The picoTK HOWTO

Thomas Gallenkamp, tgkamp@users.sourceforge.net $Id: HOWTO.sgml,v 1.4 2003/01/11 23:57:16 tgkamp Exp $

About the installation, evaluation and use of picoTK, the small footprint GUI Toolkit for embedded systems. [This document is generated from the original sgml source in either ASCII, DVI, PS or PDF format using the sgml-tools.]

Contents

1 Introduction

As its name suggests the picoTK Toolkit is focused on small systems - where the emphasis is on simple presentation of information rather than modern interactive windowing systems. Featurewise picoTK is not comparable to modern full blown GUI interfaces like Qt/embedded or nanoX. Filling the need for many small applications where the overall overhead using more feature rich GUIs is contraproductive. picoTK has its own C programming API using only a few functions. Basically picoTK is an "output only" toolkit featuring the following:

- Draws the following primitives: points, lines, text (different fonts and sizes, fixed and proportionally spaced), filled rectangles and pixmaps.
- Drivers are supplied for 3 different color depth: 1, 4 and 8 bits per pixel, corresponding to 2, 16 and 256 simultaneusly displayable colors

- Support the following "off-the-shelf" VGA modes. Legacy modes: 640x480x1bpp and 320x200x8bpp. VESA compatible VGA-Cards supporting at least VBE2.0 and *flat linear frame buffer*: 640x400x8bpp to 1280x1024x8bpp, 800x600x4bpp to 1280x1024x4bpp.
- Universal framebuffer emulator (`fbe`) for Linux/X11 using shared memory IPC. This is used for the development of the toolkit and can be used for the development of your application. That is you can have a simulation version of your application that run under Linux - and look at its virtual screen. Than you can cross-compile the same application for the target system. The framebuffer supports arbitrary sizes (up to 640x480) and 1, 4 and 8 bpp color depth.
- Together with RTEMS (`http://www.oarcorp.com`) you can have a nice graphical output embedded system using solely GNU technology.

2 The Framebuffer

Image displays using cathode ray tubes (CRT) or the more modern flat panel displays (LCD) work by repeatedly reading out "frames" from digital memory - the framebuffer - using a controller. This is required since the display technology itself cannot store a picture a sufficiently long time for the human eye to perceive a stable picture. The picture is therefore "refreshed" periodically by reading out the framebuffer. Historically the controller for getting the data out of the framebuffer and feeding the display is called a cathode ray tube controller (CRTC). [On the first IBM PCs this has been done using a Motorola MC6845 chip (back then, this chip has been used widely for other display applications as well). Todays VGA cards still have an MC6845 compatible CRTC embedded in their chip sets.]

2.1 Organization and size of the framebuffer

The organization of the framebuffer, that is how the picture information is stored in the memory and accessed by the host CPU depends on the implementation and programming of the CRTC. picoTK relies on simple frame- buffer layouts, depending on the color depth. The supported color depths are 1, 4 and 8 bits per pixel (bpp). This equals to 1 out of 2, 16 or 256 possible colors per pixel. There is a separate driver for each of these color depths. The number of pixels and the color depth determines the size of the frame- buffer memory, see the following table:

```
Mode         size/bytes
320x240x2     9600
320x240x16   38400
640x480x16  153600
```

The framebuffer is displayed left to right, up to down. The most left pixel (either 1,4 or 8 consecutive bits) occupies the most significant bit positions in the framebuffer byte. In case of 8bpp one byte simply represents one pixel. The "pixel values" - representing distinct colors - reach from 0 to 1, 0 to 15 and 0 to 255 for 1, 4 and 8 bpp modes respectively. The actual assignment of pixel values to colors depends on how the hardware works.

2.1.1 Example: 320x240x1bpp

	base+0	base+1		base+39
Line 0	76543210	76543210	...	76543210
	base+40	base+41		base+79
Line 1	76543210	76543210	...	76543210
...				
	base+9560	base+9561		base+9599
Line 239	76543210	76543210	...	76543210

Every line occupies 40 bytes, since 1 byte carries 8 pixels. A total of 9600 bytes are required for the framebuffer memory.

