QNX® Neutrino® RTOS
Customizing a BSP

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: March 06, 2020

Contents

About this guide

While QNX provides Board Support Packages (BSPs) for many common platforms and their individual variants, in some cases, you need a BSP for a board that QNX does not provide. If this is the case, you can modify a QNX BSP or develop your own.

Contents of this guide

For information about:	See:
Previous knowledge and experience required for customizing a QNX BSP	*Prerequisites*
The process for creating a BSP for a custom board	*The development process*
Locating and modifying the appropriate OS image buildfile	*OS Image Buildfiles*
Startup code elements that are often modified for custom BSPs	*Startup*
Graphics resources that are often modified for custom BSPs	*Graphics*
The framework you use to modify an I2C driver	*I2C*
Customizing network drivers	*Networking*
Reference information for customizing a generic serial port driver	*Generic serial port*
Reference information for customizing a SPI driver	*SPI*

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Constants	NULL
Data types	`unsigned short`
Environment variables	*PATH*
File and pathnames	**/dev/null**
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective** → **Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

DANGER: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1 Prerequisites

Building a BSP requires previous knowledge of and experience with QNX Neutrino RTOS and related development tools, which includes familiarity with the following technology:

- C programming
- QNX Software Development Platform (SDP) development environment, with or without QNX Momentics Tool Suite
- Writing a QNX Neutrino resource manager (device driver)

In addition, you should be generally familiar with creating BSPs for embedded systems.

The *Building Embedded Systems* guide provides detailed descriptions of QNX BSP components and the BSP build process. Because this guide includes many references to Building Embedded Systems guide, it is helpful to be familiar with it before you start.

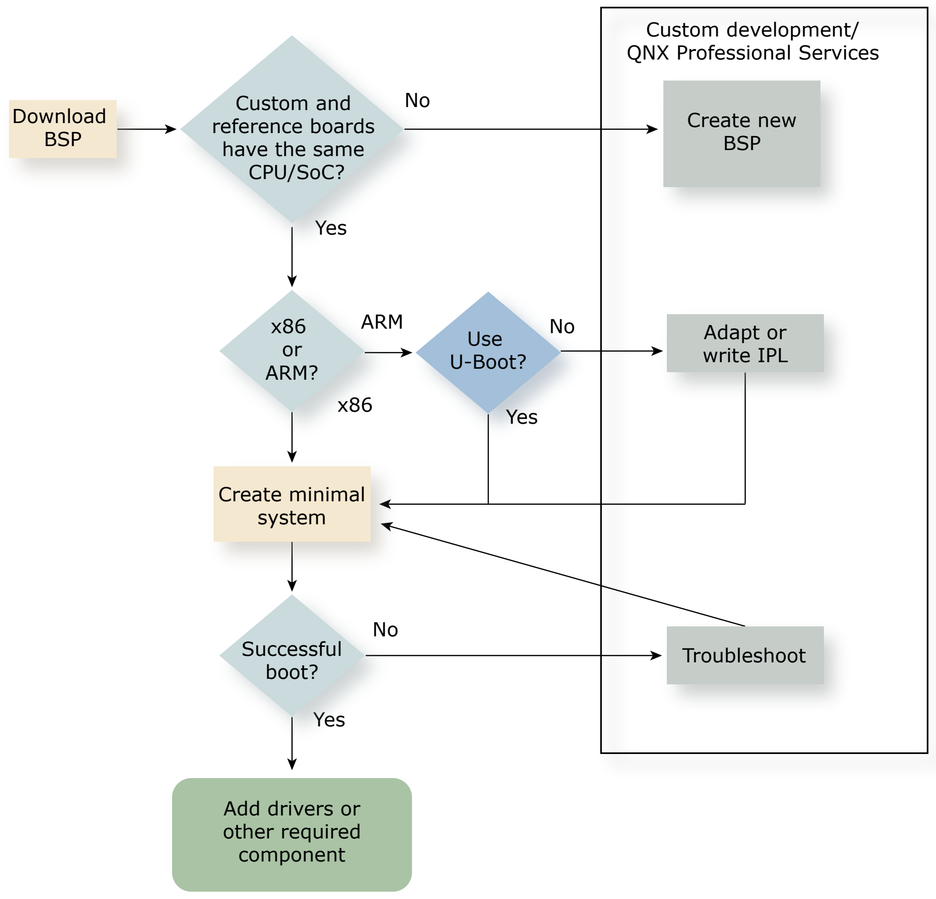
If you do not have the required knowledge and experience, QNX offers standardized training, including Realtime Programming for the QNX® Neutrino® RTOS and Writing Drivers for the QNX Neutrino RTOS.

Chapter 2
The development process

BSP development begins with a reference BSP from QNX.

The availability of compatible BSP components and your board's specifications determine which options you select for loading and booting the OS. Generally speaking, the goal is to successfully complete a step of the boot process before proceeding to the next step.

An alternative to developing custom BSP components yourself is QNX Professional Services, which is available to create custom BSP content, including drivers.



Reference BSP

Before you attempt to customize an existing BSP, familiarize yourself with the overall process of booting a QNX system on a target platform. This process includes obtaining a BSP and using it to build a bootable image that you transfer from your host system to your target board.

You can then select a BSP to use as a starting point for your custom BSP.

