QNX® Momentics® DDK

Input Devices

For targets running QNX® Neutrino® 6.3.0 or later

Printed under license by:

QNX Software Systems Co.
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: **info@qnx.com**
Web: **http://www.qnx.com/**

Publishing history

Electronic edition published 2005.

Technical support options

To obtain technical support for any QNX product, visit the **Technical Support** section in the **Services** area on our website (**www.qnx.com**). You'll find a wide range of support options, including our free web-based **Developer Support Center**.

Contents

About the Input DDK

The following table may help you find information quickly:

If you want to:	Go to:
Get an overview of input modules and how they're linked	Overview
Use a non-Photon interface to the system	Overview
Understand the source file organization for **`devi-*`**	Overview
Begin writing your own input driver	Writing an Input Device Driver
Learn about the data formats of protocol modules	Writing an Input Device Driver
Write a driver for a keyboard device	Writing an Input Device Driver
Write a driver for a touchscreen	Writing an Input Device Driver
Write a driver for a mouse	Writing an Input Device Driver
Combine *device* and *protocol* functionality in a single driver	Writing an Input Device Driver
Debug your driver	Testing and Debugging Your Driver
Look up a module function	Module Functions
Look up an interface function in the Input API	API Reference

Building DDKs

You can compile the DDK from the IDE or the command line.

- To compile the DDK from the IDE:

Please refer to the Managing Source Code chapter, and "QNX Source Package" in the Common Wizards Reference chapter of the *IDE User's Guide*.

- To compile the DDK from the command line:

  Please refer to the release notes or the installation notes for information on the location of the DDK archives.

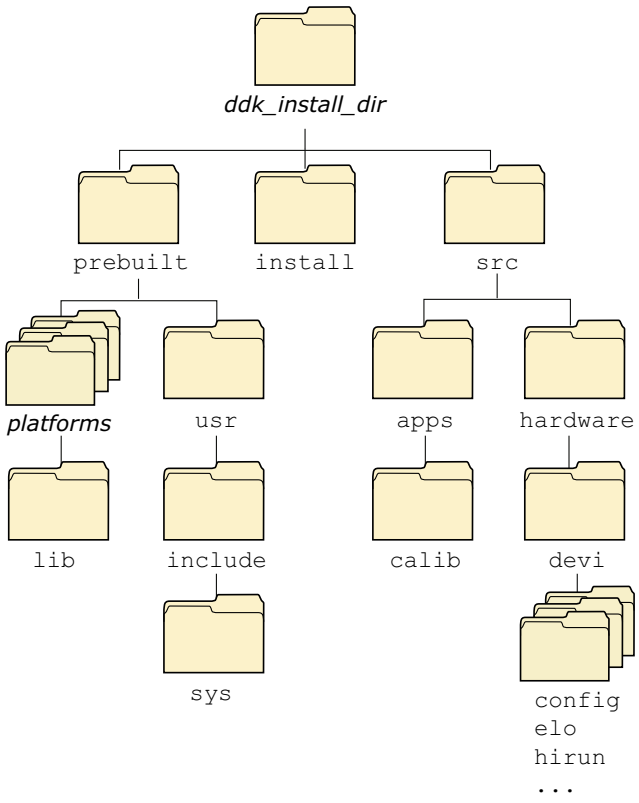
  DDKs are simple zipped archives, with no special requirements. You must manually expand their directory structure from the archive. You can install them into whichever directory you choose, assuming you have write permissions for the chosen directory.

  Historically, DDKs were placed in **`/usr/src/ddk_VERSION`** directory, e.g. **`/usr/src/ddk-6.2.1.`** This method is no longer required, as each DDK archive is completely self-contained.

  The following example indicates how you create a directory and unzip the archive file:

```
# cd ~
# mkdir my_DDK
# cd my_DDK
# unzip /path_to_ddks/ddk-device_type.zip
```

  The top-level directory structure for the DDK looks like this:



Directory structure for this DDK.

You must run:

```
. ./setenv.sh
```

before running **make**, or **make install**.

Additionally, on Windows hosts you'll need to run the **Bash** shell (**bash.exe**) before you run the **. ./setenv.sh** command.

If you fail to run the **. ./setenv.sh** shell script prior to building the DDK, you can overwrite existing binaries or libs that are installed in **$QNX_TARGET**.

Each time you start a new shell, run the **. ./setenv.sh** command. The shell needs to be initialized before you can compile the archive.

The script will be located in the same directory where you unzipped the archive file. It must be run in such a way that it modifies the current shell's environment, not a sub-shell environment.

In **ksh** and **bash** shells, All shell scripts are executed in a sub-shell by default. Therefore, it's important that you use the syntax

```
. <script>
```

which will prevent a sub-shell from being used.

Each DDK is rooted in whatever directory you copy it to. If you type **make** within this directory, you'll generate all of the buildable entities within that DDK no matter where you move the directory.

all binaries are placed in a scratch area within the DDK directory that mimics the layout of a target system.

When you build a DDK, everything it needs, aside from standard system headers, is pulled in from within its own directory. Nothing that's built is installed outside of the DDK's directory. The makefiles shipped with the DDKs copy the contents of the **prebuilt** directory into the **install** directory. The binaries are built from the source using include files and link libraries in the **install** directory.