3 Display hardware

For embedded PC solutions there is support for standard VGA cards. For more cost and space sensitive devices it makes sense to build your own hardware. For simulation purposes there is a X-Windows framebuffer emulation making testing of your application (and the toolkit !) much easier. The following discusses these three options in more detail.

3.1 The VGA card as framebuffer

The generic (old) VGA compatible cards can be used with the 1bpp and 8bpp modes. In these modes the host CPU has transparent access to the framebuffer, i.e. each byte from the framebuffer is directly accessible by the host CPU. Traditionally the VGA card occupies 64KB address range from 0xa0000 to 0xaffff. This limits the possible resolution to 800x600 for 1bpp modes and 320x200, for 8bpp mode. The legacy VGA modes which are present in all VGA cards practically limits the number of available modes further to the following two:

LILO-SVGA Mode Code	legacy VGA-Mode # (Int10,AH=0)	Resolution	Color depth
0x111	0x11	640 x 480	1 bpp
0x113	0x13	320 x 200	8 bpp

Fortunately modern PCI VGA cards can map a larger amount of the framebuffer into the host address space. Current VESA compatible cards, supporting VBE2.0 or above (VBE stands for VESA BIOS Extensions) typically have additional *flat linear frame buffer* modes available which can be used together with picoTK. The following table shows these VESA modes.

LILO-SVGA Mode Code	VESA VGA-Mode # (Int10,AX=4f02)	Resolution	Color depth
0x300	0x100	640 x 400	8 bpp
0x301	0x101	640 x 480	8 bpp
0x302	0x102	800 x 600	4 bpp
0x303	0x103	800 x 600	8 bpp
0x304	0x104	1024 x 768	4 bpp
0x305	0x105	1024 x 768	8 bpp
0x306	0x106	1280 x1024	4 bpp
0x307	0x107	1280 x1024	8 bpp

picoTK relies on foreign software to switch into the required VGA video mode. Together with RTEMS this is easily done using the `pc386_lilo` package found on the RTEMS homepage. LILO can set the legacy VGA-BIOS mode as well as the VESA modes before launching RTEMS. Note that not all modes are necessarily supported by each card - the LILO bootloader will stop if a specified video mode is not supported. The VESA modes have no fixed frame buffer address - as the Legacy modes have; but LILO can tell the frame buffer start to RTEMS and picoTK, which picks it up (the base address is stored in the boot segment INITSEG at 9000:0018 equalling linear 0x90018 as physical 32 bit address). For a general discussion on VGA hardware/software and the many different operating modes used here see [1] and [3]; and take a look into `pc386_lilo/video.S`.

3.2 Non VGA hardware

Non VGA hardware requires custom logic to readout the framebuffer and feed the pixel data to the display. Today this can easily be done using programmable CPLD chips. As framebuffer a static RAM can be used. The RAM is quasi simultanously addressed by the host CPU and address counters within the CPLD. Since the CPLD logic can be tailored to the applications needs, it need not to be register configurable at run-time. This has two distinct advantages: It makes the implementation of the logic easier. It removes the need for the software to initialize the framebuffer logic after power on.

3.3 The frame buffer emulation

picoTK contains a framebuffer emulator (`fbe`) which emulates the framebuffer hardware under X-Windows. The emulator uses Unix shared memory inter process communication [2]. With real framebuffer hardware the framebuffer is located at a fixed and well known address (e.g. 0xa0000 for VGA hardware). When using the framebuffer emulator the emulated frame buffer base address is returned by the `mmap()`-call when initializing the shared memory. From the application point of view, obtaining and setting the framebuffer address is the only difference between real hardware and emulation. The framebuffer emulator constantly looks at the shared memory, interprets its contents and updates its graphic window accordingly. The application and the framebuffer emulation asynchronously access the framebuffer shared memory. The application reads and writes the framebuffer, while the emulator reads only. The `fbe` can emulate all supported picoTK color

depths. The color depths is set by starting `fbe` with appropriate parameters. The resolution and zooming can be set as well. Execute `fbe -help` to get a list of available options.