For information about obtaining a BSP, its structure, and the Makefile instructions that build a target image, see “Working with QNX BSPs” in the *Building Embedded Systems* guide.

x86 vs. ARM platforms

Traditionally, when an x86 system boots, it provides some sort of first-stage OS loader. In older systems, this first-stage loader was the BIOS. Newer systems replace the x86 BIOS with a UEFI (Unified Extensible Firmware Interface), which provides a method for loading the OS image from the boot device and enhanced features that the system can use at runtime. In general, x86 systems allow you to load the QNX boot image using whatever boot method the hardware provides (that is, the same loader you would use to load an operating system such as Windows or Linux).

ARM-based platforms also traditionally provide a bootloader with the reference hardware. In newer systems, the U-Boot loader is the most common bootloader that ARM targets provide. You can use U-Boot to load a QNX OS image into the target's memory, the same as you would any other OS. In addition, QNX often provides a native bootloader, known as the Initial Program Loader (IPL). You normally use the IPL instead of U-Boot to accelerate the boot process or for boot operation configuration that cannot be performed using U-Boot. When you migrate an existing BSP to a new target board, QNX recommends that you use U-Boot as the bootloader for the new target board until you have completed porting all the BSP components.

QNX Neutrino uses an Initial Program Loader (IPL) to perform the following tasks:

- Initialize the hardware
- Set up a stack for the C-language environment
- Load the OS image into memory
- Initiate the QNX startup code

You can use U-Boot to replace the IPL functionality during the development process, or if your board does not support the QNX IPL (see "*Using U-Boot*").

The IPL has to be customized for the hardware to correctly configure the memory controller, set the clocks, and perform other hardware initializations.

Full information about the IPLs for each type of board, see Initial Program Loaders (IPLs)

in the *Building Embedded Systems* guide.

Using U-Boot

During the development process, it can be helpful to use U-Boot to manually boot the hardware and load the OS, even if your board supports the QNX IPL. For example, using U-Boot allows you to eliminate the IPL as a possible source of a booting problem.

In most cases, the final BSP uses an IPL because it is faster and is designed specifically to bring up with QNX OS.

CPU vs SoC

On older embedded systems, the CPU was usually a stand-alone device and all other functionality was implemented via external Integrated Circuits (ICs). For example, a board with a CPU, a network interface IC, a USB Host Controller IC, a serial UART, an external host/PCI bridge device, and so on. For this type of board architecture, it was often not feasible to try to adapt the QNX BSP for a reference design to a different board with the same CPU because the new board could have different external peripherals and require you to update or replace all of the device drivers for peripheral devices.

Newer embedded systems are increasingly using System on a Chip (SoC) processors, where the CPU core is coupled with a variety of different peripheral interfaces within the same IC package. With an SoC, when you port a BSP from a reference board to a new board using the same IC, you can re-use all of the device drivers in the BSP (or at least all the ones for peripherals contained within the SoC), with minimal changes. In this document, discussions about customizing BSPs assume that the target embedded system you are porting the BSP to uses the same SoC as the hardware that the reference BSP was developed for.

Minimal bootable system

Attempting to boot QNX Neutrino on your board with the bare minimum number of services allows you to isolate any problems. To create a minimal system, edit the OS image buildfile to exclude any services or other components that are not necessary for a baseline, functioning OS.

For example, you can exclude everything except the following components:

- Serial driver (to allow you to long in and observe the OS start up)
- The startup script
- slog2info (to allow you to get debugging information)
- Come commands such as ls, pwd, cd, cat (to allow you to view files and configuration, if you need to troubleshoot a service)

After this minimal system is running successfully, start to add the additional components you require (for example, USB drivers or networking).

The primary module you need to adapt to get a minimal QNX system working on a new target is the QNX startup module. This module takes over after the first-stage bootloader (BIOS, UEFI, U-boot, or IPL) has finished executing. The startup code executes after it has been copied from the boot device into the system's DRAM. The startup code sets up a number of in-memory data structures by populating them with information about the hardware, so that the QNX kernel and other device drivers can run. Collectively, these data structures are known as the QNX system page (syspage).

When you adapt startup code from an existing BSP to a new target board, ensure your investigation includes the following system elements:

DRAM memory size

The startup code usually has a routine that determines how much DRAM is installed in the system based on information that U-Boot or the IPL passes to it. However, in some cases, you need to manually specify how much DRAM is present in the system.

For example, the target system that the original BSP was written for has 2 gigabytes of DRAM, but your custom target has only 1 gigabyte of DRAM. To ensure that the startup code doesn't populate the system page with incorrect information, update it to specify the correct DRAM size. (If the QNX kernel thinks there is 2 gigabytes of DRAM present when there is actually only 1 gigabyte present, the kernel crashes as soon as you try to boot the system.)

CPU Frequency and clocks

As with DRAM memory size, it is important to go through the startup code to ensure that the clock initialization is correct for your target board, if the CPU core frequency or any of the peripheral clocks differ from the reference BSP.

Pinmux and GPIO configuration

Although a variety of common peripherals can be built into the SoC, the method that internally connects these peripheral blocks to the CPU core can sometimes be configurable by the software.

