Embedded Applications with Qt

Qt 3.0

Contents

Running Qt/Embedded applications

A Qt/Embedded application requires a master application to be running or to be a master application itself. The master application is primarily responsible for managing top-level window regions, pointer and keyboard input.

Any Qt/Embedded application can be a master application by constructing the QApplication object with the *QApplication::GuiServer* type, or running the application with the *-qws* command line option.

This document assumes you have the Linux framebuffer configured correctly and no master process is running. If you do not have a working Linux framebuffer you can use the Qt/Embedded virtual framebuffer, or you can run Qt/Embedded as a VNC server.

Change to a Linux console and select an example to run, e.g. examples/widgets. Make sure $QTDIR is set to the directory where you installed Qt/Embedded and add the $QTDIR/lib directory to $LD_LIBRARY_PATH, e.g.:

```
export QTDIR=$HOME/qt-version
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

Run the application with the *-qws* option:

```
cd $QTDIR/examples/widgets
./widgets -qws
```

You should see the widgets example appear. If your mouse doesn't work correctly you need to specify the type of mouse to use. You can exit the master application at any time using ctrl+alt+backspace.

If you wish to run additional applications you should run them as clients i.e. without the *-qws* option.

Displays

Qt/Embedded allows multiple displays to be used simultaneously by running multiple Qt/Embedded master processes. This is achieved using the -display command line parameter or the $QWS_DISPLAY environment variable.

The -display parameter's syntax is:

```
    [gfx driver][:driver specific options][:display number]
```

for example if you want to use the mach64 driver on fb1 as display 2:

```
    $ ./launcher -display Mach64:/dev/fb1:2
```

To try this functionality you can do the following:

1. Change to VC 1 and run the launcher: `$ cd examples/launcher $ ./launcher`
2. Switch to VC 2 and run another one: `$ cd examples/launcher $ ./launcher -display :1` Another launcher will be started. Start an application in this launcher.

3. Press ctrl+alt+F1 - back to display 0. You can also start additional applications on a particular display by specifying the display id. Change to VC 3: `$ cd examples/widgets` `$ ./widgets -display :1` will display the widgets example on dislpay :1 (VC 2).

Only the master process needs to specify the driver/device part explicitly. The clients get the information they need from the master when they connect. So once you have a master server running using a particular driver, you can just use "client -display :n" to use display n.

Mouse Input

At the time of writing Qt/Embedded supports MouseMan (default), Microsoft, IntelliMouse and some other devices specific to certain hardware (e.g. Vr touch panel). To specify the mouse to use set the $QWS_MOUSE_PROTO environment variable, e.g.:

```
export QWS_MOUSE_PROTO=IntelliMouse
```

See also Qt/Embedded Pointer Handling [p. 20].

Character input in Qt/Embedded

Internally in the client/server protocol, each key press and key release is sent as a `QWSKeyEvent`. A QWSKeyEvent contains the following fields:

- `unicode`: Unicode value
- `keycode`: Qt keycode value as defined in qnamespace.h
- `modifier`: A bitfield consisting of some of Qt::ShiftButton, Qt::ControlButton, and Qt::AltButton.
- `is_press`: TRUE if this is a key press, FALSE if it is a key release.
- `is_auto_repeat`: TRUE if this event is caused by auto repeat.

When the server receives a key event, it is sent to each client process, which is responsible for processing the key event and sending it to the right window, if any. Key events may come from several different sources.

Keyboard drivers

A keyboard driver reads data from a device and gives key events to the server.

Keyboard drivers are currently compiled into the library. They are defined in the file src/kernel/qkeyboard_qws.cpp. At the time of writing, the following drivers are defined:

`QWSTtyKeyboardHandler` Input from the system console (tty) `QWSVr41xxButtonsHandler` Input handler for the buttons on Cassiopeia-style PDAs `QWSVFbKeyboardHandler` Virtual framebuffer key input

The keyboard drivers all follow the same pattern. They read keyboard data from a device, find out which keys were pressed, and then call the static function QWSServer::processKeyEvent() with the key information.

At present, the console keyboard driver also handles console switching (Ctrl-Alt-F1...F10) and termination (Ctrl-Alt-Backspace).

To add a keyboard driver for a new device, make a subclass of `QWSKeyboardHandler` and instantiate it in `QWSServer::newKeyboardHandler()` (in qkeyboard_qws.cpp).

Key event filters (input methods)

When the server receives a key event from a keyboard driver, it first passes it through a filter.

This can be used to implement input methods, providing input of characters that are not on the keyboard.

To make an input method, subclass QWSServer::KeyboardFilter (in src/kernel/qwindowsystem_qws.h) and implement the virtual function filter(). If filter() returns FALSE, the event will be sent to the clients (using QWSServer::sendKeyEvent()). If filter() returns TRUE, the event will be stopped. To generate new key events, use QWSServer::sendKeyEvent(). (Do not use processKeyEvent(), since this will lead to infinite recursion.)

To install a keyboard event filter, use `QWSServer::setKeyboardFilter()`. Currently, only one filter can be installed at a time.

Filtering must be done in the server process.

The launcher example contains an example of a simple input method, `SimpleIM` which reads a substitution table from a file.

Pen input

Key events do not need to come from a keyboard device. The server process may call QWSServer::sendKeyEvent() at any time.

Typically, this is done by popping up a widget, and letting the user specify characters with the pointer device.

Note: the key input widget should not take focus, since the server would then just send the key events back to the input widget. One way to make sure that the input widget never takes focus is to set the `WStyle_Customize` and `WStyle_Tool` widget flags in the QWidget constructor.

The compact example contains three example input widgets:

- A simple handwriting recognition example.
- A virtual keyboard example.
- An example Unicode input widget.

Qt/Embedded Case Study - Cassiopeia E-100

Introduction

This document describes the steps involved in installing Linux on an embedded device and building a Qt/Embedded application. The target device is the Cassiopeia E-100/E-105. The device has a MIPS Vr4121 processor, 16MB RAM (32MB in E-105), a Compact Flash slot and a 240x320 16 bit per pixel LCD display.

The only part of this document that is specific to the Cassiopeia is the installation of Linux and the development tools. The example application can be compiled and run on your desktop machine.

Installing Linux

All the information and software required to get Linux running on the VR series of processors is available from the Linux VR web site. In Summary:

Install the tools

Follow the instructions at http://www.linux-vr.org/tools.html.

Build the kernel

Get a sample root ramdisk from ftp://ftp.ltc.com/pub/linux/mips/ramdisk/ramdisk

Follow the instructions at http://www.linux-vr.org/ramdisk.html to create a ramdisk.o file.

Now build your kernel http://www.linux-vr.org/kernel.html using this ramdisk object. Make sure you have at least the following configuration:

- Development/incomplete drivers
- Casio E105 Platform
- Network and System V IPC
- RAM disk and Initial RAM disk support
- Support for console on virtual terminal (so that you can see boot messages)
- /proc and ext2 filesystem support
- Simple Frame Buffer with HPC device control

Booting Linux

Follow the instructions at http://www.linux-vr.org/booting.html.

You should see the linux boot messages on the LCD display.

A Qt/Embedded Application

Usually a device such as the Cassiopeia would have a shell, configured as the Qt/Embedded server, that allows client applications to be launched. For the purposes of this tutorial, we will write a simple application that serves as the Qt/Embedded server and client. A more complete Qt/Embbeded server can be found in $QT-DIR/examples/compact.

The application that we will write is a simple note pad. It will allow notes to be created, viewed and deleted. Since the Cassiopeia doesn't have a keyboard, we will include a simple on-screen keyboard for input.

Note Pad

Our note pad user interface is very simple. It consists of a toolbar with "New" and "Delete" buttons, a combo box to select the note to view and an editing area.

Take a moment to browse the source code for Note Pad in $QTDIR/examples/notepad/. The code is quite simple, but there are some things worth noting:

1. Two fonts are set - helvetica 10 point as the application default font, and helvetica 12 point for the editor. Since we will use prerendered fonts these fonts must be prepared as described here.
2. The QApplication is constructed with the QApplication::GuiServer type specified. This makes the note pad a Qt/Embedded server. One server must be running for Qt/Embedded clients to run. In this case our application is both server and client because it is the only application we wish to run on our device.
3. The Cassiopeia has no keyboard (generally) so we must provide some means of character input with the pen. The simplest method is to display a small keyboard. The compact example includes a keyboard, so we use this code. Key and pointer input is Qt/Embedded specific, so it is surrounded by #ifdef Q_WS_QWS ... #endif so that we can compile the example with Qt/X11 or Qt/Windows if we wish.
4. The touch panel needs to be calibrated. There is a calibration module included in the compact demo, so we use this.

Creating a suitable Qt/Embedded Library

Since our application is quite simple we can remove some unneeded features from Qt/Embedded. Edit $QT-DIR/src/tools/qconfig.h and disable unneeded features as follows:

```
#define QT_NO_IMAGEIO_BMP
#define QT_NO_IMAGEIO_PPM
#define QT_NO_IMAGEIO_XBM
#define QT_NO_IMAGEIO_PNG
#define QT_NO_ASYNC_IO
#define QT_NO_ASYNC_IMAGE_IO
#define QT_NO_TRUETYPE
#define QT_NO_BDF
#define QT_NO_FONTDATABASE
#define QT_NO_MIME
#define QT_NO_SOUND
#define QT_NO_PROPERTIES
```

```
#define QT_NO_CURSOR
#define QT_NO_NETWORKPROTOCOL
#define QT_NO_COLORNAMES
#define QT_NO_TRANSFORMATIONS
#define QT_NO_PSPRINTER
#define QT_NO_PICTURE
#define QT_NO_LISTVIEW
#define QT_NO_CANVAS
#define QT_NO_DIAL
#define QT_NO_WORKSPACE
#define QT_NO_TABLE
#define QT_NO_LCDNUMBER
#define QT_NO_STYLE_MOTIF
#define QT_NO_STYLE_PLATINUM
#define QT_NO_COLORDIALOG
#define QT_NO_PROGRESSDIALOG
#define QT_NO_TABDIALOG
#define QT_NO_WIZARD
#define QT_NO_EFFECTS
```

See Qt Features for a description of the features that can be disabled. This leaves us with a small set of widgets and dialogs necessary for our application. Cross-compile the library for the mips target on the x86 platform:

```
cd $QTDIR
./configure -xplatform linux-mips-g++ -platform linux-x86-g++
make
mipsel-linux-strip $QTDIR/lib/libqt.so.2.2.0
```

The library is stripped to conserve ramdisk space.

Installation

Compile the application:

```
cd examples/notepad
make
mipsel-linux-strip notepad
```

We have chosen to link the application dynamically. While this is important if we plan to run multiple applications, it is a waste of space in an application such as notepad that is supposed to be the only application running. You can link statically by configuring with:

```
./configure -static -xplatform linux-mips-g++ -platform linux-x86-g++
```

We must install our application and its support files in the ramdisk. Mount the ramdisk using loopback device (you will need loopback device support in your kernel):