4 Getting started with picoTK

The following describes installation and configuration of picoTK. It has been tested with SuSE 7.0 and a RedHat 6.1 Linux distributions.

4.1 Requirements

- Xlib development support
- XServer with 8, 16 or 24 bits color depth. 8bpp is discouraged though.

4.2 Installation

- Choose a subdirectory for the installation of RTEMS and picoTK. picotk and rtems shall share the same parent directory, for the Makefiles to work properly.
- Unpack the distribution tarball

```
tar xzf picotk-xx.tgz
```

- Make picoTK

```
cd picotk
make
```

Note: picoTK cannot be compiled from a text console, since the `fontripper` tool requires access to a running X-Server.

4.3 Running the picoTK demo

From a common directory exacute the following. It is important to run these programs from the same directory as they need access to the same shared memory file handle `fbe_buffer`.

```
emulators/fbe &          ; This starts fbe in 4bpp mode (The demo
                           is compiled for this)
demo/demo                ; Start demo
```

4.4 Customizing picoTK

picoTK can be tailored by changing the picotk/toolkit/Makefile. This Makefile is responsible for generating the libPTK.a static library, which is to be compiled into the application.

4.4.1 Color depth and resolution settings

The following variables in `toolkit/Makefile` determine the driver settings:

```
COLOR_DEPTH
PIXLS_X
PIXLS_Y
PIXLS_TOTAL_X
BASE_ADDR
```

For some legacy and VESA VGA modes there are already prepared entries in the Makefile. Uncomment the appropriate lines and make sure to set the correct corresponding video mode in pc386_lilo's Makefile.inc as well. (For the picoTK demo application edit the file `demo/Makefile.inc.pc386_lilo` which get copied to the correct place during the make process.)

`COLOR_DEPTH` specifies the color depth - and in consequence the driver to be used. It's either 1, 4 or 8.

`PIXLS_X` and `PIXLS_Y` specify the X and Y display sizes in pixels.

`PIXLS_TOTAL_X` specifies the number of *total* X pixels. This is normally equal to PIXLS_X but eventually greater. For example custom hardwware may decide - easing implementation - to have a total X size of 512 Pixel, while only 320 are displayed. The number of total pixels is required to get the correct offset address into the framebuffer.

`BASE_ADDR` is the physical address where the linear frame buffer starts. The special macro `LILO_VGA_BASE` can be used instead of the actual address to request the frame buffer start from pc386_lilo; this is required for VESA VGA flat linear frame buffer modes.

4.4.2 Fonts

You can change the fonts included in the library by changing the Makefile. The picoTK tool `fontripper` rips a X-Font from the X-server and converts them into a C sourcefile. The file contains a (large) initialized unsigned char[] array wich contains the ripped font in a distinct format (see `PTK.h`, `struct picoFont` for the actual format). The makefile makes use of the `fontripper` for every font to be converted. fontripper accepts two arguments. Firstly the X-font name, in the way X understands fonts, i.e. the XFLD (X font logical description) containing wildcards (asterisk) for properties, which need not to be matched. Secondly `fontripper` gets a "picoTK name", this is used for referencing the font from within your picotk application. The common conventions for filenames and variable names are as follows:

```
<picoTK name>.c     The font source text
<picoTK name>.o     The conpiled picotk font
<font_picoTK name>  The reference (=variable name) for use in the
                       application
```