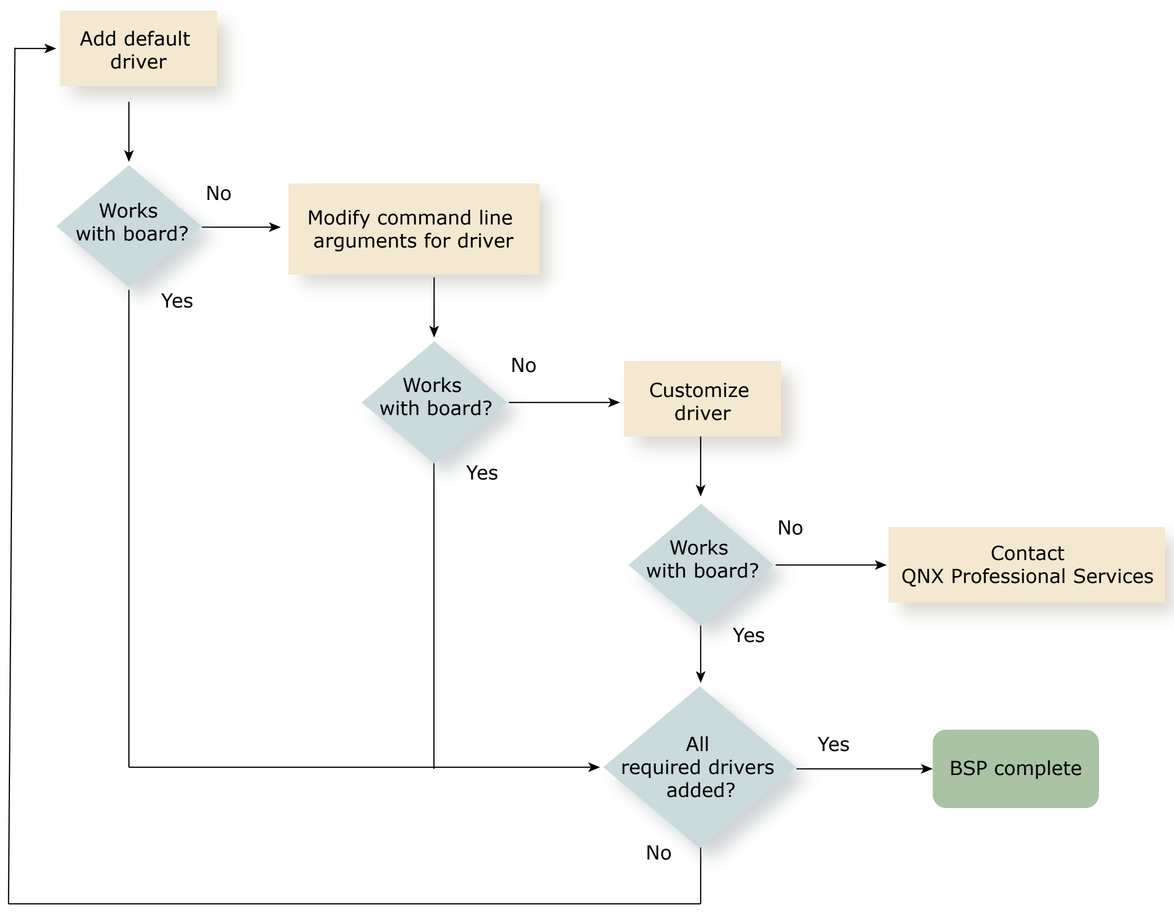
For example, a peripheral such as a UART interface can have the option of using two or three different sets of multiplexed pins that transmit UARTsignals from the SoC to external devices. Unless the custom board you are porting to uses exactly the same pin configuration as the reference board, you need to modify the startup code's *init_pinmux()* routine to ensure that each peripheral's signals are routed to the correct sets of external pins on the SoC.

Similarly, some peripherals utilize GPIO (general purpose input/output) pins on the SoC for particular functionality. To accommodate any differences in GPIO pin utilization between the reference hardware and the custom hardware, ensure that you examine the portion of the startup code that initializes the GPIO pins and modify it if necessary.

For information about building an OS image, see OS Images and OS Image Buildfiles in *Building Embedded Systems*.

Drivers

Each QNX BSP provides a variety of device drivers for the various peripherals found on the target hardware that the BSP is intended for. In general, you can re-use all device drivers for peripherals that are internal to the SoC without any modification if that the pinmux and GPIO configuration for those peripherals is properly addressed in the startup code.



In many cases, only minor modifications to the device driver's command line are necessary to enable a driver from a reference BSP to operate properly on the new target board. You can specify all parameters such as IRQ, I/O port, clock speed, and so on, on the command line. Use the command line examples from the reference BSP to determine what changes (if any) you need to make for your custom hardware.

For more information about drivers and instructions for writing one and adding it to your BSP, see *Writing a Resource Manager*.

Chapter 3
OS Image Buildfiles

The BSPs that QNX provides often come with more than one buildfile. The buildfile to select as the source for building your custom BSP depends on the features you want to support. For example, in some cases, BSPs that can support the Screen Graphics Subsystem provide a specific buildfile for systems that provide graphics functionality.

The buildfile for your BSP is located in the directory **${BSP_ROOT_DIR}/prebuilt/${PROCESSOR}/boot/build/*board*.build**. For example: **/prebuilt/aarch64le/boot/build/dekelsinore-ghost8.build**

For full information on modifying buildfiles, see OS Image Buildfiles in the *Building Embedded Systems* guide.

Chapter 4
Startup

Unlike many other operating systems that perform most initialization tasks after the OS is running, QNX Neutrino performs many key initializations using its startup program.

These initializations include timers, interrupt controllers, and cache controllers. Custom board projects can require modifications to the following board and startup code elements:

- Memory mapping
- Voltages
- Kernel callouts

For a discussion of modifying or creating the startup code, see “Modifying the startup” in the Startup Programs chapter of *Building Embedded Systems*.

Memory mapping

Your custom board’s hardware determines the memory configuration options that are available to you.

Some boards expect the startup code to load the OS at a specific address (for example, SABRE platforms).

Not all boards support compression, which prevents you from compressing the IFS to improve performance or conserve memory.