```
mkdir /mnt/ramdisk
mount -o loop ~/ramdisk /mnt/ramdisk
```

Copy the Qt/Embedded library to the ramdisk /lib directory and make the necessary links:

```
cd /mnt/ramdisk/lib
cp $QTDIR/lib/libqt.so.2.2.0 .
ln -s libqt.so.2.2.0 libqt.so.2.2
ln -s libqt.so.2.2.0 libqt.so.2
```

The fonts must be installed in /usr/local/qt-embedded/etc/fonts:

```
    cd /mnt/ramdisk
    mkdir usr/local
    mkdir usr/local/qt-embedded
    mkdir usr/local/qt-embedded/etc
    mkdir usr/local/qt-embedded/etc/fonts
    cp helvetica_100_50.qlf helvetica_120_50.qlf usr/local/qt-embedded/etc/fonts
```

When the kernel boots it looks for several files to run. In order to have our application run when the kernel boots, we rename it to /bin/sh. A /tmp directory is also used by Qt/Embedded:

```
    cp $QTDIR/examples/notepad/notepad /mnt/ramdisk/bin/sh
    mkdir /mnt/ramdisk/tmp
    umount /mnt/ramdisk
```

Create a ramdisk object, link it with the kernel, copy it to the compact flash and boot Linux. You should see the calibration screen appear on the LCD display.

Qt/Embedded Virtual Framebuffer

The virtual framebuffer allows Qt/Embedded programs to be developed on your desktop machine, without switching between consoles and X11.

The virtual framebuffer is located in `$QTDIR/tools/qvfb`.

Using the Virtual Framebuffer

1. Make sure QT_NO_QWS_VFB in `$QTDIR/src/tools/qconfig.h` is not defined and compile the Qt/Embedded library.
2. Compile qvfb as a normal Qt/X11 application and run it. Do not compile it as a Qt/Embedded application.
3. Start a Qt/Embedded master application (i.e., construct QApplication with QApplication::GuiServer flag or use the -qws command line parameter). You will need to specify to the server that you wish to use the virtual framebuffer driver, e.g.:

   ```
   widgets -qws -display QVFb:0
   ```

4. You may prefer to set the `QWS_DISPLAY` environment variable to be `QVFb:0`.

qvfb supports the following command line options:

- `-width width`: the width of the virtual framebuffer (default: 240).
- `-height height`: the height of the virtual framebuffer (default: 320).
- `-depth depth`: the depth of the virtual framebuffer (1, 8 or 32; default: 8).
- `-nocursor`: do not display the X11 cursor in the framebuffer window.
- `-qwsdisplay :id` the Qt/Embedded display id to provide (default: 0).

Virtual Framebuffer Design

The virtual framebuffer emulates a framebuffer using a shared memory region (the virtual frame buffer) and a utility to display the framebuffer in a window (qvfb). The regions of the display that have changed are updated periodically, so you will see discrete snapshots of the framebuffer rather than each individual drawing operation. For this reason drawing problems such as flickering may not be apparent until the program is run using a real framebuffer.

The target refresh rate can be set via the "View|Refresh Rate" menu item. This will cause qvfb to check for updated regions more quickly. The rate is a target only. If little drawing is being done, the framebuffer will not show any updates between drawing events. If an application is displaying an animation the updates will be frequent, and the application and qvfb will compete for processor time.

Mouse and keyboard events are passed to the Qt/Embedded master process via named pipes.

The virtual framebuffer is a development tool only. No security issues have been considered in the virtual framebuffer design. It should be avoided in a production environment; QT_NO_QWS_VFB should always be defined in production libraries.

Enabling the Linux Framebuffer

This is only a short guide. See /usr/src/linux/README and /usr/src/linux/Documentation/fb/ for detailed information. There is also a detailed explanation at http://www.linuxdoc.org/HOWTO/Framebuffer-HOWTO.html.

1. Make sure that you have the Linux kernel source code in /usr/src/linux/.
2. Log in as root and cd /usr/src/linux
3. Configure the kernel:

   Run:

   ```
   make menuconfig
   ```

   Select "Code maturity level options" and set "Prompt for development and/or incomplete code/drivers".

   Then select "Console drivers" set "Support for frame buffer devices" to built-in (even if it says EXPERIMENTAL). Then configure the driver. Most modern graphics cards can use the "VESA VGA graphics console"; use that or a driver that specifically matches your video card. Finally enable "Advanced low level driver options" and make sure that 16 and 32 bpp packed pixel support are enabled.

   When you are finished, chose exit and save.
4. Compile the kernel

   First do:

   ```
   make dep
   ```

   then:

   ```
   make bzImage
   ```

   The new kernel should now be in arch/i386/boot/bzImage.
5. Copy the kernel to the boot directory:

   ```
   cp arch/i386/boot/bzImage /boot/linux.vesafb
   ```

6. Edit /etc/lilo.conf.

   Warning: Keep a backup of /etc/lilo.conf, and have a rescue disk available. If you make a mistake here, the machine may not boot.

   The file /etc/lilo.conf specifies how the system boots. The precise contents of the file varies from system to system, this is one example:

   ```
   # LILO configuration file
   boot = /dev/hda3
   delay = 30
   image = /boot/vmlinuz
     root = /dev/hda3
     label = Linux
   ```

```
  read-only # Non-UMSDOS filesystems should be mounted read-only for checking
other=/dev/hda1
        label=nt
        table=/dev/hda
```

Warning: Keep a backup of /etc/lilo.conf, and have a rescue disk available. If you make a mistake here, the machine may not boot.

Make a new "image" section that is a copy of the first one, but with image = /boot/linux.vesafb and label = Linux-vesafb. Place it just above the first image section.

Add a line before the image section saying 'vga = 791'. (Meaning 1024x768, 16 bpp.)

With the above example, lilo.conf would now be:

```
# LILO configuration file
boot = /dev/hda3
delay = 30
vga = 791
image = /boot/linux.vesafb
  root = /dev/hda3
  label = Linux-vesafb
  read-only # Non-UMSDOS filesystems should be mounted read-only for checking
image = /boot/vmlinuz
  root = /dev/hda3
  label = Linux
  read-only # Non-UMSDOS filesystems should be mounted read-only for checking
other=/dev/hda1
        label=nt
        table=/dev/hda
```

Do not change any existing lines in the file; just add new ones.

7. To make the new changes take effect, run the lilo program:

```
    lilo
```

8. Reboot the system. You should now see a penguin logo while the system is booting. (Or more than one on a multi-processor machine.)
9. If it does not boot properly with the new kernel, you can boot with the old kernel by entering the label of the old image section at the LILO prompt. (with the example lilo.conf file, the old label is Linux.)

   If that does not work (probably because of an error in lilo.conf), boot the machine using your rescue disk, restore /etc/lilo.conf from backup and re-run lilo.
10. Testing: Here's a short program that opens the frame buffer and draws a gradient-filled red square.

```
#include
#include
#include
#include
#include

int main()
{
    int fbfd = 0;
    struct fb_var_screeninfo vinfo;
    struct fb_fix_screeninfo finfo;
    long int screensize = 0;
    char *fbp = 0;
    int x = 0, y = 0;
```

```
long int location = 0;

// Open the file for reading and writing
fbfd = open("/dev/fb0", O_RDWR);
if (!fbfd) {
    printf("Error: cannot open framebuffer device.\n");
    exit(1);
}
printf("The framebuffer device was opened successfully.\n");

// Get fixed screen information
if (ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo)) {
    printf("Error reading fixed information.\n");
    exit(2);
}

// Get variable screen information
if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
    printf("Error reading variable information.\n");
    exit(3);
}

printf("%dx%d, %dbpp\n", vinfo.xres, vinfo.yres, vinfo.bits_per_pixel );

// Figure out the size of the screen in bytes
screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;

// Map the device to memory
fbp = (char *)mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED,
                   fbfd, 0);
if ((int)fbp == -1) {
    printf("Error: failed to map framebuffer device to memory.\n");
    exit(4);
}
printf("The framebuffer device was mapped to memory successfully.\n");

x = 100; y = 100;       // Where we are going to put the pixel

// Figure out where in memory to put the pixel
for ( y = 100; y < 300; y++ )
    for ( x = 100; x < 300; x++ ) {

        location = (x+vinfo.xoffset) * (vinfo.bits_per_pixel/8) +
                   (y+vinfo.yoffset) * finfo.line_length;