4.5 Customizing the framebuffer emulator

Start `fbe` with option `-help` to get a list of available options

5 Getting started with picoTK for RTEMS

picoTK makes a compact and easy to integrate GUI add-on for the RTEMS real- time operating systems. RTEMS is an OpenSource POSIX compliant real-time kernel written and maintained by the Online Application Research Corporation, Huntsville, AL. RTEMS is available for many embedded 32-bit platforms, including the PC, the Motorola PowerPC and the 68K. The most current versions of RTEMS now support TCP/IP networking, making it suitable for internet connected appliances. RTEMS is for real-time world, what Linux is for the dektop/server OSses. The following describes how to install RTEMS for a Linux development host and a PC (i386) target. The required GNU C/C++ cross-compiler toolchains for other host/target pairs are available precompiled from OAR's site `http://www.oarcorp.com/` as well. I personally used a SuSE 7.0 Linux system - any other current Linux distribution shall do as well.

5.1 Getting started with RTEMS

The following steps summarize the installation of RTEMS 4.5, the required additional tools for creating PC bootable floppy disks and linking of the picoTK demo application.

5.1.1 Get the distribution files

Get the following files from `ftp://ftp.rtems.com/pub/rtems/releases/4.5.0/....`

```
(for i386 Linux host, i386-Target):
 .../c_tools/linux_x86/RPMS/i386-rtems/
   i386-rtems-binutils-2.9.5.0.24-1.i386.rpm
   i386-rtems-gcc-gcc2.95.2newlib1.8.2-7.i386.rpm
   i386-rtems-gdb-4.18-4.i386.rpm
 .../c_tools/linux_x86/RPMS/shared/
   rtems-base-binutils-2.95.2newlib1.8.2-7.i386.rpm
   rtems-base-gcc-gcc2.95.2newlib1.8.2-7.i386.rpm
   rtems-base-gdb-4.18-4.i386.rpm
```

These are the precompiled toolchain packages for i386/i386 host/target combination.

```
.../
  rtems-4.5.0.tgz
```

This is the RTEMS source. We have to compile it using the above toolchain.

```
.../contrib
     pc386_lilo.tar.gz
```

RTEMS Bootloader. Allows booting RTEMS from floppy disk.

```
.../
     hello_world_c.tgz
```

Simple test program. Instructive for building your own Makefiles.

5.1.2 Install the RPMs

Become root. From the directory where you collected the RPMs type:

```
rpm -i rtems-base*rpm i386-rtems*rpm
```

(Accidently the alphabetic order implied by the bash wildcards gets the RPMs installed in the right order)

5.1.3 Add RTEMS toolchain directory to your path

```
.../
    export PATH=$PATH:/opt/rtems/bin
```

Assuming you use `bash` as your command shell. It is only needed for the following initial compilation of RTEMS itsel. Your own projects will find the compiler using a special environment variable as described in section "Set the environment"(5.1.6 ()).

5.1.4 Install the RTEMS source

Become an ordinary user. Create directory "rtems" somewhere in your home, e.g. /home/thomas/rtems. The rtems directory shall be on the same level as the "picotk" directory, see 4.1.2. The directory structure looks at follows (after the installation of all packages):

```
.../picotk/toolkit
   |      /emulators
   |      /demo/
   |           /rtems
   /rtems/rtems-4.5.0
         /build
         /pc386_lilo
```

Cd to the rtems directory and untar RTEMS:

```
cd
mkdir rtems
cd rtems
tar xzf <...>/rtems-4.5.0.tgz
```

Create empty directory in which the compilation results will be held during compilation. (See note below.)

```
mkdir build
```

Up to now you shall have the following dir structure:

```
.../rtems/rtems.4.5.0
         /build
```

IMPORTANT NOTE (Just for a better understanding): This is very special with RTEMS and might cause some trouble to the RTEMS newbie: Source texts and compilation output (i.e. objects) are kept in different directory trees (`.../rtems-4.5.0` and `.../build` respectively). Sometimes it is necessary to change the RTEMS source text (for doing configuration, tweaking drivers, etc.) Changing/Recompiling RTEMS requires these three steps:

1. Change source in the source-tree `.../rtems-4.5.0`
2. Compile in `.../rtems/build` directory using `make RTEMS_BSP=pc386`
3. Install RTEMS in `.../rtems/build` directory using `make install RTEMS_BSP=pc386`. This copies things into the "public" installation directories. The default settings of RTEMS - which are wise to use nevertheless - spatter RTEMS around the `/usr` and `/opt` directories. Don't forget this step.