The type of removable media used to store the IPL and the IFS affects the IPL design. The most common type of media used are non-linearly-mapped devices (for example, an eMMC, SD card, or SPI NOR Flash device). The storage provided by these devices can't be mapped directly into the processor's address space. Because the processor can't address the OS image, the IPL needs to copy the entire image (including the startup code) into RAM. For more information, see “Image storage” in the Initial Program Loaders (IPLs) chapter of *Building Embedded Systems*.

Voltages

A system that uses the QNX Graphics Framework (GF) needs to power up the GPU before it initializes the graphics functionality. Refer to the hardware specifications for your board to determine whether your GPU requires the startup code to perform this task. For example, some boards enable the GPU by default and others require you to set the voltage and enable it.

Depending on your system’s requirements, you can configure the startup code to turn on only some graphics resources (for example, the display controller only). Other graphics resources can then be turned on after startup when they are needed (for example, using a driver).

Callouts

The kernel uses callouts to provide the QNX Neutrino kernel and process manager with hardware-specific code. If your hardware platform isn't supported by the callouts in the startup library, or by any of the available board-specific callouts, you write your own.

For information about the callouts and how to write a kernel callout, see Kernel Callouts in *Building Embedded Systems*.

Chapter 5 Graphics

A board’s GPU can provide the QNX Neutrino with the following graphics resources:

- Display controller
- 2D and 3D accelerators
- video capture interface (for example, CSI)

For a discussion of requirements and options for powering up the GPU, see “*Voltages*”.

Display controller and display resolution

The default BSP for a reference board includes a display driver (WFD driver) that works with that board’s display controller. If your custom board implements the display controller using a bridge chip (sometimes called a transceiver) that is different than the one the reference board uses, you can implement the Wfdcfg library to set up your display. The Wfdcfg library provides the display driver with the modes and attributes of the display hardware.

For information on using the `graphics.conf` configuration file to configure the display controller, see “Configure `wfd device` subsection” in the “Configuring Screen” chapter of *Screen Graphics Subsystem Developer's Guide*.

For information on using the `graphics.conf` configuration file to configure the physical display that your board uses, see “Configure `display` subsection” in the “Configuring Screen” chapter of *Screen Graphics Subsystem Developer's Guide*.

If your display controller requires a custom driver, contact your hardware vendor or QNX Professional Services for assistance.

2D and 3D accelerators

If the BSP provided with the reference board does not work with your custom board’s 2D or 3D accelerators, contact your hardware vendor or QNX Professional Services for assistance.

Video capture interface

For reference boards that support a video capture interface, the default BSP provides a driver that supports it. This driver does not support every possible decoder the video capture interface can use.

If your custom board implements the video capture interface using a decoder that is different than the one the reference board uses, you can modify the video capture framework libraries to provide support for it. The framework includes a `libcapture-board-*-*.so` library that support's the board's SoC and decoder chip and a `libcapture-decoder-*.so` library that initializes and applies properties to decoders.

For information about the video capture framework and the tasks involved in implementing video capture, see *Video Capture Developer's Guide*.

Graphic settings discovery tools

The generic x86 and x86_64 APIC and UEFI BSPs provide a utility called `drm-probe-displays` that you can run on the target to determine the correct settings for the `graphics.conf` file. For more information about this utility, see `drm-probe-displays` in the “Using the Screen Graphics Subsystem” chapter of the *Generic x86 and x86_64 BSP User's Guide*.

For other board platforms, to determine the appropriate graphics.conf settings, see the board’s hardware specification.

Chapter 6
I2C

I2C (Inter-Integrated Circuit) is a simple serial protocol that connects multiple devices in a master-slave relationship.

In most cases, the BSP for your reference board includes the source for an I2C driver that you can try to use with your custom board.

You can use the I2C Framework to modify the I2C driver, if required.

I2C (Inter-Integrated Circuit) Framework

I2C (Inter-Integrated Circuit) is a simple serial protocol that connects multiple devices in a master-slave relationship. Multiple master devices may share a single bus. The same device may function as both a master and a slave in different transactions. The I2C specification defines these transfer speed ranges:

- 100 Kbit/s
- 400 Kbit/s
- 3.4 Mbit/s

The I2C framework is intended to facilitate consistent implementation of I2C interfaces. The framework consists of the following parts:

hardware/i2c/*

The hardware interface.

lib/i2c

The resource manager layer.

<hw/i2c.h>

A public header file that defines the hardware and application interfaces.

The most common application of the I2C bus is low-bandwidth access to a slave device's registers, such as:

- programming an audio codec
- programming a RTC
- reading a temperature sensor
- reading from an EPROM

Typically, only a few bytes are exchanged on the bus.

You can implement the I2C master as a single-threaded resource manager, or as a dedicated application. The primary advantages of a resource manager interface are:

- It presents a clear, easy-to-understand interface to the application developer.
- It mediates between accesses by multiple applications to one or more slave devices.
- It enforces consistency between different I2C interfaces.

For a dedicated I2C-bus application, a hardware access library is more efficient. The hardware interface, which defines the interface to this library, is useful as a starting point for developers and facilitates maintenance and code portability.

Hardware interface

This is the interface to the code that implements the hardware-specific functionality of an I2C master. It's defined in **<hw/i2c.h>**.

