function libraries
disassemblers
compilers
text editors
text filters
communications support
text formatters
interpreters
bulletin boards
co-routines
compiler compilers
window packages
assemblers
games
tutorials
math packages
link editors
languages
cross compilers
preprocessors
function libraries
disassemblers
compilers
text editors
text filters
communications support
text formatters
interpreters
bulletin boards

Volume II

The Users' Group Library

Disks 200–249

A Directory Of User-Supported C Source Code

Edited By
Robert Ward
And Kenji Hino

Portable Graphics Package For PC-Clones Contributed By Member In West Germany

By Rainer Gerhards

[**Editor's Note:** The following article was adapted from documentation available on the disk.]

Introduction

This library was first designed as a demonstration version of a very fast assembly language package. All procedures were first written in C and then converted to 8086 assembly language. The assembly language version is to be a commercial project, but the basic graphics routines are available in C as CUG227. In part, because the C version offers some excellent examples of C coding, I am placing it in the public domain.

But this is not just a demonstration library. This C version has some features which the assembly language library lacks. This C version, for example, supports world coordinates and a hardware-independent kernel.

This C library forms the base of a highly portable graphics system, which will run on a CP/M machine as well as on a big UNIX system. Currently it has been tested with only certain IBM hardware, because I don't own a CP/M or UNIX system. The current version compiles under both Lattice C v4.0 and Datalight v2.23, although the assembly modules do not support Datalight's macros. I hope, that other users will help refine the package, to make it really portable and powerful.

The current version is only a base, really incomplete. I put this version in the public domain because I hope that some of you have some interesting ideas and algorithms and want to tell me about them. So I hope to get a full praxis-driven library. In addition, I hope that some of you want to port the low-level parts of the library to different machines and graphics environments, making the library really portable. Everybody should feel free to contact me!

Current Features

The most powerful feature of the library is its portability and hardware independence. You may use the library on many machines, as long as an appropriate driver is provided.

The library currently supports:

- switching between different display modes (graphics/nongraphics).
- drawing pixels.
- drawing lines.
- drawing boxes.
- filling boxes with given colors (e.g., clear windows).
- drawing full and partial ellipses.
- painting any region on the screen with a given color.
- specification of objects in a global coordinate system, freeing you from the given hardware parameters. Also very useful in numerical applications.
- printing graphics screen as a hardcopy function. This is currenly limited to the Hercules graphics card and Epson compatible printers.
- clearing the graphics screen.
- reading the color of a specified pixel.

There is currently no function to display text on the graphics screen. This is a great disadvantage, but the function will be implemented in the near future. In the meantime you may use BIOS services. Unfortunately, BIOS services are restricted only to the official video-modes and so cannot be used with the Hercules graphics card. Also drivers for hardware other than PC-clones are not supplied.

Library Design

The library is partitioned into two logical parts: a high-level, machine independent graphics kernel and a low-level machine dependent part. There are header (*.h*) files for each part. The application program doesn't need to know which function resides in which part of the graphics library. If it uses the official entry points, it will be portable to any new video device that is supported by the library. The available entry points are listed under Available Functions.

High-level Functions

The high-level part is written completely in C. It includes the graphics primitives such as line drawing, and area filling. This part uses no direct hardware parameters or interfaces. Instead it calls upon low-level, hardware-dependent functions through standardized interfaces. If you need special coding for special hardware or greater speed, you may include this coding in the high-level part, but use conditional compilation to keep it portable. Though this part is hardware independent, it needs to be recompiled for every new video device. The high-level part must know some common hardware parameters like the screen resolution. The low-level part must include specialized code for each different device.

To perform this recompilation successfully the high-level routines use the information contained in the low-level header files. So it is very important that every new device define its hardware parameters in its corresponding header file. A unique mode-identifier must also be defined for every device. This identifier allows the high-level routines (and the user's code!) to use special hardware features but remain portable.

Low-level Library Part

The low-level part is written in C and assembler. This C coding uses non-standard functions (such as *import()* and *outport()*) to perform its work. Because of this, recompiling this portion under different compilers may be difficult. Only tasks, which couldn't be performed in C are written in assembler. The most commonly called assembly functions are read pixel and write pixel.