5.1.5 Configure, build and install RTEMS

```
cd build
../rtems-4.5.0/configure --target=i386-rtems
make RTEMS_BSP=pc386
su                    /* Become root */
make install RTEMS_BSP=pc386
chmod -R 777 /usr/local/pc386
```

The chmod line is necessary since the RTEMS Makefiles insist on writing into that directory - causing write errors when being an ordinary user.

Compiling RTEMS might take a considerable amount of time (15-20 min on a K6-3/400).

5.1.6 Set the environment

RTEMS requires this additional setting in your environment:

```
export RTEMS_MAKEFILE_PATH=/usr/local/pc386       (sic!)
```

Add this line to the end of your personal `.bashrc` file. Logout and login again for the settings to take effect.

(The RTEMS_MAKEFILE_PATH points to the directory where a "Makefile.inc" file resides.)

5.1.7 Install pc386_lilo

As an ordinary user type:

```
cd .../rtems
tar xzf .../pc386_lilo.tgz
```

5.1.8 Install and Compile the Hello World demo

You can skip installation and test of the hello_world_demo and go straight to the picoTK RTEMS demo, ch. 5.2.

As ordinary user

```
cd .../rtems
tar xzf .../hello_world_c.tgz
cd hello_world_c
make
```

If you get an error about the not writable file in `/usr/local/pc386/...` you can either simply ignore this message or execute the `chmod` line from section "Configure, build and install RTEMS"(5.1.5 ()).

There shall be an RTEMS "executable" now named `o-optimize/test.exe`

5.1.9 Install pc386_lilo and generate hello_world boot disk

```
cd ..
cd rtems
tar xzf .../pc386_lilo.tgz
cd pc386_lilo
```

Edit `Makefile.inc`. Change the `RTEMS_IMAGE=...` line to include the hello world binary (it is required to enter the full path), e.g.:

```
RTEMS_IMAGE=$(HOME)/rtems/hello_world_c/o-optimize/test.exe
```

Then

```
make
```

This looks like the end of a Linux kernel compilation and yields a zImage file. Copy this file to a diskette and check the hello_world output. There is a bug in some versions of the GNU linker `ld` yielding in the following error:

```
ld: cannot open binary: No such file or directory
```

If this is the case edit the Makefile and substitute all occurences of `-oformat` by `-oformat`.

5.2 Compile the picoTK RTEMS library and the RTEMS demo

Now you are ready to compile the picoTK RTEMS library. This uses the RTEMS cross tool chain and the RTEMS OS installation - i.e. you definitely need to have them installed at this point.

5.2.1 Compile the RTEMS library

Change directory to where you installed picoTK. Execute

```
make rtems
```

This shall yield a file called "libPTKrtems.a" in the toolkit subdirectory. That is the statically linked picoTK library for RTEMS.

5.2.2 Compile RTEMS picoTK demo

```
cd demo
make rtems
make zImage
```

Make sure that `RTEMS_IMAGE` is pointing to your actual `demo.exe` before executing make `zImage`. If you get an error from `ld` please refer to the section describing the hello_world demo. This yields a file bootable zImage which can be copied to a diskette using

```
cp zImage /dev/fd0
```

(Depending on the permissions of `/dev/fd0` you might have to do this as root. You can set permissions of `/dev/fd0` to 666, i.e. `chmod 666 /dev/fd0` and futurely execute the copy command as an ordinary user)

Simply boot this diskette on a PC.