Function table

The `i2c_master_funcs_t` structure is a table of pointers to functions that you can provide for your hardware. The higher-level code calls these functions.

```
typedef struct {
    size_t size;    /* size of this structure */
    int (*version_info)(i2c_libversion_t *version);
    void *(*init)(int argc, char *argv[]);
    void (*fini)(void *hdl);
    i2c_status_t (*send)(void *hdl, void *buf, unsigned int len,
                         unsigned int stop);
    i2c_status_t (*recv)(void *hdl, void *buf, unsigned int len,
                         unsigned int stop);
    int (*abort)(void *hdl, int rcvid);
    int (*set_slave_addr)(void *hdl, unsigned int addr, i2c_addrfmt_t fmt);
    int (*set_bus_speed)(void *hdl, unsigned int speed, unsigned int *ospeed);
    int (*driver_info)(void *hdl, i2c_driver_info_t *info);
    int (*ctl)(void *hdl, int cmd, void *msg, int msglen,
               int *nbytes, int *info);
} i2c_master_funcs_t;
```

- In a multimaster system, the *send* and *recv* functions should handle bus arbitration.
- If the I2C interface supports DMA, you can implement it in the *send* and *recv* functions.

version_info function

The higher-level code calls the *version_info* function to get information about the version of the library. The prototype for this function is:

```
int version_info( i2c_libversion_t *version );
```

The *version* argument is a pointer to a `i2c_libversion_t` structure that this function must fill in. This structure is defined as follows:

```
typedef struct {
    unsigned char   major;
    unsigned char   minor;
    unsigned char   revision;
} i2c_libversion_t;
```

This function should set the members of this structure as follows:

```
version->major = I2CLIB_VERSION_MAJOR;
version->minor = I2CLIB_VERSION_MINOR;
version->revision = I2CLIB_REVISION;
```

The function must return:

0

Success.

-1

Failure.

init function

The *init* function initializes the master interface. The prototype for this function is:

```
void *init( int argc, char *argv[]);
```

The arguments are those passed on the command line.

The function returns a handle that's passed to all other functions, or NULL if an error occurred.

fini function

The *fini* function cleans up the driver and frees any memory associated with the given handle. The prototype for this function is:

```
void fini( void *hdl);
```

The argument is the handle that the *init* function returned.

send function

The *send* function initiates a master send. An optional event is sent when the transaction is complete (and the data buffer can be released). If this function fails, no transaction has been initiated.

The prototype for this function is:

```
i2c_status_t send(
               void *hdl,
               void *buf,
               unsigned int len,
               unsigned int stop );
```

The arguments are:

hdl

The handle returned by the *init* function.

buf

A pointer to the buffer of data to send.

len

The length, in bytes, of the data to send.

stop

If this is nonzero, the function sets the stop condition when the send completes.

The function returns one of the following:

I2C_STATUS_DONE

The transaction completed (with or without an error).

I2C_STATUS_ERROR

An unknown error occurred.

I2C_STATUS_NACK

Slave no-acknowledgement.

I2C_STATUS_ARBL

Lost arbitration.

I2C_STATUS_BUSY

The transaction timed out.

I2C_STATUS_ABORT

The transaction was aborted.

recv function

The *recv* function initiates a master receive. An optional event is sent when the transaction is complete (and the data buffer can be used). If this function fails, no transaction has been initiated.

The prototype for this function is:

```
i2c_status_t recv(
               void *hdl,
               void *buf,
               unsigned int len,
               unsigned int stop );
```

The arguments are:

hdl

The handle returned by the *init* function.

buf

A pointer to the buffer in which to put the received data.

len

The length, in bytes, of the buffer.

stop

If this is nonzero, the function sets the stop condition when the receive completes.

The function returns one of the following:

I2C_STATUS_DONE

The transaction completed (with or without an error).

I2C_STATUS_ERROR

An unknown error occurred.

I2C_STATUS_NACK

Slave no-acknowledgement.

I2C_STATUS_ARBL

Lost arbitration.

I2C_STATUS_BUSY

The transaction timed out.

I2C_STATUS_ABORT

The transaction was aborted.

abort function

The *abort* function forces the master to free the bus. It returns when the stop condition has been sent. The prototype for this function is:

```
int abort (
       void *hdl,
       int rcvid );
```

The arguments are:

hdl

The handle returned by the *init* function.

rcvid

The receive ID of the client.

The function must return:

0

Success.

-1

Failure.

set_slave_addr function

The *set_slave_addr* function specifies the target slave address. The prototype for this function is:

```
int set_slave_addr (
       void *hdl,
       unsigned int addr,
       i2c_addrfmt_t fmt );
```

The arguments are:

hdl

The handle returned by the *init* function.

addr

The target slave address.

fmt

The format of the address; one of:

- I2C_ADDRFMT_7BIT
- I2C_ADDRFMT_10BIT

The function must return:

0

Success.

-1

Failure.

set_bus_speed function

The *set_bus_speed* function specifies the bus speed. If an invalid bus speed is requested, this function should return a failure and leave the bus speed unchanged. The prototype for this function is:

```
int set_bus_speed(
        void *hdl,
        unsigned int speed,
        unsigned int *ospeed );
```

The arguments are:

hdl

The handle returned by the *init* function.

speed

The bus speed. The units are implementation-defined.

ospeed

NULL, or a pointer to a location where the function should store the actual bus speed.

The function must return:

0

Success.

-1

Failure.

driver_info function

The *driver_info* function returns information about the driver. The prototype for this function is:

```
int driver_info(
       void *hdl,
       i2c_driver_info_t *info );
```

The arguments are:

hdl

The handle returned by the *init* function.

info

A pointer to a `i2c_driver_info_t` structure where the function should store the information:

```
typedef struct {
    uint32_t    speed_mode;
    uint32_t    addr_mode;
    uint32_t    reserved[2];
} i2c_driver_info_t;
```

For the *speed_mode* member, OR together the appropriate values from the following list to indicate the supported speeds:

I2C_SPEED_STANDARD

Up to 100 Kbit/s.