The low-level functions are called very often from the high-level routines, especially the read and write pixel functions. To improve throughput, these functions should be optimized. The current low-level drivers aren't optimal, but they are a good starting point for future improvements.

User Program Interface

The user program interface is defined in the various header (*.h*) files. There are two classes of header file: first one (and only one) header file defining the high-level functions and constants. This header file is named *graphlib.h*. Second there are many header files defining the low-level functions and constants.

These low-level header files may be used in two different ways: First you may include the device independent header files by name, e.g. *hercgraf.h* for the Hercules graphics card. This has the disadvantage of requiring hardware dependent include statements in the user code. If you want to move to a different video device, you must first change the header file names in all of your source files.

The preferred way to use the header files is to include a generic filename, e.g. *graf.h*. You may then copy the correct header file (e.g., *hercgraf.h*) to this generic file. If you want to switch to another video device, just copy the new header file to the generic file and recompile. There is nothing more to do; no code must be changed!

Development Environment

The library was implemented on two IBM-AT's. One is equipped with a Hercules-compatible monochrome graphics card, one with the Quad-EGA and a monochrome display. The package hasn't been tested with a standard color-graphics adapter, but I think the EGA-routines will run on the standard CG card.

Both AT's are equipped with 512 Kb memory without an 80287. Both are running under PC-DOS v3.1 (German version). One machine also runs XENIX. The package has been tested with some resident programs loaded. The software environment consists of *sh* and related utilities by Allen Holub, Lattice C compiler and Microsoft MASM v3.0 and related products. The files were edited using the Turbo Pascal editor.

My own (new) AT is equipped with 512 Kb, no 80287 and the Quad-EGA. I currently only own the sh and utilities, the Datalight C compiler, the Blaise runoff formatter and the standard IBM PC linker.

Planned Enhancements

Below I list some enhancements I am currently working on or plan to implement in the future. In addition to sharing these ideas, I want to get some feedback from you, resulting in a better and better library. This feedback may be in the form of new ideas, algorithms and, of course, critique. I can apply the library in only a restricted set of applications. Thus, without your feedback, my library would always be restricted to my needs.

But enough about feedback, here are the facts about future enhancements:

Better low-level drivers. The current low-level drivers aren't really optimal. Optimizing these drivers will speed up the whole library.

Text output. The text-output function will provide italics as well as boldface. Different character resolutions will be available. I would even like to include loadable character sets.

Virtual graphics pages will be implemented. Multiple display pages will be available even on devices which do not directly support them in hardware. On memory-limited systems, inactive pages will be swapped to a mass-storage device.

An implementation for XENIX. I'm not very familiar with the XENIX operating system, nor with graphics programming using XENIX and graphics terminals. In the first phase I will try to port the library, making it functional only at the XENIX console (where I can still rely on BIOS calls). If this port is successful, I will attempt to implement support for other terminals.

(Help! If someone has some information about this job, please write me. I will greatly appreciate any help you can contribute.)

Available Function Calls

The following is a brief description of the available function calls. These functions will work in every implementation. In the current pre-version of the graphics library there may be some video device drivers which do not support all functions. This is explicitly noted and will be corrected in the near future. Each function is listed with its name, parameters and a description.

•box

```
box (x1, y1, x2, y2, color);
int x1, y1; upper left corner
int x2, y2; lower right corner
int color; border color
```

This function draws a border of *color* covering the given box. The box is specified through the upper left (x1, y1) and to lower right (x2, y2) corner. The box itself isn't modified (e.g., must be cleared explicitly).

•circle

```
circle (x, y, radius, color, aspect);
int x, y; center coordinate;
int radius; circle radius;
int color; circle color float aspect;
aspect ratio;
```

This function draws a circle.

•convxco

```
locx = convxco(x) float x;
      x coordinate to convert
int locx; hardware x coordinate
```

This function converts a global coordinate to a local, hardware dependent coordinate. Its return value may be directly passed to other functions.

•convyco

```
locy = convyco(y) float y;
      y coordinate to convert
int locy; hardware y coordinate
```

This function converts a global coordinate to a local, hardware dependent coordinate. Its return value may be directly passed to other functions.