6 The picoTK C application interface (API)

picoTK is compiled into a single statically linkable library `libPTK.a` (the RTEMS version of the library is called `libPTKrtems.a` instead). The user's application is linked against this library. There is a single include file `PTK.h` which hold the prototypes for the exported procedures. The following briefly describes all API functions, which are intended to be used by the end user. The remaining functions which are used for interfacing the picoTK low-level hardware driver to the generic picoTK "kernel" are discussed in the section instead.

picoTK is intended to be thread-safe. I.e. in a multitasking, which RTEMS is, multiple tasks can quasi-concurrently work on (different) parts of the screen.

6.1 Initializing picoTK and the graphical context

```
void picoInit(void)
```

`picoInit()` is to be called before any other library functions. It initialises picoTK, clears the framebuffer to the default background color (normally black) and creates the default *graphic context*. Most of picoTK's functions have a parameter "graphic context".

The graphic context struct stores an environment, allowing different tasks to have independent contexts. The graphical context is simply a data structure of type `struct picoGC`. It stores such things as foreground,

background colors and the current font. If a NULL pointer is used instead of a pointer to `struct picoGC` the default graphical context is used.

```
void picoCreateGC(struct picoGC *gc)
```

`picoCreateGC()` initializes `gc` before it can be used.

6.2 Setting Options

The following operations can be carried out on the graphical context.

```
picoSetForeground(struct picoGC *gc, int color)
picoSetBackground(struct picoGC *gc, int color)
picoSetFont(struct picoGC *gc, struct picoFont *font)
```

Sets foregroundcolor, backgroundcolor and font for the given graphical context `gc`. Foreground, backgroundcolor and font have meaningful defaults set by `picoCreateGC`. The foreground color defaults to 1,15 and 15 for 1,4 and 8 bit per pixel modes respectively, the backgrond color defaults to 0), the default font is a 8 by 13 pixel monospaced font (`&font_8x13`).

Some fonts are already part of the picoTK library, other can be added. They can be referenced by a pointer to a `struct picoFont` . The following fonts are readily usable (without having to change the Makefile settings in the `toolkit` directory):

```
&font_10x20
&font_9x15
&font_8x13
&font_8x16
&font_7x14
&font_6x12
&font_5x7
&font_helv_b34
&font_helv_m20
&font_helv_m24
&font_helv_m25
&font_helv_m34
&font_school_m34
```

The ampersand is used to get the address of the font, which is casted to type `struct picoFont`

6.3 Coordinate system and color model

The coordinate system origin `(0,0)` is in the upper left screen corner. Positive values run to the right and down. Text coordinates are always referenced to the upper left corner of a (thought) surrounding rectangle. The same is true for pixmaps.

Color values start at 0 and go to 1, 15 and 255 for 1, 4 and 8 bit per pixel modes respectively. The actual mapping of color values to colors depends on the hardware implementation. picoTK applications know

nothing about the actual color mappings, i.e. there is no function to read the RGB values corresponding to a specific color value. On the other hand the programmer wants and has to know the actual mapping. Additionally the process of color pixmap generation from color pictures requires to know the color mapping. In future picoTK releases there will be a special configuration file `colorn.map` for each color depth which contains the mapping in a defined format. This means that the color mapping is known at compile-time (through `colorn.map`) but is unknown at run-time.

6.4 Drawing primitives

```
void picoDrawPoint(struct picoGC *gc, int x, int y)
void picoDrawLine(struct picoGC *gc,int x1, int y1, int x2, int y2)
void picoDrawRect(struct picoGC *gc,int x, int y, int w, int h)
void picoFillRect(struct picoGC *gc,int x, int y, int w, int h)
void picoReverseRect(struct picoGC *gc,int x, int y, int w, int h)
int  picoDrawText(struct picoGC *gc,int x, int y, char *txt)
int  picoDrawTextCentered(struct picoGC *gc, int x, int y, char *txt)
int  picoDrawPixmap (struct picoGC *gc, int x, int y,
                     struct picoPixmap *pixmap)
```

`picoDrawPoint()` plots a single pixel at position `x`, `y` using the foreground color.