I2C_SPEED_FAST

Up to 400 Kbit/s.

I2C_SPEED_HIGH

Up to 3.4 Mbit/s.

Set the *addr_mode* to one of the following to indicate the supported address format:

- I2C_ADDRFMT_7BIT
- I2C_ADDRFMT_10BIT

The function must return:

0

Success.

-1

Failure.

ctl function

The *ctl* function handles a driver-specific *devctl()* command. The prototype for this function is:

```
int ctl(
      void *hdl,
      int cmd,
      void *msg,
      int msglen,
      int *nbytes,
      int *info );
```

The arguments are:

hdl

The handle returned by the *init* function.

cmd

The device command.

msg

A pointer to the message buffer. The function can change the contents of the buffer.

msglen

The length of the message buffer, in bytes.

nbytes

The number of bytes being returned. This must not be greater than *msglen*.

info

A pointer to a location where the function can store extra status information returned by *devctl()*.

The function must return:

EOK

Success.

Any other *errno* value

Failure.

Access function

This function is used by higher-level code (such as the resource manager interface) to access the hardware-specific functions. It must be implemented.

```
int i2c_master_getfuncs(i2c_master_funcs_t *funcs, int tabsize);
```

This function must fill in the given table with the hardware-specific functions.

The arguments are:

funcs

The function table to fill in. The library initializes this table before calling this function. If you haven't implemented a function in the table, leave its entry unchanged; don't set it to NULL.

tabsize

The size of the structure that *funcs* points to, in bytes.

Don't change the *size* member of the structure.

To set an entry in the table, use the *I2C_ADD_FUNC()* macro:

```
#define I2C_ADD_FUNC(tabletype, table, entry, func, tabsize) …
```

For example:

```
I2C_ADD_FUNC( i2c_master_funcs_t, funcs, init, my_init, tabsize);
```

The function must return:

0

Success.

-1

Failure.

Sample calls

A typical sequence of hardware library calls is as follows:

```
#include <hw/i2c.h>

i2c_master_funcs_t  masterf;
i2c_libversion_t    version;
i2c_status_t        status;
void *hdl;

i2c_master_getfuncs(&masterf, sizeof(masterf));

masterf.version_info(&version);
if ((version.major != I2CLIB_VERSION_MAJOR) ||
    (version.minor > I2CLIB_VERSION_MINOR))
{
    /* error */
    ...
}

hdl = masterf.init(...);
```

```
masterf.set_bus_speed(hdl, ...);
masterf.set_slave_addr(hdl, ...);

status = masterf.send(hdl, ...);
if (status != I2C_STATUS_DONE) {
    /* error */
    if (!(status & I2C_STATUS_DONE))
        masterf.abort(hdl);
}

status = masterf.recv(hdl, ...);
if (status != I2C_STATUS_DONE) {
    /* error */
    ...
}

masterf.fini(hdl);
```

Application interfaces

Shared-library interface

When an I2C master device is dedicated for use by a single application, you can compile the hardware interface code as a library and link it to the application.

To increase code portability, the application should call *i2c_master_getfuncs()* to access the hardware-specific functions. Accessing the hardware library through the function table also makes it easier for an application to load and manage more than one hardware library.

The resource manager interface uses the hardware library in a similar way.

Resource manager interface

The resource manager interface is designed to mediate accesses to a single master. In a system with multiple masters, a separate instance of the resource manager should be run for each master.

The resource manager layer registers a device name (usually **/dev/i2c0**). Applications access the I2C master by issuing *devctl()* commands to the device name.

The supported *devctl()* commands and their formats are defined in **<hw/i2c.h>**.

Supporting data types

Many of the *devctl()* commands use the `i2c_addr_t` structure, which is defined as:

```
typedef struct {
    uint32_t addr;    /* I2C address */
    uint32_t fmt;     /* I2C_ADDRFMT_7BIT or I2C_ADDRFMT_10BIT */
} i2c_addr_t;
```

DCMD_I2C_BUS_RESET

```
#include <hw/i2c.h>

#define DCMD_I2C_BUS_RESET          __DION (_DCMD_I2C, 12)
```

The arguments to *devctl()* are:

Argument	Value
filedes	A file descriptor that you obtained by opening the device.
dcmd	DCMD_I2C_BUS_RESET
dev_data_ptr	NULL
n_bytes	0
dev_info_ptr	NULL

The DCMD_I2C_BUS_RESET command resets the bus.

Input

None.

Output

None.

If an error occurs, the command returns:

EIO

The driver either doesn't support resetting the bus, or it couldn't reset the controller.

DCMD_I2C_DRIVER_INFO

```
#include <hw/i2c.h>

#define DCMD_I2C_DRIVER_INFO         __DIOF (_DCMD_I2C, 10, i2c_driver_info_t)
```

The arguments to *devctl()* are:

Argument	Value
filedes	A file descriptor that you obtained by opening the device.
dcmd	DCMD_I2C_DRIVER_INFO
dev_data_ptr	A pointer to a `i2c_driver_info_t`
n_bytes	`sizeof(i2c_driver_info_t)`
dev_info_ptr	NULL

The DCMD_I2C_DRIVER_INFO command returns information about the hardware library.

Input

None.

Output

An `i2c_driver_info_t` structure that contains driver information; see the description of the *driver_info function*, earlier in this technote.