        if ( vinfo.bits_per_pixel == 32 ) {
            *(fbp + location) = 100;        // Some blue
            *(fbp + location + 1) = 15+(x-100)/2;     // A little green
            *(fbp + location + 2) = 200-(y-100)/5;    // A lot of red
            *(fbp + location + 3) = 0;      // No transparency
        } else  { //assume 16bpp
            int b = 10;
            int g = (x-100)/6;     // A little green
            int r = 31-(y-100)/16;    // A lot of red
            unsigned short int t = r<<11 | g << 5 | b;
            *((unsigned short int*)(fbp + location)) = t;
        }
        }
    munmap(fbp, screensize);
    close(fbfd);
    return 0;
}
```

Fonts in Qt/Embedded

Supported Formats

Qt/Embedded supports four font formats:

- **TrueType (TTF)** - the scalable font technology now standard on MS-Windows and Apple Macintosh, and becoming popular on X11.
- **Postscript Type1 (PFA/PFB)** - scalable fonts often used by printers, also popular on X11. These are similar in functionality to TTF fonts and are not discussed further in this document.
- **Bitmap Distribution Format fonts (BDF)** - a standard format for non-scalable fonts. A large number of BDF fonts are supplied as part of standard X11 distributions - most of these can be used with Qt/Embedded. You should *not* use these in a production system: they are very slow to load and take up a *lot* of storage space. Instead, render the BDF to a QPF.
- **Qt Prerendered Font (QPF)** - a light-weight non-scalable font format specific to Qt/Embedded.

Support for each of these font formats, except QPF which is always enabled, can be enabled or disabled independently by using the Qt/Embedded Features Definition There is support in Qt/Embedded for writing a QPF font file from any font, thus you can initially enable TTF and BDF formats, save QPF files for the fonts and sizes you need, then remove TTF and BDF support.

See the `tools/makeqpf` for a tool that assists producing QPF files from the TTF and BDF, or just run your application with the `-savefonts` option.

Memory Requirements

With TTF fonts, each character in the font at a given point size is only rendered when first used in a drawing or metrics operation. With BDF fonts all characters are rendered when the font is used. With QPF fonts, the characters are stored in the same format as Qt uses when drawing.

As an example, a 10-point Times font containing the ASCII characters uses around 1300 bytes when stored in QPF format.

Taking advantage of the way the QPF format is structured, Qt/Embedded memory-maps the data rather than reading and parsing it. This reduces the RAM consumption even further.

The scalable fonts use a larger amount of memory per font, but these fonts can give a memory saving if many different sizes of each font are needed.

Smooth Fonts

TTF, PFA, and QPF fonts can be rendered as *smooth* anti-aliased fonts to give superior readability, especially on low-resolution devices. The difference between smooth and non-smooth fonts is illustrated below (you may need to change your display to low resolution to see the difference):

Unsmooth

Smooth

In Qt/Embedded 2.2.1, smooth fonts use 8 times as much memory as non-smooth fonts. This multiplier will be reduced to a configurable 2 or 4 (ie. 4-level and 16-level shading rather than the current excessive 256-level shading).

Unicode

All fonts used by Qt/Embedded use the Unicode character encoding. Most fonts available today use this encoding, but they usually don't contain all the Unicode characters. A *complete* 16-point Unicode font uses over 1 MB of memory.

The font definition file

When Qt/Embedded applications run, they look for a file called `$QTDIR/etc/fonts/fontdir` or `/usr/local/qt-embedded/etc/fonts/fontdir`. This file defines the fonts available to the application. It has the following format:

name file renderer italic weight size flags

where

Field	Value
name	`Helvetica`, `Times`, etc.
file	`helvR0810.bdf`, `verdana.ttf`, etc.
renderer	`BDF` or `TTF`
italic	`y` or `n`
weight	`50` is normal, `75` is bold, etc.
size	`0` for scalable or pointsize times 10 (e.g., `120` for 12pt)
flags	• `s`: smooth (anti-aliased) • `u`: unicode range when saving (default is Latin-1) • `a`: ascii range when saving (default is Latin-1)

The font definition file does not specify QPF fonts; these are loaded directly from the directory containing the `fontdir` file, and must be named *name_size_weightitalicflag*.qpf, where

Field	Value
name	`helvetica`, `times`, etc. (in lowercase)
size	pointsize times 10 (e.g., `120` for 12pt)
italicflag	`i` for italic, otherwise nothing.
weight	`50` is normal, `75` is bold, etc.

If an application is run with the `-savefonts` command-line option, then whenever a font other than a QPF font is used, a corresponding QPF file is saved. This allows you to easily find the font usage of your applications and to generate QPF files so that you can eventually reduce the memory usage of your applications by disabling TTF and BDF support from Qt/Embedded. option, or by modifying the initialization of `qws_savefonts` in `kernel/qapplication_qws.cpp` of the Qt/Embedded library source code. In extreme cases of memory-saving, it is possible to save partially-rendered fonts (eg. only the characters in "Product NameTM") if you are certain that these are the only characters you will need from the font. See QMemoryManager::savePrerenderedFont() for this functionality.

Notes

The font definition file, naming conventions for font files, and the format of QPF files may change in versions of Qt/Embedded after 2.2.1.

When enabled, Qt/Embedded uses the powerful FreeType2 library to implement TrueType and Type1 support.

Qt/Embedded Pointer Handling

Pointer handling in Qt/Embedded works for any mouse-like device such as a touchpanel, a trackball, or real mouse.

In a real device, only a small number of pointer devices (usually one) would be supported, but for demonstration purposes, Qt/Embedded includes a large number of supported devices.

Mouse Protocols

Qt/Embedded normally auto-detects the mouse type and device if it is one of the supported types on `/dev/psaux` or one of the `/dev/ttyS?` serial lines. If multiple mice are detected, all may be used simultaneously.

Alternatively, you may set the environment variable `QWS_MOUSE_PROTO` to determine which mouse to use. This environment variable may be set to:

<*protocol*>`:`<*device*>

where <*protocol*> is one of:

- MouseMan
- IntelliMouse
- Microsoft

and <*device*> is the mouse device, often `/dev/mouse`. If no such variable is specified, the built-in default is `Auto`, which enables auto-detection of the mouse protocol and device.

To add another protocol, new subclasses of QAutoMouseSubHandler or QMouseHandler can be written in `kernel/qwsmouse_qws.cpp`.

Touch Panels

Qt/Embedded ships with support for the NEC Vr41XX touchpanel and the iPAQ touchpanel. These are subclasses of QCalibratedMouseHandler which is in turn a subclass of QMouseHandler in `kernel/qwsmouse_qws.cpp`.

Adding an accelerated graphics driver to Qt/Embedded

Qt/Embedded has the capacity to make use of hardware accelerators. To do so for a PCI or AGP driver, you need to perform the following steps:

1) Define an accelerated descendant of QLinuxFbScreen. This should implement QVoodooScreen::connect to map its registers. Use qt_probe_bus to get a pointer to PCI config space. This is where you should check that you're being pointed at the right device (using PCI device/manufacturer ID information). Then use PCI config space to locate your device's accelerator registers in physical memory and mmap the appropriate region in from /dev/mem. There is no need to map the framebuffer, QLinuxFbScreen will do this for you. Return FALSE if a problem occurs at any point. QVoodooScreen::initDevice will be called only by the QWS server and is guaranteed to be called before any drawing is done (and so is a good place to set registers to known states). connect will be called by every connecting client.

2) Define an accelerated descendant of QGfxRaster. This is where the actual drawing code goes. Anything not implemented by hardware can be passed back to QGfxRaster to do in software. Use the optype variable to make sure that accelerated and unaccelerated operations are synchronised (if you start drawing via software into an area where the hardware accelerator is still drawing then your drawing operations will appear to be in the wrong order). optype is stored in shared memory and is set to 0 by unaccelerated operations; accelerated operations should set it to 1. When a software graphics operation is requested and optype is 1, QGfxRaster::sync() is called; you should provide your own implementation of this which waits for the graphics engine to go idle. lastop is also available for optimisation and is stored in the shared space - this will not be set by the software-only QGfx and can be used to store the type of your last operation (e.g. drawing a rectangle) so that part of the setup for the next operation can be avoided when a lot of the same operations are performed in sequence. All drawing operations should be protected via a QWSDisplay::grab() before any registers, lastop or optype are accessed, and ungrabbed() at the end. This prevents two applications trying to access the accelerator at once and possibly locking up the machine. It's possible that your source data is not on the graphics card so you should check in such cases and fall back to software if necessary. Note that QGfxRaster supports some features not directly supported by QPainter (for instance, alpha channels in 32-bit data and stretchBlt's). These features are used by Qt; stretchBlt speeds up QPixmap::xForm and drawPixmap into a transformed QPainter, alpha channel acceleration is supported for 32-bit pixmaps.

3) If you wish, define an accelerated descendant of QScreenCursor. restoreUnder,saveUnder,drawCursor and draw should be defined as null operations. Implement set, move, show and hide. 4k is left for your cursor at the end of the visible part of the framebuffer (i.e. at (width*height*depth)/8)

4) Implement initCursor and createGfx in your QScreen descendant. Implement useOffscreen and return true if you can make use of offscreen graphics memory.

5) Implement a small function qt_get_screen_mychip, which simply returns a new QMychipScreen

6) Add your driver to the DriverTable table in qgfxraster_qws.cpp - e.g.

{ "MyChip", qt_get_screen_mychip,1 },

The first parameter is the name used with QWS_DISPLAY to request your accelerated driver.

7) To run with your new driver, export QWS_DISPLAY=MyChip (optionally MyChip:/dev/fb to request a different Linux framebuffer than /dev/fb0) run the program

If your driver is not PCI or AGP you'll need to inherit QScreen instead of QLinuxFbScreen and implement similar functionality to QLinuxFbScreen, but otherwise the process should be similar. The most complete example driver is qgfxmach64_qws.cpp; qgfxvoodoo_qws.cpp may provide a smaller and more easy to understand driver.

Qt/Embedded as a VNC Server

The VNC protocol allows you to view and interact with the computer's display from anywhere on the network.

To use Qt/Embedded in this way, `configure` Qt with the `-vnc` option, and ensure you also enable 16-bit display support. Run your application via:

```
application -display VNC:0
```

then, run a VNC client pointed at the machine that is running your application. For example, using the X11 VNC client to view the application from the same machine:

```
vncviewer localhost:0
```

By default, Qt/Embedded will create a 640 by 480 pixel display. You can change this by setting the `QWS_SIZE` environment variable to another size, eg. `QWS_SIZE=240x320`.

VNC clients are available for a vast array of display systems - X11, Windows, Amiga, DOS, VMS, and dozens of others.

The Qt Virtual Framebuffer is an alternative technique. It uses shared memory and thus is much faster and smoother, but it does not operate over a network.

The Qt/Embedded-specific classes

Qt/Embedded classes fall into two classes - the majority are used by every Qt/Embedded program, some are used only by the Qt/Embedded server. The Qt/Embedded server program can be a client as well, as in the case of a single-process installation. All Qt/Embedded specific source files live in src/kernel and are suffixed _qws. -> indicates inheritance.

Client classes:

QFontManager There is one of these per application. At application startup time it reads the font definition file from $QTDIR/etc/fonts/fontdir (or /usr/local/etc/qt-embedded/fonts/fontdir if QTDIR is undefined). It keeps track of all font information and holds a cache of rendered fonts. It also creates the font factories - QFontManager::QFontManager is the place to add constructors for new factories. It provides a high-level interface to requesting a particular font and calls QFontFactories to load fonts from disk on demand. Note that this only applies to BDF and Truetype fonts; Qt/Embedded's optimised .qpf font file format bypasses the QFontManager mechanism altogether. There should be no need to modify this class unless you wish to change font matching or cacheing behaviour.

QDiskFont This contains information about a single on-disk font file (e.g. /usr/local/etc/qt-embedded/times.ttf, would be an example). It holds the file path, information about whether the font is scalable, its weight, size and Qt/Embedded name and so on. This information is used so that QFontManager can find the closest match disk font (it uses a scoring mechanism weighted towards matching names, then whether the font's italic, then weight). There should be no reason to modify this class.

QRenderedFont There is one and only one QRenderedFont for every unique font currently loaded by the system (that is, each unique combination of name, size, weight, italic or not, anti-aliased or not) QRenderedFonts are reference counted; once noone is using the QRenderedFont it is deleted along with its cache of glyph bitmaps. The QDiskFont it was loaded from remains opened by its QFontFactory however. There should be no reason to modify this class, unless you wish to change the way in which glyphs are cached.

QFontFactory (and descendants QFontFactoryBDF,QFontFactoryTtf) These provide support for particular font formats, for instance the scalable Truetype and Type1 formats (both supported in QFontFactoryTtf, which uses Freetype 2) and the bitmap BDF format used by X. It's called to open an on-disk font; once a font is opened it remains opened, for quickness in creating new font instances from the disk font. It can also create a QRenderedFont and convert from unicode values to an index into the font file - glyphs are stored in the order and indexes they are defined in the font rather than in unicode order for compactness. There should be no need to modify this class, but it should be inherited if you wish to add a different type of font renderer (e.g. for a custom vector font format).

QGlyph This describes a particular image of a character from a QRenderedFont - for example, the letter 'A' at 10 points in Times New Roman, bold italic, anti-aliased. It contains pointers to a QGlyphMetrics structure with information about the character and to the raw data for the glyph - this is either a 1-bit mask or an 8-bit alpha channel. Each QRenderedFont creates these on demand and caches them once created (note that this is not currently implemented for Truetype fonts). You would only need to modify this class if you were, for instance, modifying Qt/Embedded to support textured fonts, in which case you would also need to modify QGfxRaster.

QMemoryManagerPixmap/QMemoryManager This handles requests for space for pixmaps and also keeps track of QPF format fonts (these are small 'state dumps' of QRenderedFonts, typically 2-20k in size; they can be mmap'd direct from disk in order to save memory usage). If a QPF font is found which matches a font request no new QRenderedFont need be created for it. It's possible to strip out all QFontFactory support and simply use QPFs if your font needs are modest (for instance, if you only require a few fixed point sizes). Note that no best-match loading is performed with QPFs, as opposed to those loaded via QFontManager, so if you don't have the correct QPF for a point size text in that size will simply not be displayed. There should be no need to modify this class.

QScreen -> QLinuxFbScreen -> accelerated screens, QTransformedScreen -> QVfbScreen

These encapsulate the framebuffer Qt/Embedded is drawing to, provide support for mapping of coordinates for rotating framebuffers, allow manipulation of the colour palette and provide access to offscreen graphics memory for devices with separate framebuffer memories. This is used for cacheing pixmaps and allowing accelerated pixmap->screen blt's. QLinuxFbScreen and the accelerated screens use the Linux /dev/fb interface to get access to graphics memory and information about the characteristics of the device. The framebuffer device to open is specified by QWS_DISPLAY. Only QTransformedScreen implements the support for rotated framebuffers. QVfbScreen provides an X window containing an emulated framebuffer (a chunk of shared memory is set aside as the 'framebuffer' and blt'd into the X window) - this is intended as a debugging device allowing users to debug their applications under Qt/Embedded without leaving X. The accelerated screen drivers check to see if they can drive the device specified by QWS_CARD_SLOT (which defaults to the usual position of an AGP slot if not specified) and mmap its on-chip registers from /dev/mem. They may also do chip-specific setup (initialising registers to known values and so on). Finally, QScreen's are used to create new QScreenCursors and QGfxes. If you wish to modify the way pixmaps are allocated in memory, subclass or modify QLinuxFbScreen. If you're writing an accelerated driver you will need to subclass QScreen or QLinuxFbScreen.

QScreenCursor -> accelerated cursor -> QVfbCursor This handles drawing the on-screen mouse cursor, saving and restoring the screen under it for the non-accelerated cursor types. Subclassing QScreenCursor is optional in an accelerated driver (you would only want to do so if the hardware supports a hardware cursor).

QGfx -> RasterBase -> Raster -> accelerated driver -> QGfxVfb -> QGfxTransformedRaster This class encapsulates drawing operations, a little like a low-level QPainter. QGfxRaster and its descendants are specifically intended for drawing into a raw framebuffer. They can take an offset for drawing operations and a clipping region in order to support drawing into windows. You will need to subclass the QGfxRaster template in order to implement an accelerated driver. If you're brave modifying QGfxRaster would allow you to customise how drawing is done or add support for a new bit depth/pixel format.

QLock, QLockHolder This encapsulates a System V semaphore, used for synchronising access to memory shared between Qt/Embedded clients. QLockHolder is a utility class to make managing and destroying QLocks easier. There should be no need to modify this class unless porting Qt/Embedded to an operating system without System V IPC.

QDirectPainter This is a QPainter which also gives you a pointer to the framebuffer of the window it's pointing to, the window's clip region and so on. It's intended to easily allow you to do your own pixel-level manipulation of window contents. There should be no reason to modify this class.

QWSSoundServer,Client The Qt/Embedded server contains a simple sound player and mixer. Clients can request the server play sounds specified as files. There should be no need to modify this class unless porting Qt/Embedded to an operating system without a Linux-style /dev/dsp.

QWSWindow This contains the server's notion of an individual top level window - the region of the framebuffer it's allocated, the client that created it and so forth. There should be no reason to modify this class.

QWSKeyboardHandler->subtypes This handles keyboard/button input. QWSKeyboardHandler is subclassed to provide for reading /dev/tty, an arbitrary low-level USB event device (for USB keyboards) and some PDA button devices. Modifying QWSKeyboardHandler would allow you to support different types of keyboard (currently only a fairly standard US PC style keyboard is supported); subclassing it is the preferred way to handle non-pointer input devices.

QWSMouseHandler->QCalibratedMouseHandler->mouse types ->mouse types This handles mouse/touchpanel input. Descendants of QCalibratedMouseHandler make use of filtering code which pretends 'jittering' of the pointer on touchscreens; some embedded devices do this filtering in the kernel in which case the driver doesn't need to inherit from QCalibratedMouseHandler. Subclassing QCalibratedMouseHandler is preferred for touchpanels without kernel filtering; inheriting QWSMouseHandler is the way to add any other type of pointing device (pen tablets, touchscreens, mice, trackballs and so forth).

QWSDisplay This class exists only in the Qt/Embedded server and keeps track of all the top-level windows in the system, as well as the keyboard and mouse. You would only want to modify this if making deep and drastic modifications to Qt/Embedded window behaviour (alpha blended windows for example).

QWSServer This manages the Qt/Embedded server's Unix-domain socket connections to clients. It sends and receives QWS protocol events and calls QWSDisplay in order to do such things as change the allocation region of windows. The only reason to modify this would be to use something other than some sort of socket-like mechanism to communicate between Qt/Embedded applications (in which case modify QWSClient too). If you have something like Unix domain sockets, modify QWSSocket/QWSServerSocket instead. Don't add extra QWS events to communicate between applications, use QCOP instead.

QWSClient This encapsulates the client side of a Qt/Embedded connection and can marshal and demarshal events. There should be no reason to modify this except to use something radically different from Unix domain sockets to communicate between Qt/Embedded applications.

QWSDisplayData This manages a client's QWSClient, reading and interpreting events from the QWS server. It connects to the QWS server on application startup, getting information about the framebuffer and creating the memory manager. Other information about the framebuffer comes directly from /dev/fb in QLinuxFbScreen. There should be no reason to modify this.

QWSCommands These encapsulate the data sent to and from the QWS server. There should be no reason to modify them.

QCopChannel QCop is a simple IPC mechanism for communication between Qt/Embedded applications. String messages with optional binary data can be sent to different channels. The mechanism itself is designed to be bare-bones in order for users to build whatever mechanism they like on top of it.

QWSManager This provides Qt/Embedded window management, drawing a title bar and handling user requests to move, resize the window and so on. There should be no reason to modify it but you should subclass it if you want to modify window behaviour (point to click versus focus follows mouse, for instance).

QWSDecoration Descendants of this class are different styles for the Qt/Embedded window manager, for instance QWSWindowsDecoration draws Qt/Embedded window frames in the style of Windows CE. Subclass it in order to provide a new window manager appearance (the equivalent of a Windows XP or Enlightenment theme).

QWSPropertyManager This provides the QWS client's interface to the QWS property system (a simpler version of the X property system, it allows you to attach arbitrary data to top-level windows, keyed by an integer). There should be no reason to modify it.

QWSRegionManager Used by both client and server to help manage top-level window regions. There should be no reason to modify it.

QWSSocket, QWSServerSocket Provides Unix-domain sockets. Modify this if you're porting to a non-Unix OS but have something analogous to Unix-domain sockets (a byte-oriented, reliable, ordered transmission mechanism, although you can probably implement it with something like a message queue as well).

QCopChannel Class Reference

The QCopChannel class provides communication capabilities between several clients.

`#include <qcopchannel_qws.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QCopChannel** (const QCString & channel, QObject * parent = 0, const char * name = 0)
- virtual **~QCopChannel** ()
- QCString **channel** () const
- virtual void **receive** (const QCString & msg, const QByteArray & data)

Signals

- void **received** (const QCString & msg, const QByteArray & data)

Static Public Members

- bool **isRegistered** (const QCString & channel)
- bool **send** (const QCString & channel, const QCString & msg)
- bool **send** (const QCString & channel, const QCString & msg, const QByteArray & data)

Detailed Description

The QCopChannel class provides communication capabilities between several clients.

The Qt Cop (QCOP) is a COmmunication Protocol, allowing clients to communicate both within the same address space and between different processes.

Currently, this facility is only available on Qt/Embedded. On X11 and Windows we are exploring the use of existing standards such as DCOP and COM.

QCopChannel provides send() and isRegistered() which are static functions that are usable without an object.

The channel() function returns the name of the channel.

In order to *listen* to the traffic on a channel, you should either subclass QCopChannel and reimplement receive(), or connect() to the received() signal.

Member Function Documentation

QCopChannel::QCopChannel (const QCString & channel, QObject * parent = 0, const char * name = 0)

Constructs a QCop channel and registers it with the server using the name *channel*. The standard *parent* and *name* arguments are passed on to the QObject constructor.

QCopChannel::~QCopChannel () [virtual]

Destroys the client's end of the channel and notifies the server that the client has closed its connection. The server will keep the channel open until the last registered client detaches.

QCString QCopChannel::channel () const

Returns the name of the channel.

bool QCopChannel::isRegistered (const QCString & channel) [static]

Queries the server for the existance of *channel*.

Returns TRUE if *channel* is registered.

void QCopChannel::receive (const QCString & msg, const QByteArray & data) [virtual]

This virtual function allows subclasses of QCopChannel to process data received from their channel.

The default implementation emits the received() signal.

Note that the format of *data* has to be well defined in order to extract the information it contains.

Example:

```
void MyClass::receive( const QCString &msg, const QByteArray &data )
{
    QDataStream stream( data, IO_ReadOnly );
    if ( msg == "execute(QString,QString)" ) {
        QString cmd, arg;
        stream >> cmd >> arg;
        ...
    } else if ( msg == "delete(QString)" ) {
        QString filenname;
        stream >> filename;
        ...
    } else ...
}
```

This example assumes that the *msg* is a DCOP-style function signature and the *data* contains the function's arguments. (See send().)

Using the DCOP convention is a recommendation, but not a requirement. Whatever convention you use the sender and receiver *must* agree on the argument types.

See also send() [p. 29].

void QCopChannel::received (const QCString & msg, const QByteArray & data) [signal]

This signal is emitted with the *msg* and *data* whenever the receive() function gets incoming data.

bool QCopChannel::send (const QCString & channel, const QCString & msg, const QByteArray & data) [static]

Send the message *msg* on channel *channel* with data *data*. The message will be distributed to all clients subscribed to the channel.

Note that QDataStream provides a convenient way to fill the byte array with auxiliary data.

Example:

```
    QByteArray ba;
    QDataStream stream( ba, IO_WriteOnly );
    ba << QString("cat") << QString("file.txt");
    QCopChannel::send( "System/Shell", "execute(QString,QString)", ba );
```

Here the channel is "System/Shell". The *msg* is an arbitrary string, but in the example we've used the DCOP convention of passing a function signature. Such a signature is formatted as functionname(types) where types is a list of zero or more comma-separated type names, with no whitespace, no consts and no pointer or reference marks, i.e. no "*" or "&".

Using the DCOP convention is a recommendation, but not a requirement. Whatever convention you use the sender and receiver *must* agree on the argument types.

See also receive() [p. 28].

bool QCopChannel::send (const QCString & channel, const QCString & msg) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Send the message *msg* on channel *channel*. The message will be distributed to all clients subscribed to the channel.

See also receive() [p. 28].

QScreen Class Reference

The QScreen class and its descendants manage the framebuffer and palette.

`#include <qgfx_qws.h>`

Public Members

- **QScreen** (int display_id)
- virtual **~QScreen** ()
- virtual bool **initDevice** ()
- virtual bool **connect** (const QString & displaySpec)
- virtual void **disconnect** ()
- virtual int **initCursor** (void * end_of_location, bool init = FALSE)
- virtual void **shutdownDevice** ()
- virtual void **setMode** (int, int, int)
- virtual bool **supportsDepth** (int d) const
- virtual QGfx * **createGfx** (unsigned char * bytes, int w, int h, int d, int linestep)
- virtual QGfx * **screenGfx** ()
- virtual void **save** ()
- virtual void **restore** ()
- virtual void **blank** (bool on)
- virtual int **pixmapOffsetAlignment** ()
- virtual int **pixmapLinestepAlignment** ()
- virtual bool **onCard** (unsigned char * p) const
- virtual bool **onCard** (unsigned char * p, ulong & offset) const
- virtual void **set** (unsigned int, unsigned int, unsigned int, unsigned int)
- virtual int **alloc** (unsigned int r, unsigned int g, unsigned int b)
- int **width** () const
- int **height** () const
- int **depth** () const
- virtual int **pixmapDepth** () const
- int **pixelType** () const
- int **linestep** () const
- int **deviceWidth** () const
- int **deviceHeight** () const
- uchar * **base** () const
- virtual uchar * **cache** (int, int)
- virtual void **uncache** (uchar *)
- int **screenSize** () const
- int **totalSize** () const

- QRgb * **clut** ()
- int **numCols** ()
- virtual QSize **mapToDevice** (const QSize & s) const
- virtual QSize **mapFromDevice** (const QSize & s) const
- virtual QPoint **mapToDevice** (const QPoint &, const QSize &) const
- virtual QPoint **mapFromDevice** (const QPoint &, const QSize &) const
- virtual QRect **mapToDevice** (const QRect & r, const QSize &) const
- virtual QRect **mapFromDevice** (const QRect & r, const QSize &) const
- virtual QImage **mapToDevice** (const QImage & i) const
- virtual QImage **mapFromDevice** (const QImage & i) const
- virtual QRegion **mapToDevice** (const QRegion & r, const QSize &) const
- virtual QRegion **mapFromDevice** (const QRegion & r, const QSize &) const
- virtual int **transformOrientation** () const
- virtual bool **isTransformed** () const
- virtual bool **isInterlaced** () const
- virtual void **setDirty** (const QRect &)
- int * **opType** ()
- int * **lastOp** ()

Detailed Description

The QScreen class and its descendants manage the framebuffer and palette.

QScreens act as factories for the screen cursor and QGfx's. QLinuxFbScreen manages a Linux framebuffer; accelerated drivers subclass QLinuxFbScreen. There can only be one screen in a Qt/Embedded application.

See also Qt/Embedded.

Member Function Documentation

QScreen::QScreen (int display_id)

Create a screen; the *display_id* is the number of the Qt/Embedded server to connect to.

QScreen::~QScreen () [virtual]

Destroys a QScreen

int QScreen::alloc (unsigned int r, unsigned int g, unsigned int b) [virtual]

Given an RGB value *r g b*, return an index which is the closest match to it in the screen's palette. Used in paletted modes only.

uchar * QScreen::base () const

Returns a pointer to the start of the framebuffer.

void QScreen::blank (bool on) [virtual]

If *on* is true, blank the screen. Otherwise unblank it.

uchar * QScreen::cache (int, int) [virtual]

This function is used to store pixmaps in graphics memory for the use of the accelerated drivers. See QLinuxFbScreen (where the cacheing is implemented) for more information.

QRgb * QScreen::clut ()

Returns the screen's colour lookup table (colour palette). This is only valid in paletted modes (8bpp and lower).

bool QScreen::connect (const QString & displaySpec) [virtual]

This function is called by every Qt/Embedded application on startup. It maps in the framebuffer and in the accelerated drivers the graphics card control registers. *displaySpec* has the following syntax:

```
[gfx driver][:driver specific options][:display number]
```

for example if you want to use the mach64 driver on fb1 as display 2:

```
Mach64:/dev/fb1:2
```

displaySpec is passed in via the QWS_DISPLAY environment variable or the -display command line parameter.

QGfx * QScreen::createGfx (unsigned char * bytes, int w, int h, int d, int linestep) [virtual]

Creates a gfx on an arbitrary buffer *bytes*, width *w* and height *h* in pixels, depth *d* and *linestep* (length in bytes of each line in the buffer). Accelerated drivers can check to see if *bytes* points into graphics memory and create an accelerated Gfx.

int QScreen::depth () const

Gives the depth in bits per pixel of the framebuffer. This is the number of bits each pixel takes up rather than the number of significant bits, so 24bpp and 32bpp express the same range of colours (8 bits of red, green and blue)

int QScreen::deviceHeight () const

Gives the full height of the framebuffer device, as opposed to the height which Qt/Embedded will actually use. These can differ if the display is centered within the framebuffer.

int QScreen::deviceWidth () const

Gives the full width of the framebuffer device, as opposed to the width which Qt/Embedded will actually use. These can differ if the display is centered within the framebuffer.

void QScreen::disconnect () [virtual]

This function is called by every Qt/Embedded application just before exitting; it's normally used to unmap the framebuffer.

int QScreen::height () const

Gives the height in pixels of the framebuffer.

int QScreen::initCursor (void * end_of_location, bool init = FALSE) [virtual]

This is used to initialise the software cursor - *end_of_location* points to the address after the area where the cursor image can be stored. *init* is true for the first application this method is called from (the Qt/Embedded server), false otherwise.

bool QScreen::initDevice () [virtual]

This function is called by the Qt/Embedded server when initialising the framebuffer. Accelerated drivers use it to set up the graphics card.

bool QScreen::isInterlaced () const [virtual]

Returns TRUE if the display is interlaced (for instance a television screen); otherwise returns FALSE. If TRUE drawing is altered to look better on such displays.

bool QScreen::isTransformed () const [virtual]

Returns TRUE if the screen is transformed (for instance, rotated 90 degrees); otherwise returns FALSE. QScreen's version always returns FALSE.

int * QScreen::lastOp ()

Returns the screens last operation.

int QScreen::linestep () const

Returns the length in bytes of each scanline of the framebuffer.

QSize QScreen::mapFromDevice (const QSize & s) const [virtual]

Map a framebuffer coordinate to the coordinate space used by the application. Used by the rotated driver; the QScreen implementation simply returns *s*.

QPoint QScreen::mapFromDevice (const QPoint &, const QSize &) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Map a framebuffer coordinate to the coordinate space used by the application. Used by the rotated driver; the QScreen implementation simply returns the point.

QRect QScreen::mapFromDevice (const QRect & r, const QSize &) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Map a framebuffer coordinate to the coordinate space used by the application. Used by the rotated driver; the QScreen implementation simply returns *r*.

QImage QScreen::mapFromDevice (const QImage & i) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Transforms an image so that it matches the application coordinate space (e.g. rotating it 90 degrees counterclockwise). The QScreen implementation simply returns *i*.

QRegion QScreen::mapFromDevice (const QRegion & r, const QSize &) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Transforms a region so that it matches the application coordinate space (e.g. rotating it 90 degrees counterclockwise). The QScreen implementation simply returns *r*.

QSize QScreen::mapToDevice (const QSize & s) const [virtual]

Map a user coordinate to the one to actually be drawn. Used by the rotated driver; the QScreen implementation simply returns *s*.

QPoint QScreen::mapToDevice (const QPoint &, const QSize &) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Map a user coordinate to the one to actually be drawn. Used by the rotated driver; the QScreen implementation simply returns the point passed in.

QRect QScreen::mapToDevice (const QRect & r, const QSize &) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Map a user coordinate to the one to actually be drawn. Used by the rotated driver; the QScreen implementation simply returns *r*.

QImage QScreen::mapToDevice (const QImage & i) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Transforms an image so that it fits the device coordinate space (e.g. rotating it 90 degrees clockwise). The QScreen implementation simply returns *i*.

QRegion QScreen::mapToDevice (const QRegion & r, const QSize &) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Transforms a region so that it fits the device coordinate space (e.g. rotating it 90 degrees clockwise). The QScreen implementation simply returns *r*.

int QScreen::numCols ()

Returns the number of entries in the colour table returned by clut().

bool QScreen::onCard (unsigned char * p) const [virtual]

Returns true if the buffer pointed to by *p* is within graphics card memory, false if it's in main RAM.

bool QScreen::onCard (unsigned char * p, ulong & offset) const [virtual]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This checks whether the buffer specified by *p* is on the card (as per the other version of onCard) and returns an offset in bytes from the start of graphics card memory in *offset* if it is.

int * QScreen::opType ()

Returns the screen's operation type.

int QScreen::pixelType () const

Returns an integer (taking the same values as QGfx::PixelType) that specifies the pixel storage format of the screen.

int QScreen::pixmapDepth () const [virtual]

Gives the preferred depth for pixmaps. By default this is the same as the screen depth, but for the VGA16 driver it's 8bpp.

int QScreen::pixmapLinestepAlignment () [virtual]

Returns the value in bytes to which individual scanlines of pixmaps held in graphics card memory should be aligned. This is only useful for accelerated drivers. By default the value returned is 64 but it can be overridden by individual accelerated drivers.

int QScreen::pixmapOffsetAlignment () [virtual]

Returns the value in bytes to which the start address of pixmaps held in graphics card memory should be aligned. This is only useful for accelerated drivers. By default the value returned is 64 but it can be overridden by individual accelerated drivers.

void QScreen::restore () [virtual]

Restore the state of the graphics card from a previous save()

void QScreen::save () [virtual]

Saves the state of the graphics card - used so that, for instance, the palette can be restored when switching between linux virtual consoles. Hardware QScreen descendants should save register state here if necessary if switching between virtual consoles (for example to/from X) is to be permitted.

QGfx * QScreen::screenGfx () [virtual]

Returns a QGfx (normally a QGfxRaster) initialised to point to the screen, with an origin at 0,0 and a clip region covering the whole screen.

int QScreen::screenSize () const

Returns the size in bytes of the screen. This is always located at the beginning of framebuffer memory (i.e. at base()).

void QScreen::set (unsigned int, unsigned int, unsigned int, unsigned int) [virtual]

Set an entry in the colour palette. A noop in this class, implemented in QLinuxFbScreen.

void QScreen::setDirty (const QRect &) [virtual]

Indicates which section of the screen has been altered. Used by the VNC and VFB displays; the QScreen version does nothing.

void QScreen::setMode (int, int, int) [virtual]

This function can be used to set the framebuffer width, height and depth. It's currently unused.

void QScreen::shutdownDevice () [virtual]

Called by the Qt/Embedded server on shutdown; never called by a Qt/Embedded client. This is intended to support graphics card specific shutdown; the unaccelerated implementation simply hides the mouse cursor.

bool QScreen::supportsDepth (int d) const [virtual]

Returns true if the screen supports a particular color depth *d*. Possible values are 1,4,8,16 and 32.

int QScreen::totalSize () const

Returns the size in bytes of available graphics card memory, including the screen. Offscreen memory is only used by the accelerated drivers.

int QScreen::transformOrientation () const [virtual]

Used by the rotated server. The QScreeen implementation returns 0.

void QScreen::uncache (uchar *) [virtual]

This function is called on pixmap destruction to remove them from graphics card memory.

int QScreen::width () const

Gives the width in pixels of the framebuffer.

QWSServer Class Reference

The QWSServer class provides server-specific functionality in Qt/Embedded.

