrMousePendingEvent(void);
```

returns non-zero value.

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 GrMouseEventEnable(int enable_kb,int enable_ms);
```

Note that GrMouseInit set both by default. If you want to use GrMouseGetEvent and GrKeyRead at the same time, a call to GrMouseEventEnable(0,1) is needed before input process.

See the **test/mousetst.c** example.

Writing/reading PNM graphics files

GRX includes functions to load/save a context from/to a PNM file.

PNM is a group of simple graphics formats from the NetPbm distribution. NetPbm can convert from/to PNM lots of graphics formats, and apply some transformations to PNM files. (Note. You don't need the NetPbm distribution to use this functions).

There are six PNM formats:

```
P1 text PBM (bitmap)
P2 text PGM (gray scale)
P3 text PPM (real color)
P4 binary PBM (bitmap)
P5 binary PGM (gray scale)
P6 binary PPM (real color)
```

GRX can handle the binary formats only (get the NetPbm distribution if you need convert text to binary formats).

To save a context in a PNM file you have three functions:

```
int GrSaveContextToPbm( GrContext *grc, char *pbmfn, char *docn );
int GrSaveContextToPgm( GrContext *grc, char *pgmfn, char *docn );
int GrSaveContextToPpm( GrContext *grc, char *ppmfn, char *docn );
```

they work both in RGB and palette modes, grc must be a pointer to the context to be saved, if it is NULL the current context is saved; p-mfn is the file name to be created and docn is an optional text comment to be written in the file, it can be NULL. Three functions return 0 on succes and -1 on error.

GrSaveContextToPbm dumps a context in a PBM file (bitmap). If the pixel color isn't Black it asumes White.

GrSaveContextToPgm dumps a context in a PGM file (gray scale). The colors are quantized to gray scale using .299r + .587g + .114b.

GrSaveContextToPpm dumps a context in a PPM file (real color). To load a PNM file in a context you must use:

```
int GrLoadContextFromPnm( GrContext *grc, char *pnmfn );
```

it support reading PBM, PGM and PPM binary files. grc must be a pointer to the context to be written, if it is NULL the current context is used; p-mfn is the file name to be read. If context dimensions are lesser than pnm dimensions, the function loads as much as it can. If color mode is not in RGB mode, the routine allocates as much colors as it can. The function returns 0 on succes and -1 on error.

To query the file format, width and height of a PNM file you can use:

```
int GrQueryPnm( char *ppmfn, int *width, int *height, int *maxval );
```

pnmfn is the name of pnm file; width returns the pnm width; height returns the pnm height; maxval returns the max color component value. The function returns 1 to 6 on success (the PNM format) or -1 on error.

The two next functions:

```
int GrLoadContextFromPnmBuffer( GrContext *grc, const char *pnmbuf );
int GrQueryPnmBuffer( const char *pnmbuf, int *width, int *height, int *max-
val );
```

work like GrLoadContextFromPnm and GrQueryPnmBuffer, but they get his input from a buffer instead of a file. This way, pnm files can be embeded in a program (using the bin2c program by example).

Writing/reading PNG graphics files

GRX includes functions to load/save a context from/to a png file. But note, for this purpose it needs the libpng library, and to enable the png support before make the GRX lib.

Use next function to save a context in a PNG file:

```
int GrSaveContextToPng( GrContext *grc, char *pngfn );
```

it works both in RGB and palette modes, grc must be a pointer to the context to be saved, if it is NULL the current context is saved; pngfn is the file name to be created. The function returns 0 on succes and -1 on error.

To load a PNG file in a context you must use:

```
int GrLoadContextFromPng( GrContext *grc, char *pngfn, int use_alpha );
```

grc must be a pointer to the context to be written, if it is NULL the current context is used; pngfn is the file name to be read; set use_alpha to 1 if you want to use the image alpha channel (if available). If context dimensions are lesser than png dimensions, the function loads as much as it can. If color mode is not in RGB mode, the routine allocates as much colors as it can. The function returns 0 on succes and -1 on error.