If an error occurs, the command returns:

EIO

The driver query failed.

DCMD_I2C_MASTER_RECV (deprecated)

The DCMD_I2C_MASTER_RECV command executes a master receive transaction, using the slave device address and bus speed specified for the current connection.

Input

`i2c_messagehdr_t` — the message header

Output

`uint8_t[]` — the receive data

The message header is overwritten.

If an error occurs, the command returns:

EIO

The master receive failed. Causes include: bad slave address, bad bus speed, bus is busy.

EINVAL

Bad message format.

ENOMEM

Insufficient memory.

EPERM

The master is locked by another connection.

DCMD_I2C_MASTER_SEND (deprecated)

The DCMD_I2C_MASTER_SEND command execute a master send transaction, using the slave device address and bus speed specified for the current connection.

Input

- `i2c_masterhdr_t` — the message header:

```
typedef struct {
    uint32_t len;     /* length of data to send/recv, in bytes
                         (not including this header) */
    uint32_t stop;    /* send stop when complete? (0=no, 1=yes) */
} i2c_masterhdr_t;
```

- `uint8_t[]` — the data buffer

Output

None.

If an error occurs, the command returns:

EIO

The master send failed. Causes include: bad slave address, bad bus speed, bus is busy.

EFAULT

An error occurred while accessing the data buffer.

EINVAL

Bad message format.

ENOMEM

Insufficient memory.

EPERM

The master is locked by another connection.

DCMD_I2C_RECV

```
#include <hw/i2c.h>

#define DCMD_I2C_RECV                   __DIOTF(_DCMD_I2C, 6, i2c_recv_t)
```

The arguments to *devctl()* are:

Argument	Value
filedes	A file descriptor that you obtained by opening the device.
dcmd	DCMD_I2C_RECV
dev_data_ptr	A pointer to a `i2c_recv_t`, followed by the receive buffer
n_bytes	`sizeof(i2c_recv_t)` plus the size of the receive buffer
dev_info_ptr	NULL

The DCMD_I2C_RECV command executes a master receive transaction. It returns when the transaction is complete.

Input

- `i2c_recv_t` — the message header
- `uint8_t[]` — the receive buffer

Output

- `i2c_recv_t` — the message header (unchanged)
- `uint8_t[]` — the receive data in the buffer

The `i2c_recv_t` structure is defined as:

```
typedef struct {
    i2c_addr_t slave;   /* slave address */
    uint32_t   len;     /* length of receive data in bytes */
    uint32_t   stop;    /* send stop when complete? (0=no, 1=yes) */
} i2c_recv_t;
```

If an error occurs, the command returns:

EIO

The master send failed. Causes include: bad slave address, bad bus speed, bus is busy.

EINVAL

Bad message format.

ENOMEM

Insufficient memory.

EPERM

The master is locked by another connection.

DCMD_I2C_SEND

```
#include <hw/i2c.h>

#define DCMD_I2C_SEND                 __DIOT (_DCMD_I2C, 5, i2c_send_t)
```

The arguments to *devctl()* are:

Argument	Value
filedes	A file descriptor that you obtained by opening the device.
dcmd	DCMD_I2C_SEND
dev_data_ptr	A pointer to a `i2c_send_t` that's followed by additional data
n_bytes	`sizeof(i2c_send_t)` plus the size of the additional data
dev_info_ptr	NULL

The DCMD_I2C_SEND command executes a master send transaction. It returns when the transaction is complete.

Input

- `i2c_send_t` — the message header:

```
typedef struct {
    i2c_addr_t slave;  /* slave address */
    uint32_t   len;    /* length of send data in bytes */
    uint32_t   stop;   /* send stop when complete? (0=no, 1=yes) */
} i2c_send_t;
```

- `uint8_t[]` — the data buffer

Output

None.

If an error occurs, the command returns:

EIO

The master send failed. The causes include: bad slave address, bad bus speed, bus is busy.

EFAULT

An error occurred while accessing the data buffer.

EINVAL

Bad message format.

ENOMEM

Insufficient memory.

EPERM

The master is locked by another connection.

DCMD_I2C_SENDRECV

```
#include <hw/i2c.h>

#define DCMD_I2C_SENDRECV           __DIOTF(_DCMD_I2C, 7, i2c_sendrecv_t)
```

The arguments to *devctl()* are:

Argument	Value
filedes	A file descriptor that you obtained by opening the device.
dcmd	DCMD_I2C_SENDRECV
dev_data_ptr	A pointer to a `i2c_sendrecv_t` that's followed by a buffer
n_bytes	`sizeof(i2c_sendrecv_t)` plus the size of the buffer
dev_info_ptr	NULL

The DCMD_I2C_SENDRECV command executes a send followed by a receive. This sequence is typically used to read a slave device's register value. When multiple applications access the same slave device, it is necessary to execute this sequence atomically to prevent register reads from being interrupted.

Input

- `i2c_sendrecv_t` — the message header
- `uint8_t[]` — a buffer, containing the send data, that's large enough to hold the receive data

Output

- `i2c_sendrecv_t` — the message header (unchanged)
- `uint8_t[]` — the receive data in the buffer

The `i2c_sendrecv_t` structure is defined as:

```
typedef struct {
    i2c_addr_t slave;      /* slave address */
    uint32_t   send_len;   /* length of send data in bytes */
    uint32_t   recv_len;   /* length of receive data in bytes */
    uint32_t   stop;       /* set stop when complete? */
} i2c_sendrecv_t;
```

If an error occurs, the command returns:

EIO

The master send failed. Causes include: bad slave address, bad bus speed, bus is busy.