```
#include <qwindowsystem_qws.h>
```

Public Members

- **QWSServer** (int flags = 0, QObject * parent = 0, const char * name = 0)
- **~QWSServer** ()
- enum **ServerFlags** { DisableKeyboard = 0x01, DisableMouse = 0x02 }
- enum **GUIMode** { NoGui = FALSE, NormalGUI = TRUE, Server }
- class **KeyMap** { }
- class **KeyboardFilter** { }
- QWSWindow * **windowAt** (const QPoint & pos)
- const QPtrList<QWSWindow> & **clientWindows** ()
- void **openMouse** ()
- void **closeMouse** ()
- void **openKeyboard** ()
- void **closeKeyboard** ()
- void **refresh** ()
- void **refresh** (QRegion & r)
- void **enablePainting** (bool e)
- QWSPropertyManager * **manager** ()
- enum **WindowEvent** { Create = 0x01, Destroy = 0x02, Hide = 0x04, Show = 0x08, Raise = 0x10, Lower = 0x20, Geometry = 0x40 }

Signals

- void **windowEvent** (QWSWindow * w, QWSServer::WindowEvent e)
- void **newChannel** (const QString &)

Static Public Members

- void **sendKeyEvent** (int unicode, int keycode, int modifiers, bool isPress, bool autoRepeat)
- const KeyMap * **keyMap** ()
- void **setKeyboardFilter** (KeyboardFilter * f)
- void **setDefaultMouse** (const char * m)
- void **setDefaultKeyboard** (const char * k)

- void **setMaxWindowRect** (const QRect & r)
- void **sendMouseEvent** (const QPoint & pos, int state)
- void **setDesktopBackground** (const QImage & img)
- void **setDesktopBackground** (const QColor & c)
- QWSMouseHandler * **mouseHandler** ()
- QWSKeyboardHandler * **keyboardHandler** ()
- void **setKeyboardHandler** (QWSKeyboardHandler * kh)
- void **setScreenSaverInterval** (int ms)
- bool **screenSaverActive** ()
- void **screenSaverActivate** (bool activate)

Detailed Description

The QWSServer class provides server-specific functionality in Qt/Embedded.

When you run a Qt/Embedded application, it either runs as a server or connects to an existing server. If it runs as a server, some additional operations are provided via the QWSServer class.

This class is instantiated by QApplication for Qt/Embedded server processes. You should never construct this class yourself.

A pointer to the QWSServer instance can be obtained via the global qwsServer variable.

The mouse and keyboard devices can be opened with openMouse() and openKeyboard(). (Close them with close-Mouse() and closeKeyboard().)

The display is refreshed with refresh(), and painting can be enabled or disabled with enablePainting().

Obtain the list of client windows with clientWindows() and find out which window is at a particular point with windowAt().

Many static functions are provided, for example, setKeyboardFilter(), setKeyboardHandler(), setDefaultKeyboard() and setDefaultMouse().

The size of the window rectangle can be set with setMaxWindowRect(), and the desktop's background can be set with setDesktopBackground().

The screen saver is controlled with setScreenSaverInterval() and screenSaverActivate().

See also Qt/Embedded.

Member Type Documentation

QWSServer::GUIMode

This determines what sort of QWS server to create:

- `QWSServer::NoGui` - This is used for non-graphical Qt applications.
- `QWSServer::NormalGUI` - A normal Qt/Embedded application (not the server).
- `QWSServer::Server` - A Qt/Embedded server (e.g. if -qws has been specified on the command line.

QWSServer::ServerFlags

This enum is used to pass various options to the window system server. Currently defined are:

- `QWSServer::DisableKeyboard` - Ignore all keyboard input.
- `QWSServer::DisableMouse` - Ignore all mouse input.

QWSServer::WindowEvent

This specifies what sort of event has occurred to a top level window:

- `QWSServer::Create` - A new window has been created (QWidget constructor).
- `QWSServer::Destroy` - The window has been closed and deleted (QWidget destructor).
- `QWSServer::Hide` - The window has been hidden with QWidget::hide().
- `QWSServer::Show` - The window has been shown with QWidget::show() or similar.
- `QWSServer::Raise` - The window has been raised to the top of the desktop.
- `QWSServer::Lower` - The window has been lowered.
- `QWSServer::Geometry` - The window has changed size or position.

Member Function Documentation

QWSServer::QWSServer (int flags = 0, QObject * parent = 0, const char * name = 0)

Construct a QWSServer class with parent *parent*, called *name* and flags *flags*.

Warning: This class is instantiated by QApplication for Qt/Embedded server processes. You should never construct this class yourself.

QWSServer::~QWSServer ()

Destruct QWSServer

const QPtrList<QWSWindow> & QWSServer::clientWindows ()

Returns the list of top-level windows. This list will change as applications add and remove wigdets so it should not be stored for future use. The windows are sorted in stacking order from top-most to bottom-most.

void QWSServer::closeKeyboard ()

Closes keyboard device(s).

void QWSServer::closeMouse ()

Closes the pointer device(s).

void QWSServer::enablePainting (bool e)

If *e* is TRUE, painting on the display is enabled; if *e* is FALSE, painting is disabled.

const KeyMap * QWSServer::keyMap () [static]

Returns the keyboard mapping table used to convert keyboard scancodes to Qt keycodes and unicode values. It's used by the keyboard driver in qkeyboard_qws.cpp.

QWSKeyboardHandler * QWSServer::keyboardHandler () [static]

Returns the primary keyboard handler.

QWSPropertyManager * QWSServer::manager ()

Returns the QWSPropertyManager, which is used for implementing X11-style window properties.

QWSMouseHandler * QWSServer::mouseHandler () [static]

Returns the primary mouse handler.

void QWSServer::openKeyboard ()

Opens the keyboard device(s).

void QWSServer::openMouse ()

Opens the mouse device(s).

void QWSServer::refresh ()

Refreshes the entire display.

void QWSServer::refresh (QRegion & r)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Refreshes the region *r*.

void QWSServer::screenSaverActivate (bool activate) [static]

If *activate* is TRUE the screensaver is activated immediately; if *activate* is FALSE the screensaver is deactivated.

bool QWSServer::screenSaverActive () [static]

Returns TRUE if the screensaver is active (i.e. the screen is blanked); otherwise returns FALSE.

void QWSServer::sendKeyEvent (int unicode, int keycode, int modifiers, bool isPress, bool autoRepeat) [static]

Send a key event. You can use this to send key events generated by "virtual keyboards". *unicode* is the unicode value of the key to send, *keycode* the Qt keycode (e.g. Key_Left), *modifiers* indicates whether, Shift/Alt/Ctrl keys are pressed, *isPress* is TRUE if this is a key down event and FALSE if it's a key up event, and *autoRepeat* is TRUE if this is an autorepeat event (i.e. the user has held the key down and this is the second or subsequent key event being sent).

void QWSServer::sendMouseEvent (const QPoint & pos, int state) [static]

Send a mouse event. *pos* is the screen position where the mouse event occurred and *state* is a mask indicating which buttons are pressed.

void QWSServer::setDefaultKeyboard (const char * k) [static]

Set the keyboard driver to *k*, e.g. if $QWS_KEYBOARD is not defined. The default is platform-dependant.

void QWSServer::setDefaultMouse (const char * m) [static]

Set the mouse driver *m* to use if $QWS_MOUSE_PROTO is not defined. The default is platform-dependent.

void QWSServer::setDesktopBackground (const QImage & img) [static]

Sets the image *img* to be used as the background in the absence of obscuring windows.

void QWSServer::setDesktopBackground (const QColor & c) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the color *c* to be used as the background in the absence of obscuring windows.

void QWSServer::setKeyboardFilter (KeyboardFilter * f) [static]

Sets a filter *f* to be invoked for all key events from physical keyboard drivers (events sent via processKeyEvent()). The filter is not invoked for keys generated by virtual keyboard drivers (events sent via sendKeyEvent()).

void QWSServer::setKeyboardHandler (QWSKeyboardHandler * kh) [static]

Sets the primary keyboard handler to *kh*.

void QWSServer::setMaxWindowRect (const QRect & r) [static]

Sets the area of the screen which Qt/Embedded applications will consider to be the maximum area to use for windows to *r*.

See also QWidget::showMaximized() [Widgets with Qt].

void QWSServer::setScreenSaverInterval (int ms) [static]

Sets the timeout for the screensaver to *ms* milliseconds. A setting of zero turns off the screensaver.

QWSWindow * QWSServer::windowAt (const QPoint & pos)

Returns the window containing the point *pos* or 0 if there is no window under the point.

void QWSServer::windowEvent (QWSWindow * w, QWSServer::WindowEvent e) [signal]

This signal is emitted whenever something happens to a top level window (e.g. it's created or destroyed). *w* is the window to which the event of type *e* has occurred.

QWSDecoration Class Reference

The QWSDecoration class allows the appearance of the Qt/Embedded Window Manager to be customized.

`#include <qwsdecoration_qws.h>`

Public Members

- **QWSDecoration** ()
- virtual **~QWSDecoration** ()
- enum **Region** { None = 0, All = 1, Title = 2, Top = 3, Bottom = 4, Left = 5, Right = 6, TopLeft = 7, TopRight = 8, BottomLeft = 9, BottomRight = 10, Close = 11, Minimize = 12, Maximize = 13, Normalize = 14, Menu = 15, LastRegion = Menu }
- virtual QRegion **region** (const QWidget * widget, const QRect & rect, Region type = All)
- virtual void **close** (QWidget * widget)
- virtual void **minimize** (QWidget * widget)
- virtual void **maximize** (QWidget * widget)
- virtual QPopupMenu * **menu** (const QWidget *, const QPoint &)
- virtual void **paint** (QPainter * painter, const QWidget * widget)
- virtual void **paintButton** (QPainter * painter, const QWidget * widget, Region type, int state)

Detailed Description

The QWSDecoration class allows the appearance of the Qt/Embedded Window Manager to be customized.

Qt/Embedded provides window management to top level windows. The appearance of the borders and buttons (the decoration) around the managed windows can be customized by creating your own class derived from QWS-Decoration and overriding a few methods.

This class is non-portable. It is available *only* in Qt/Embedded.

See also QApplication::qwsSetDecoration() and Qt/Embedded.

Member Type Documentation

QWSDecoration::Region

This enum describes the regions in the window decorations.

- `QWSDecoration::None` - used internally.
- `QWSDecoration::All` - the entire region used by the window decoration.

- `QWSDecoration::Title` - Displays the window title and allows the window to be moved by dragging.
- `QWSDecoration::Top` - allows the top of the window to be resized.
- `QWSDecoration::Bottom` - allows the bottom of the window to be resized.
- `QWSDecoration::Left` - allows the left edge of the window to be resized.
- `QWSDecoration::Right` - allows the right edge of the window to be resized.
- `QWSDecoration::TopLeft` - allows the top-left of the window to be resized.
- `QWSDecoration::TopRight` - allows the top-right of the window to be resized.
- `QWSDecoration::BottomLeft` - allows the bottom-left of the window to be resized.
- `QWSDecoration::BottomRight` - allows the bottom-right of the window to be resized.
- `QWSDecoration::Close` - clicking in this region closes the window.
- `QWSDecoration::Minimize` - clicking in this region minimizes the window.
- `QWSDecoration::Maximize` - clicking in this region maximizes the window.
- `QWSDecoration::Normalize` - returns a maximized window to previous size.
- `QWSDecoration::Menu` - clicking in this region opens the window operations menu.

Member Function Documentation

QWSDecoration::QWSDecoration ()

Constructs a decorator.

QWSDecoration::~QWSDecoration () [virtual]

Destroys a decorator.

void QWSDecoration::close (QWidget * widget) [virtual]

Called when the user clicks in the Close region.

widget is the QWidget to be closed.

The default behaviour is to close the widget.

void QWSDecoration::maximize (QWidget * widget) [virtual]

Called when the user clicks in the Maximize region.

widget is the QWidget to be maximized.

The default behaviour is to resize the widget to be full-screen. This method can be overridden to, e.g. avoid launch panels.

QPopupMenu * QWSDecoration::menu (const QWidget *, const QPoint &) [virtual]

Called to create a QPopupMenu containing the valid menu operations.

The default implementation adds all possible window operations.

void QWSDecoration::minimize (QWidget * widget) [virtual]

Called when the user clicks in the Minimize region.

widget is the QWidget to be minimized.

The default behaviour is to ignore this action.

void QWSDecoration::paint (QPainter * painter, const QWidget * widget) [virtual]

Override to paint the border and title decoration around *widget* using *painter*.

void QWSDecoration::paintButton (QPainter * painter, const QWidget * widget, Region type, int state) [virtual]

Override to paint a button *type* using *painter*.

widget is the widget whose button is to be drawn. *state* is the state of the button. It can be a combination of the following ORed together:

- `QWSButton::MouseOver`
- `QWSButton::Clicked`
- `QWSButton::On`

QRegion QWSDecoration::region (const QWidget * widget, const QRect & rect, Region type = All) [virtual]

Returns the requested region *type* which will contain *widget* with geometry *rect*.

QWSKeyboardHandler Class Reference

The QWSKeyboardHandler class implements the keyboard driver/handler for Qt/Embedded.

`#include <qkeyboard_qws.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QWSKeyboardHandler** ()
- virtual **~QWSKeyboardHandler** ()

Protected Members

- virtual void **processKeyEvent** (int unicode, int keycode, int modifiers, bool isPress, bool autoRepeat)

Detailed Description

The QWSKeyboardHandler class implements the keyboard driver/handler for Qt/Embedded.

The keyboard driver/handler handles events from system devices and generates key events.

A QWSKeyboardHandler will usually open some system device in its constructor, create a QSocketNotifier on that opened device and when it receives data, it will call processKeyEvent() to send the event to Qt/Embedded for relaying to clients.

See also Qt/Embedded.

Member Function Documentation

QWSKeyboardHandler::QWSKeyboardHandler ()

Constructs a keyboard handler. The handler *may* be passed to the system for later destruction with QWSServer::setKeyboardHandler(), although even without doing this, the handler can function, calling processKeyEvent() to emit events.