To query the width and height of a PNG file you can use:

```
int GrQueryPng( char *pngfn, int *width, int *height );
```

pngfn is the name of png file; width returns the png width; height returns the png height. The function returns 0 on success or -1 on error.

The function:

```
int GrPngSupport( void );
```

returns 1 if there is png support in the library, 0 otherwise. If there is not support for png, dummy functions are added to the library, returning error (-1) ever.

Writing/reading PNG graphics files

GRX includes functions to load/save a context from/to a jpeg file. But note, for this purpose it needs the libjpeg library, and to enable the jpeg support before make the GRX lib.

Use next function to save a context in a JPEG file:

```
int GrSaveContextToJpeg( GrContext *grc, char *jpegfn, int quality );
```

it works both in RGB and palette modes, grc must be a pointer to the context to be saved, if it is NULL the current context is saved; jpegfn is the file name to be created; quality is a number between 1 and 100 to drive the compression quality, use higher values for better quality (and bigger files), you can use 75 as a standard value, normally a value between 50 and 95 is good. The function returns 0 on succes and -1 on error.

This function saves a context in a grayscale JPEG file:

```
int GrSaveContextToGrayJpeg( GrContext *grc, char *jpegfn, int quality );
```

parameters and return codes are like in GrSaveContextToJpeg. The colors are quantized to gray scale using .299r + .587g + .114b.

To load a JPEG file in a context you must use:

```
int GrLoadContextFromJpeg( GrContext *grc, char *jpegfn, int scale );
```

grc must be a pointer to the context to be written, if it is NULL the current context is used; jpegfn is the file name to be read; set scale to 1, 2, 4 or 8 to reduce the loaded image to 1/1, 1/2, 1/4 or 1/8. If context dimensions are lesser than jpeg dimensions, the function loads as much as it can. If color mode is not in RGB mode, the routine allocates as much colors as it can. The function returns 0 on succes and -1 on error.

To query the width and height of a JPEG file you can use:

```
int GrQueryJpeg( char *jpegfn, int *width, int *height );
```

jpegfn is the name of jpeg file; width returns the jpeg width; height returns the jpeg height. The function returns 0 on success or -1 on error.

The function:

```
int GrJpegSupport( void );
```

returns 1 if there is jpeg support in the library, 0 otherwise. If there is not support for jpeg, dummy functions are added to the library, returning error (-1) ever.

Miscellaneous functions

Here we will describe some miscellaneous functions.

```
unsigned GrGetLibraryVersion(void);
```

GrGetLibraryVersion returns the GRX version API, like a hexadecimal coded number. By example 0x0241 means 2.4.1 Because grx20.h defines the GRX_VERSION_API macro, you can check if both, the library and the include file, are in the same version using if(GrGetLibraryVersion() == GRX_VERSION_API)

```
unsigned GrGetLibrarySystem(void);
```

This functions returns a unsigned integer identifing the system you are working in. grx20.h defines some macros you can use:

```
/* these are the supported configurations: */
#define GRX_VERSION_TCC_8086_DOS        1   /* also works with BCC */
#define GRX_VERSION_GCC_386_DJGPP       2   /* DJGPP v2 */
#define GRX_VERSION_GCC_386_LINUX       3   /* the real stuff */
#define GRX_VERSION_GENERIC_X11         4   /* generic X11 version */
#define GRX_VERSION_WATCOM_DOS4GW       5   /* GS - Watcom C++ 11.0 32 Bit
#define GRX_VERSION_GCC_386_WIN32       7   /* WIN32 using Mingw32 */
#define GRX_VERSION_MSC_386_WIN32       8   /* WIN32 using MS-VC */
#define GRX_VERSION_GCC_386_CYG32       9   /* WIN32 using CYGWIN */
#define GRX_VERSION_GCC_386_X11        10   /* X11 version */
#define GRX_VERSION_GCC_X86_64_LINUX   11   /* console framebuffer 64 */
#define GRX_VERSION_GCC_X86_64_X11     12   /* X11 version 64 */
```