EFAULT

An error occurred while accessing the data buffer.

EINVAL

Bad message format.

ENOMEM

Insufficient memory.

EPERM

The master is locked by another connection.

DCMD_I2C_SET_BUS_SPEED

```
#include <hw/i2c.h>

#define DCMD_I2C_SET_BUS_SPEED      __DIOT (_DCMD_I2C, 2, uint32_t)
```

The arguments to *devctl()* are:

Argument	Value
filedes	A file descriptor that you obtained by opening the device.
dcmd	DCMD_I2C_SET_BUS_SPEED
dev_data_ptr	A pointer to a `uint32_t`
n_bytes	`sizeof(uint32_t)`
dev_info_ptr	NULL

The DCMD_I2C_SET_BUS_SPEED command sets the bus speed for the current connection. You should set the bus speed before attempting a data-transfer operation.

Input

`uint32_t` — the bus speed.

Output

None.

If an error occurs, the command returns:

EINVAL

Bad message format.

DCMD_I2C_SET_SLAVE_ADDR (deprecated)

The DCMD_I2C_SET_SLAVE_ADDR command sets the slave device address for the current connection. You should set the slave device address before attempting a master send or receive transaction.

Input

`i2c_addr_t` — the slave device address and the address format

Output

None.

This command doesn't return any error codes.

DCMD_I2C_STATUS

```
#include <hw/i2c.h>

#define DCMD_I2C_STATUS             __DIOF (_DCMD_I2C, 11, i2c_status_t)
```

The arguments to *devctl()* are:

Argument	Value
filedes	A file descriptor that you obtained by opening the device.
dcmd	DCMD_I2C_STATUS
dev_data_ptr	A pointer to an `i2c_status_t`
n_bytes	sizeof(i2c_status_t)
dev_info_ptr	NULL

The DCMD_I2C_STATUS command gets the status of the device.

Input

None.

Output

The status; one of:

I2C_STATUS_DONE

The transaction completed (with or without an error).

I2C_STATUS_ERROR

An unknown error occurred.

I2C_STATUS_NACK

Slave no-acknowledgement.

I2C_STATUS_ARBL

Lost arbitration.

I2C_STATUS_BUSY

The transaction timed out.

I2C_STATUS_ABORT

The transaction was aborted.

Resource manager design

The resource manager layer is implemented as a library that's statically linked with the hardware library. We currently provide a single-threaded manager, **libi2c-master**.

On startup, the `resmgr` layer does the following:

```
i2c_master_getfuncs(&masterf)
masterf.init()
masterf.set_bus_speed()
```

The resource manager then makes itself run in the background.

Here's how the resource manager handles these *devctl()* commands:

- DCMD_I2C_DRIVER_INFO calls *masterf.driver_info()*.
- DCMD_I2C_SET_BUS_SPEED and DCMD_I2C_SET_SLAVE_ADDR only update the state of the current connection.
- DCMD_I2C_SEND, DCMD_I2C_RECV, DCMD_I2C_SENDRECV, DCMD_I2C_MASTER_SEND, and DCMD_I2C_MASTER_RECV result in the following:

  ```
  if (bus_speed has changed)
      masterf.set_bus_speed()
  masterf.set_slave_address()
  masterf.send() or masterf.recv()
  ```

The resource manager thread remains occupied until the transaction is complete and replies to the client.

You can terminate the resource manager by sending a SIGTERM to it.

Chapter 7
Networking

If the networking hardware on your custom board is different than what the reference board uses, you may have to modify the network driver provided with the BSP to enable networking.

Ethernet

For example, the BSP network driver works with the SoC's Ethernet controller found on the reference and custom boards, but the custom board has different hardware for the physical layer (PHY chip) that provides the connection to the Ethernet.

Most of the QNX network drivers provide all the PHY-related functions using the file **mii.c**. For example, the BSP for an i.MX6 SoloX includes the following file:

${QNX_BSP_ROOT_DIR}\src\hardware\devnp\mx6x\mii.c

Where **${QNX_BSP_ROOT_DIR}** is the name of the directory that you extracted the BSP archive to.

The network driver links to a specialized driver library (`libnetdrvr.a`), which contains all standard PHY functions, including those for initialization tasks.

If the PHY on your custom board requires different initialization routines, add the following to the **mii.c** file:

- The specific initialization routines
- A check for the specific PHY, to allow its initialization routines to be performed

You can use the macros `PHYID_VENDOR()` and `PHYID_MODEL()` included in the **mdi.h** file to determine the PHY vendor and model.

For information on the BSP file structure, see "BSP structure and contents" in the Working with QNX BSPs chapter of *Building Embedded Systems*.

Wireless

If you require a custom wireless network driver, contact QNX Professional Services.

`qconn`, Telnet, SSH

In many cases, communication with your board via `qconn`, Telnet, or SSH is helpful during development and testing, but is not required in production versions. To improve board security, ensure that these protocols are disabled in your final BSP.

Chapter 8
Generic serial port

The BSP that QNX provides for each reference board contains a driver for a generic serial port. If this driver does not work with your custom board, you can customize it.

For information on the generic serial port drivers, see the devc-* entries in *Utilities Reference*. For example, see `devc-serpci`.

Chapter 9
SPI

The BSP that QNX provides for each reference board contains an SPI driver. If this driver does not work with your custom board, you can customize it.

For information about the SPI framework that implements the driver, see SPI (Serial Peripheral Interface) Framework in the QNX Neutrino technotes.

Index