QWSKeyboardHandler::~QWSKeyboardHandler () [virtual]

Destroys a keyboard handler. Note that if you have called QWSServer::setKeyboardHandler(), you may not delete the handler.

void QWSKeyboardHandler::processKeyEvent (int unicode, int keycode, int modifiers, bool isPress, bool autoRepeat) [virtual protected]

Subclasses call this function to send a key event. The server may additionally filter the event before sending it on to applications.

- *unicode* is the Unicode value for the key, or 0xFFFF is none is appropriate.
- *keycode* is the Qt keycode for the key (see Qt::Key). for the list of codes).
- *modifiers* is the set of modifier keys (see Qt::Modifier).
- *isPress* says whether this is a press or a release.
- *autoRepeat* says whether this event was generated by an auto-repeat mechanism, or an actual key press.

QWSMouseHandler Class Reference

The QWSMouseHandler class is a mouse driver/handler for Qt/Embedded.

`#include <qwsmouse_qws.h>`

Inherits QObject [Additional Functionality with Qt].

Public Members

- **QWSMouseHandler** ()
- virtual **~QWSMouseHandler** ()
- virtual void **clearCalibration** ()
- virtual void **calibrate** (QWSPointerCalibrationData *)

Protected Members

- void **mouseChanged** (const QPoint & pos, int bstate)

Detailed Description

The QWSMouseHandler class is a mouse driver/handler for Qt/Embedded.

The mouse driver/handler handles events from system devices and generates mouse events.

A QWSMouseHandler will usually open some system device in its constructor, create a QSocketNotifier on that opened device and when it receives data, it will call mouseChanged() to send the event to Qt/Embedded for relaying to clients.

See also Qt/Embedded.

Member Function Documentation

QWSMouseHandler::QWSMouseHandler ()

Constructs a mouse handler. This becomes the primary mouse handler.

Note that once created, mouse handlers are controlled by the system and should not be deleted.

QWSMouseHandler::~QWSMouseHandler () [virtual]

Destroys the mouse handler. You should not call this directly.

void QWSMouseHandler::calibrate (QWSPointerCalibrationData *) [virtual]

This method is reimplemented in the calibrated mouse handler to set calibration information (from, for instance, the QPE calibration screen). This version does nothing.

void QWSMouseHandler::clearCalibration () [virtual]

This method is reimplemented in the calibrated mouse handler to clear calibration information. This version does nothing.

void QWSMouseHandler::mouseChanged (const QPoint & pos, int bstate) [protected]

To be called by the mouse handler to signal that the mouse is at position *pos* and the mouse buttons are in state *bstate*.

QWSWindow Class Reference

The QWSWindow class provides server-specific functionality in Qt/Embedded.

`#include <qwindowsystem_qws.h>`

Public Members

- **QWSWindow** (int i, QWSClient * client)
- **~QWSWindow** ()
- int **winId** () const
- const QString & **name** () const
- const QString & **caption** () const
- QWSClient * **client** () const
- QRegion **requested** () const
- QRegion **allocation** () const
- bool **isVisible** () const
- bool **isPartiallyObscured** () const
- bool **isFullyObscured** () const
- void **raise** ()
- void **lower** ()
- void **show** ()
- void **hide** ()
- void **setActiveWindow** ()

Detailed Description

The QWSWindow class provides server-specific functionality in Qt/Embedded.

When you run a Qt/Embedded application, it either runs as a server or connects to an existing server. If it runs as a server, some additional functionality is provided by the QWSServer class.

This class maintains information about each window and allows operations to be performed on the windows.

You can get the window's name(), caption() and winId(), along with the client() that owns the window.

The region the window wants to draw on is returned by requested(); the region that the window is allowed to draw on is returned by allocation().

The visibility of the window can be determined using isVisible(), isPartiallyObscured() and isFullyObscured(). Visibility can be changed using raise(), lower(), show(), hide() and setActiveWindow().

See also Qt/Embedded.

Member Function Documentation

QWSWindow::QWSWindow (int i, QWSClient * client)

Constructs a new top-level window, associated with the client *client* and giving it the id *i*.

QWSWindow::~QWSWindow ()

Destructor.

QRegion QWSWindow::allocation () const

Returns the region that the window is allowed to draw onto including any window decorations but excluding regions covered by other windows.

See also requested() [p. 53].

const QString & QWSWindow::caption () const

Returns this window's caption.

QWSClient * QWSWindow::client () const

Returns the QWSClient that owns this window.

void QWSWindow::hide ()

Hides the window.

bool QWSWindow::isFullyObscured () const

Returns TRUE is the window is completely obsured by another window or by the bounds of the screen; otherwise returns FALSE.

bool QWSWindow::isPartiallyObscured () const

Returns TRUE is the window is partially obsured by another window or by the bounds of the screen; otherwise returns FALSE.

bool QWSWindow::isVisible () const

Returns TRUE if the window is visible; otherwise returns FALSE.

void QWSWindow::lower ()

Lowers the window below other windows.

const QString & QWSWindow::name () const

Returns the name of this window.

void QWSWindow::raise ()

Raises the window above all other windows except "Stay on top" windows.

QRegion QWSWindow::requested () const

Returns the region that the window has requested to draw onto including any window decorations.

See also allocation() [p. 52].

void QWSWindow::setActiveWindow ()

Make this the active window (i.e. sets the keyboard focus to this window).

void QWSWindow::show ()

Shows the window.

int QWSWindow::winId () const

Returns the Id of this window.

Installing Qt/Embedded

This installation procedure is written for Linux. It may need to be modified for other platforms.

1. Unpack the archive if you have not done so already

   ```
   cd
   gunzip qt-embedded-version-commercial.tar.gz    # uncompress the archive
   tar xf qt-embedded-version-commercial.tar       # unpack it
   ```

   This document assumes that the archive is installed as `~/qt-`*`version`*.
2. Compile the Qt/Embedded library and examples.

   ```
   cd ~/qt-version
   export QTDIR=~/qt-version
   ./configure
   make
   ```

   The configuration system is designed to allow platform-specific options to be added, but in general all Linux system which have framebuffer support can use the "linux-generic-g++" platform. The configuration system also supports cross-compilers: to build *on* Linux/x86 *for* the Linux/MIPSEL target, you would use:

   ```
   ./configure -platform linux-x86-g++ -xplatform linux-mips-g++
   ```

   Only a small number of configurations are predefined, all much the same. Configurations files are found in `configs/`.
3. Enable framebuffer support.

   You may need to recompile your kernel to enable the framebuffer. This document does not describe how to do this; the HOWTO-Framebuffer page contains a short description. (You should see a penguin logo at boot time when the frame buffer is enabled.)

   For Matrox G100/G200/G400 use the matrox frame buffer driver.

   For NVidia TNT cards use the nvidia frame buffer driver.

   For Mach64 and most other cards, use the vesafb driver.

   Note that some cards are only supported in VGA16 mode, this will not work with the current version of Qt/Embedded, since VGA/16 is not yet supported. You may need to upgrade your kernel, or even switch to an experimental kernel.

   The frame buffer must also be enabled with a boot parameter. See `/usr/src/linux/Documentation/fb` for details.

   The fbset program, which should be included in Linux distributions, may be used to switch video modes without rebooting the system. The video mode active when the server is started will be used. (8-bit modes are still experimental.) NOTE: fbset does not work with the vesafb driver.
4. Change permissions.

   To run Qt/Embedded, you need write access to the framebuffer device `/dev/fb0`.

   You also need read access to the mouse device. (Note that `/dev/mouse` is normally a symbolic link; the actual mouse device must be readable.)

5. How to run the demonstration program.

   Log into a virtual console and do:

   ```
       cd ~/qt-version/
       ./start-demo
   ```

6. Miscellaneous troubleshooting and known bugs.

   To kill gpm, run the following command as root:

   ```
       gpm -k
   ```

   In some cases, if the server does not work, it will work when run as root.

   Some example programs may not compile with GCC 2.95.

   Show processes using the framebuffer:

   ```
   fuser -v /dev/fb0
   ```

   Kill such processes:

   ```
   fuser -vk /dev/fb0
   ```

   or harsher:

   ```
   fuser -k -KILL /dev/fb0
   ```

   Show existing semaphores:

   ```
   ipcs
   ```

   Remove semaphores:

   ```
   ipcrm
   ```

   The communication between client and server is done through the named pipe `/tmp/.QtEmbedded`; sometimes it may need to be deleted (eg. if you run Qt/Embedded as root then later as an unprivileged user).

7. Customization.

   The Qt/Embedded library can be reduced in size by removing unneeded features.

Qt/Embedded environment variables

QWS_SW_CURSOR If defined, always use a software mouse cursor even when using an accelerated driver that supports a hardware cursor

QWS_DISPLAY Defines display type and framebuffer - e.g. Voodoo3 Mach64:/dev/fb1 Defaults to unaccelerated Linux framebuffer driver on /dev/fb0. Valid drivers are QVfb, VGA16, LinuxFb (unaccelerated Linux framebuffer), Mach64 (accelerated for ATI Mach64 cards such as the Rage Pro), Voodoo3 (accelerated for the 3dfx Voodoo 3, should also work on Voodoo Banshee), Matrox (should work on all Matrox graphics cards since the Matrox Millennium), Transformed(for rotated displays), SVGALIB and VNC. Transformed displays have a special format - within the specification should be a multiple of 90 degrees rotation specified as Rot, for instance Transformed:Rot90.

QTDIR If defined tells Qt/Embedded to where to find its fonts - fontdir should be in $QTDIR/etc/fonts/. If undefined it's assumed to be /usr/local/qt-embedded

QWS_SIZE If defined forces Qt/Embedded into a window of x size centred within the screen, e.g. 320x200

QWS_NOMTRR If defined, don't use Memory Type Range Registers to define the framebuffer as write-combined on x86. Write-combining speeds up graphics output.

QWS_CARD_SLOT Tells the accelerated drivers which card to attempt to accelerate. This should be a path in /proc/bus/pci. It defaults to /proc/bus/pci/01/00.0 - the first device on the second PCI bus in the system, which is normally the AGP card.

QWS_USB_KEYBOARD If defined, instead of opening /dev/tty open the USB low-level event device defined in QWS_USB_KEYBOARD (e.g. /dev/input/event0) - this is useful if you wish to run X and Qt/Embedded side by side on different framebuffers.

QWS_MOUSE_PROTO Defined as :, e.g. Microsoft:/dev/ttyS0. If you want to use a USB mouse directly (separate from X) use MouseMan:/dev/input/mouse0 or similar. Valid mouse protocls are Auto (automatically sense protocol), MouseMan, IntelliMouse, Microsoft, QVfbMouse (only useful with QVfb) and TPanel, a sample touch panel driver.

QWS_KEYBOARD Defines the keyboard type. Multiple keyboards can be handled at once, input will be read from all of them. Valid values: Buttons (an iPaq button device if QT_QWS_IPAQ is compiled, otherwise one for the Cassiopeia), QVfbKeyboard (only useful with QVfb), and TTY (either a USB keyboard or /dev/tty depending if QWS_USB_KEYBOARD is defined)

Qt/Embedded Performance Tuning

When building embedded applications on low-powered devices, a number of options are available that would not be considered in a desktop application environment. These options reduce the memory and/or CPU requirements at the cost of other factors.

- **Tuning the functionality of Qt**
- General programming style
- Static vs. Dynamic linking
- Alternative memory allocation

General programming style

The following guidelines will improve CPU performance:

- Create dialogs and widgets once, then QWidget::hide() and QWidget::show() them, rather than creating them and deleting them every time they are needed. This will use a little more memory, but will be much faster. Try to create them the first time "lazily" to avoid slow startup (only create the Find dialog the first time the user invokes it).

Static vs. Dynamic linking

Much CPU and memory is used by the ELF linking process. You can make significant savings by using a static build of your application suite. This means that rather than having a dynamic library (`libqte.so`) and a collection of executables which link dynamically to that library, you build all the applications into a single executable and statically link that with a static library (`libqt.a`). This improves start-up time, and reduces memory usage, at the expense of flexibility (to add a new application, you must recompile the single executable) and robustness (if one application has a bug, it might harm other applications). If you need to install end-user applications, this may not be an option, but if you are building a single application suite for a device with limited CPU power and memory, this option could be very beneficial.

To compile Qt as a static library, add the `-static` options when you run configure.

To build your application suite as an all-in-one application, design each application as a stand-alone widget or set of widgets, with only minimal code in the main() function. Then, write an application that gives some way to choose among the applications (eg. a QIconView). The QPE is an example of this. It can be built either as a set of dynamically-linked executables, or as a single static application.

Note that you should generally still link dynamically against the standard C library and any other libraries which might be used by other applications on your device.

Alternative memory allocation

We have found that the libraries shipped with some C++ compilers on some platforms have poor performance in the built-in "new" and "delete" operators. You might gain performance by re-implementing these functions. For example, you can switch to the plain C allocators by adding the following to your code:

```
void* operator new[]( size_t size )
{
    return malloc( size );
}

void* operator new( size_t size )
{
    return malloc( size );
}

void operator delete[]( void *p )
{
    free( p );
}

void operator delete[]( void *p, size_t size )
{
    free( p );
}

void operator delete( void *p )
{
    free( p );
}

void operator delete( void *p, size_t size )
{
    free( p );
}
```

Porting your applications to Qt/Embedded

Existing Qt applications should require no porting provided there is no platform dependent code. Platform dependent code includes system calls, calls to the underlying window system (Windows or X11), and Qt platform specific methods such as QApplication::x11EventFilter().

For cases where it is necessary to use platform dependent code there are macros defined that can be used to enable/disable code for each platform using `#ifdef` directives:

Platform	**Macro**
Qt/X11	Q_WS_X11
Qt/Windows	Q_WS_WIN
Qt/Embedded	Q_WS_QWS

QEmbed - File and Image Embedder

The QEmbed tool, found in `qt/tools/qembed`, converts arbitrary files into C++ code. This is useful for including image files and other resources directly into your application rather than loading the data from external files.

QEmbed can also generate uncompressed versions of images that can be included directly into your application, thus avoiding both the external file and the need to parse the image file format. This is useful for small images such as icons for which compression is not a great gain.

Usage

```
qembed [ general-files ] [ --images image-files ]
```

`general-files` These files can be any type of file. `--images` *`image-files`* These files must be in image formats supported by Qt.

Output

The output from QEmbed is a C++ header file which you should include in a C++ source file. In that file, you should make a wrapper function that suites your application. Two functions are provided, and your wrapper function could just call one of these, or you can implement your own. To use the functions in the simplest way:

```
#include "generated_qembed_file.h"

// Just call the generated function
// "name" is the original image filename WITHOUT the extension
QImage myFindImage(const char* name)
{
    return qembed_findImage(name);
}

// Just call the generated function
// "name" is the original filename WITH the extension
QByteArray myFindData(const char* name)
{
    return qembed_findData(name);
}
```

Alternatively, look at the output from QEmbed and write a function tailored to your needs.

Issues to be aware of in porting Qt/Embedded to another OS

Qt/Embedded is designed to be reasonably platform-independent. However, there is currently only a Linux implementation. The following dependencies will need to be addressed if you intend to port to another operating system (files that you need to modify follow each section):

System V IPC (shared memory and semaphores) is used to share window regions between client and server. You will need to provide something similar unless you want a single-application setup (i.e. running only one program, which is the server). System V semaphores are also used for synchronising access to the framebuffer. (qwindowsystem_qws.cpp, qwsregionmanager_qws.cpp, qapplication_qws.cpp, qlock_qws.cpp)

Unix-domain sockets are used to communicate things like keyboard events, requests to raise windows and QCOP messages between applications. Again, you will need to provide something similar unless you want a single-application setup. It should be possible to implement something like this using message queues or similar mechanisms; with the exception of QCOP messages (which are generated by client applications and not Qt/Embedded) individual messages should be no more than a few bytes in length. (qwssocket_qws.cpp)

The Linux framebuffer device is used to map in the drawing area. You will need to replace it (by creating a new class of QScreen) with something else giving a byte pointer to a memory-mapped framebuffer, plus information about width, height and bit depth (which most likely you can simply hard-code). If your framebuffer is not memory-mapped or is in an unsupported format or depth you will need to modify QGfxRaster as well. (qgfxlinuxfb_qws.cpp)

The accelerated drivers currently use the Linux QScreen and use /proc/bus/pci to map in PCI config space. However, these are only example drivers; the chances are that you will need to write your own driver in any case, and you will need to provide your own way to map in control registers. (qgfxmach64_qws.cpp, qgfxvoodoo_qws.cpp, qgfxmatrox_qws.cpp)

Qt/Embedded makes use of the standard C library and some Posix functions. Mostly the latter are concentrated in platform dependent code anyway (e.g. mmap() to map in the Linux framebuffer).

Sound uses a Linux /dev/dsp style device. If you want to use the Qt/Embedded sound server you'll need to reimplement it. (qsoundqss_qws.cpp)

Qt/Embedded uses select() to implement QSocketDevices and listen for events to/from the Qt/Embedded server application. (qapplication_qws.cpp)

Index