`picoDrawLine()` draws a one pixel wide line between (`x1`,`y1`) and (`x2`,`y2`) using the foreground color

`picoDrawText()` draws Text starting at `x`, `y` (upper left corner of rectangle around text). Use GC-specified foreground/background color for text display. Use the GC-specified font. The string pointed through by `txt` is parsed for newline characters, which causes the text to extend to the next line.

`picoDrawTextCentered()` is similar to `picoDrawText()` but centered around `x` posistion, i.e. text extends to left and right of `x` to the same amount

`picoDrawPixmap()` draws pixmap at `x`,`y` (upper left corner of rectangle around pixmap). Use foreground/background color for pixmap (monochrome pixmaps only).

`picoDrawRect()` draws Rectangle with an upper left corner at (`x`,`y`) and lower right corner at (`x+w-1`,`y+h-1`)

`picoFillRect()` draws a filled Rectangle with an upper left corner at (`x`,`y`) and lower right corner at (`x+w-1`,`y+h-1`)

`picoReverseRect()` reverses rectangular area with an upper left corner at (`x`,`y`) and lower right corner at (`x+w-1`,`y+h-1`). Reversing means inverting every bit of the current color value.

6.5 Scrollable terminal box

This is a nice feature. picoTK can manage multiple scrollable terminal boxes in which you can print comfortably with a special printf compatible function (`picoTermPrintf()`). This makes it easy to dump debug information or whatever else you want onto the screen. Currently newline and return characters are interpreted to allow formatting. The following describes the respective API functions.

```
int  picoTerminalCreate(struct picoTerminal *term,
                        struct picoGC *gc,
```

```
                        int x, int y, int wc, int hc)
int  picoTerminalPutc(struct picoTerminal *term, int ch)
void picoTerminalPrintf(struct picoTerminal *term, char *fmt, ...)
```

`picoTerminalCreate()` creates a scrollable Terminal Window using the settings (font, foreground color, backgroundcolor) from the `gc`. The font is required to have fixed spacing in order for the terminal to work properly. (`x`,`y`) is the upper left corner of the terminal window. `wc` (width in characters) is the number of columns. `hc` (height in character) is the number of line. The actual size in pixels is depends on the font size and is internally calculated. It can be obtained after the call to `picoTerminalCreate()` by looking into `term->w` and `term->h`.

`picoTerminalPutc()` types single character into given terminal. The terminal handles some special ASCII characters and VT100/320 escape sequences:

`\n` (ADE 10) (Newline) moves cursor down one line.

`\r` (ADE 13) (Carriage Return) moves cursor to the beginning of the current line, i.e. there is no implicit newline.

`\b` (ADE 8) (Backspace) moves cursor left and deletes that character.

`\e[30m` ... `\e[38m` sets the Text foreground color (color selection compatible to the Linux console).

`\e[40m` ... `\e[48m` sets the Text background color (color selection compatible to the Linux console).

`\e[A` ... `\e[D` VT100 type cursor control for up, down, right and left respectively.

`picoTerminalPrintf()` types characters to the terminal according to the format string and the following parameters. It is compatible to how POSIX printf() handles its format string. Special character are interpreted by the terminal just as with `picoTerminalPutc()`

6.6 Text metrics functions

```
int  picoTextWidth(struct picoFont *font, char *txt)
int  picoTextHeight(struct picoFont *font, char *txt)
```

These functions calculate the width and height of the text `txt` formatted using `font`. These functions are useful when implementing your own text justification algorithms. The size returned is exactly the size of the surrounding rectangle of a text drawn using the same text and font with `picoDrawText()` or `picoDrawTextCentered()`.