Note. On Linux, GrGetLibrarySystem returns GRX_VERSION_GCC_386_LINUX even in the X11 version.

```
void GrSetWindowTitle(char *title);
```

GrSetWindowTitle sets the main window title in the X11 an Win32 versions. It doesn't do nothing in the DOS and Linux-SvgaLib versions.

```
void GrSleep(int msec);
```

This function stops the program execution for msec miliseconds.

```
GrContext *GrCreateFrameContext(GrFrameMode md,int w,int h,
           char far *memory[4],GrContext *where);
```

This function is like GrCreateContext, except that you can specify any valid memory frame mode, not only the Screen associated frame mode. It can be used for special purposes (see GrBitBlt1bpp for an example).

```
void GrBitBlt1bpp(GrContext *dst,int dx,int dy,GrContext *src,
     int x1,int y1,int x2,int y2,GrColor fg,GrColor bg);
```

This special function does a bitblt from a 1bpp context (a bitmap really), using fg and bg like the color+opcode when bit=1 and bit=0 respectively. Here is an example:

```
pContext = GrCreateFrameContext(GR_frameRAM1, sizex, sizey, NULL, NULL);
/* draw something (black and white) into the bitmap */
GrSetContext(pContext);
GrClearContext( GrBlack() );
GrLine(0, 0, sizex-1, sizey-1, GrWhite());
GrLine(0, sizey-1, sizex-1, 0, GrWhite());

/* Put the bitmap into the screen */
GrSetContext(NULL);
fcolor = GrAllocColor( 255,0,0 );
bcolor = GrAllocColor( 0,0,255 );
```

```
GrBitBlt1bpp(NULL,x,y,pContext,0,0,sizex-1,sizey-1,fcolor,bcolor);
```

BGI interface

From the 2.3.1 version, GRX includes the BCC2GRX library created by Hartmut Schirmer. The BCC2GRX was created to allow users of GRX to compile graphics programs written for Borland-C++ and Turbo-C graphics interface. BCC2GRX is not a convenient platform to develop new BGI programs. Of course you should use native GRX interface in such cases!

Read the readme.bgi file for more info.

Pascal interface

The Pascal (gpc) support is produced by two unit files **pascal/grx.pas** and **pascal/bgi/graph.pas** which are the Pascal translations of the C header files **include/grx20.h + include/grxkeys.h** and **include/libbcc.h**.

Compilation of the examples and installation of the header files is allowed by setting INCLUDE_GPC_SUPPORT=y in makedef.grx.

The unit files contain at the beginning instructions to load the required libraries (libgrx20..., depending on the system) and addon libraries (e.g. libpng). You can uncomment manually the addons you want. You can also use the configure script which does that automatically, together with editing the makedefs.grx file.

By default they are installed in a **units** directory below the INSTALLDIR directory. But you can put them where you like.

References

Official GRX site	http://grx.gnu.de
GRX mailing list archive	http://grx.gnu.de/archive/grx/en/
MGRX site (fork from GRX)	http://mgrx.fgrim.com
NetPbm distribution	http://netpbm.sourceforge.net
PNG library	http://www.libpng.org/pub/png/libpng.html
JPEG library	http://www.ijg.org
TIFF library	http://www.remotesensing.org/libtiff/

Table of Contents

Abstract 3

GRX2 User's Manual 5

Hello world 5
Data types and function declarations 6
Setting the graphics driver 6
Setting video modes 6
Graphics contexts 9
Context use 12
Color management 12
Portable use of a few colors 14
Graphics primitives 15
Non-clipping graphics primitives 18
Customized line drawing 18
Pattern filled graphics primitives 19
Patterned line drawing 22
Image manipulation 22
Text drawing 23
Drawing in user coordinates 28
Graphics cursors 30
Keyboard input 31
Mouse event handling 32
Writing/reading PNM graphics files 35
Writing/reading PNG graphics files 36
Writing/reading PNG graphics files 37
Miscellaneous functions 37
BGI interface 39
Pascal interface 39
References 39