•ellipsis

```
ellipsis (x, y, rx, ry, ws, we, color)
int x, y; center coordinate
int rx; x - 'radius'
int ry; y - 'radius'
int ws; begin angle 0..360
int we; end angle 0..360
int color; line color
```

This function draws any sort of ellipse. It is often called circle, but I think this name should better be reserved for a function, which only draws a full circle (see above). This function may not only draw a circle or any possible ellipse, it is also capable of drawing only parts of them. This feature is often used in pie-charts. Because of its great flexibility, this function is much slower than circle. If you only want a full circle (or ellipse) you should call *circle*.

•fillbox

```
fillbox (x1, y1, x2, y2, color);
int x1, y1; upper left corner
int x2, y2; lower right corner
int color; fill color
```

This function fills a given box with the specified color. The box is specified through the upper left (x1, y1) and the lower right (x2, y2) corner. This function is the counterpart to box, which draws the border.

•get pixel

```
color = getpixel (x, y);
int x, y; coordinate of the pixel
```



```
int color; returned color of that
```

This function reads (gets) the color of a spec

•line

```
line (x1, y1, x2, y2, color);
int x1, y1; starting coordinate
int x2, y2; ending coordinate
int color; line color
```

This function draws a line of color between
dinate.

•paint

```
paint (x, y, paintcir, border)
int x, y; coordinate of a point
      within the area
int paintcir; the color used to
int border; is the color of the
      border defining the are
```

This function paints an area. The area is de
and the coordinate of one pixel within the area
paintclr. This function uses several subroutine
tializer—the main work is performed by the ot

•print_screen_function_-_hardcopy

Allows a program-controlled hardcopy of t
Warning! This function is currently only ava
video devices will be added in the future.

•prtgraf

```
prtgraf();
```

This function prints the entire graphics-scre
developed using a NEC P6 printer, but should
patible printer.
Warning: this function depends on hardwa

•setgloco

```
setgloco (x_beg, y_beg, x_end, y
float x_beg, y_beg; minimum
      coordinates values
float x_end, y_end; maximum
```

hardware dependent coordinate. Its

hardware dependent coordinate. Its

ed circle, but I think this name should
l circle (see above). This function may
apable of drawing only parts of them.
at flexibility, this function is much
e) you should call *circle*.

r. The box is specified through the
is function is the counterpart to box,


```
int color; returned color of that pixel
```

This function reads (gets) the color of a specified pixel.

•line

```
line (x1, y1, x2, y2, color);
int x1, y1; starting coordinate
int x2, y2; ending coordinate
int color; line color
```

This function draws a line of color between the starting (x1, y1) and ending (x2, y2) coordinate.

•paint

```
paint (x, y, paintcir, border)
int x, y; coordinate of a point
        within the area
int paintcir; the color used to paint
int border; is the color of the
        border defining the area
```

This function paints an area. The area is defined by a border of a specified color (*border*) and the coordinate of one pixel within the area (x, y). The color used to paint is given in *paintclr*. This function uses several subroutines and a recursive algorithm! It's only the initializer—the main work is performed by the other routines.

•print_screen_function_-_hardcopy

Allows a program-controlled hardcopy of the current graphics page.
Warning! This function is currently only available for the Hercules graphics card. Other video devices will be added in the future.

•prtgraf

```
prtgraf();
```

This function prints the entire graphics-screen on a dot-matrix printer. The function was developed using a NEC P6 printer, but should run with little alteration on every Epson compatible printer.
Warning: this function depends on hardware parameters!

•setgloco

```
setgloco (x_beg, y_beg, x_end, y_end)
float x_beg, y_beg; minimum
        coordinates values
float x_end, y_end; maximum
```

```
coordinates values
```

This function initializes the global/local coordinate system. This system allows you to address your pixels based on a global coordinate system. This system is independent from the hardware coordinate system. Global coordinates may be converted to local coordinates, which are to be used to address the hardware. So your application needn't look at the present video hardware but may instead use a hardware independent coordinate system. In addition you may use floats, not only integers as coordinates. This is a great advantage in numerical applications.

Caution: This function initializes the system, so any call to the convert routines will return garbage, until this function is called!

•setpixel

```
setpixel (x, y, color);
int x, y; coordinate of the pixel
int color; pixel-color
```

This function sets a pixel of color *color* at the specified coordinate.