7 picoTK driver API

picoTK is split into two parts. Firstly a generic part, which contains the hardware independent code. Currently the generic part is fully contained in the file `PTK_generic.`. Secondly the driver part, which adapts picoTK to a specific hardware platform. Three drivers are furnished with picoTK for 1, 4 and 8 bit color depth respectively - they are contained in the files `PTK_driver_1bpp.c`, `PTK_driver_4bpp.c` and `PTK_driver_8bpp.c` respectively. They are alternatively linked together with the generic part to form the picoTK library. That is a change of the color depth yields a different picoTK library. Some of the functions in the driver are non trivial - like the text draw functions - while others are. The implementation is not yet

fully optimized for speed. At present the 1 bit per pixel implementation is best optimized. The following describes the picoTK driver API functions.

7.1 Drawing functions

```
void picoDriverInit(void)
void picoDrawPointRaw(int x, int y, int color)
void picoFillRectRaw(int x, int y, int w, int h, int color)
void picoReverseRectRaw(int x, int y, int w, int h)
int  picoDrawTextRaw(int x, int y, char *txt,
                     struct picoFont *font, int fgc, int bgc)
int  picoDrawPixmapRaw(int x, int y,
                       struct picoPixmap *pm, int fgc, int bgc)
int  picoScrollRaw(int x, int y, int w, int h, int delta, int bgc)
```

`picoDriverInit()` Initializes the hardware, which is basically the clearing of the framebuffer. If your system requires intialization of a CRTC this has to been done here as well. When using RTEMS with the pc_386_lilo package the VGA subsystem has already been initialized and clearing the screen is all which has to be done. (Actually the PC's BIOS has already done the clearing of the screen on its own).

`picoDrawPoint()` This function draws a single point. It shall be optimized for speed as it is called for the line drawing and rectangle drawing functions as well.

`picoFillRectRaw()` Fills a rectangle window of the given size at the given origin with the specified color.

`picoReverseRectRaw()` Reverses a rectangle of the given size at the given origin. Every pixel value within this rectangle is inverted bitwise.

`picoDrawTextRaw()` Draws text. As opposed to the high-level routine `picoDrawText()` it does not do processing of non printables like linebreaks. This function is relatively time consuming. As it generally has to shift any byte from the font data to fit at the correct screen position. The 1 bit per pixel driver has an optimization for 8 pixel wide monospaced fonts on modulo 8 x position boundaries.

`picoDrawPixmapRaw()` Copies a picoTK pixmap to the screen. Currently only 1 bit color depth pixmaps are supported. With the 4 and 8 bit color depth drivers the pixmap is mapped to the screen by using the foreground and background colors for pixmap color values of 0 and 1 respectively.

`picoScrollRaw()` Horizontally scrolls the screen up or down by delta lines. Positive values of delta scroll down, negative scroll up.

7.2 Info functions

```
int picoInfoSizeX(void);
int picoInfoSizeY(void);
int picoInfoColorDepth(void);
```

`picoInfoSizeX()` and `picoInfoSizeY()` return the number of displayed pixels in X and Y direction respectively.

`picoInfoColorDepth()` return the number of bits per pixel. Values of 1, 4 or 8 mean 2, 16 or 256 simultaneously displayable colors. Color values range from 0 to 1, 15 and 255 respectively.

8 Document History

2003-Sep-21 (tgkamp)

Adjusted doc to reflect RTEMS-4.5.0 usage rather than RTEMS-4.5.0-beta3b.

9 References

[1]

Ferraro, R., Programmer's Guide to the EGA and VGA Cards, Addison-Wesley, 1989

This book covers register programming of VGA cards. It explained the addressing and access to graphics memory for the different legacy operating modes of EGA and VGA cards. It is hopelessly out of date and does not cover the current accelerated cards and VESA.

[2]

Stevens, R., Unix Network Programming, Volume2, Prentice-Hall, 1998

This excellent book covers the use of shared memory for interprocess communication in detail - as well as many other UNIX IPC methods.

[3]

VESA Bios Extensions core functions standard, Version 3.0, Video electronics standards association, 1998, Freely available as PDF from www.vesa.org, Documents the Int10-BIOS, Function 0x4f extensions.