Chapter 1

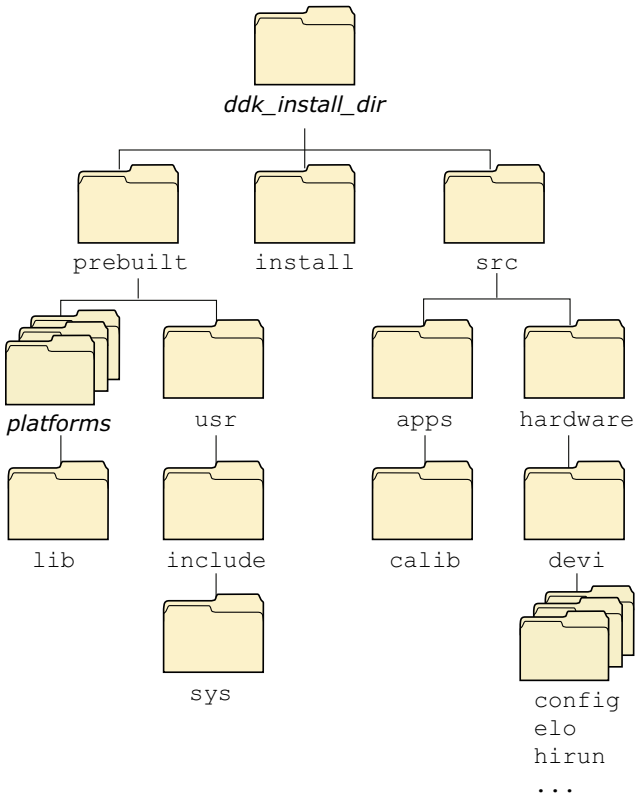
Overview

In this chapter. . .

This chapter provides an overview of writing input device drivers for QNX Neutrino. Use this along with the code in the **`sample`** and **`hirun`** directories (under *ddk_install_dir*`/ddk-input/src/hardware/devi`).

DDK source code

When you install the DDK package, the source is put into a directory under the *ddk_install_dir*`/ddk-input` directory. Currently, the directory structure for the Input DDK looks like this:



Directory structure for the Input DDK.

Inside an input driver

The input driver consists of two main components:

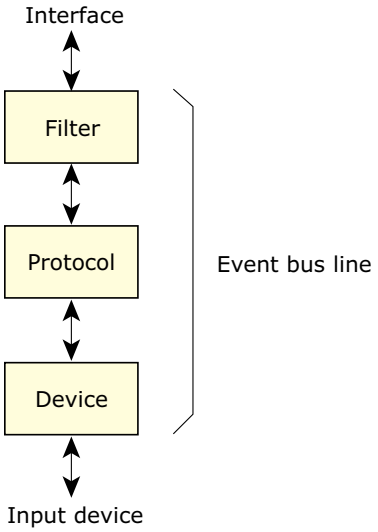
- a group of input modules
- a library used in manipulating these modules.

At run time, modules are linked together to form a data path for gathering data from an input device, processing the data, and then outputting the data to the system.

There are three types of modules:

- device modules
- protocol modules
- filter modules.

The modules are typically organized as follows:



The input chain.

When modules are linked together, they form an “event bus line.” Data passes from an input device up the event bus line and out to the system. Conversely, configuration control flows the other way (i.e. down the line to the device).

Types of event bus lines

There are three different types of event bus lines:

- relative
- absolute
- keyboard

The term “relative” simply means that the device provides position data that’s relative to the last location it reported. This is typically the method that mice use.

An “absolute” bus line is used with devices (e.g. touchscreens) that provide position data at absolute coordinates.

Finally, a “keyboard” type of bus line is one in which some sort of keypad device provides codes for every key press and release.

How modules are linked

As mentioned earlier, there are three types of modules:

Device-layer module
: Responsible for communicating with a hardware or software device. It typically has no knowledge of the format of the data from the device; it’s responsible only for getting data.

Protocol-layer module
: Interprets the data it gets from a device module according to a specific protocol.

Filter module
: Provides any further data manipulation common to a specific class of event bus.

Modules are linked together according to the command-line parameters passed into the input driver. The command line has the following format:

devi-*driver_name* **[***options***] protocol [***protocol_options***]**
[device [*device_options***]] [filter [***filter_options***]]**

In this example:

```
devi-hirun ps2 kb -2 &
```

the elements are as follows:

hirun — the "high-runner" input driver, which contains mouse and keyboard drivers used in most desktop systems.

ps2 — specifies the PS/2 mouse protocol, a three-byte protocol indicating mouse movement and button states.

kb — specifies the **kb** device module, which can communicate with a standard PC 8042-type keyboard controller.

-2 — specifies an option to the **kb** module, telling it to set up communication to its second (or auxiliary) port, which is for a PS/2 mouse.

You don't need to specify a filter module, because the three classes of event bus lines are represented by three modules, called **rel**, **abs**, and **keyboard**. When the input driver parses the command line, it can tell from the **ps2** module that it needs to link in the **rel** filter-module. The only time you would explicitly specify a filter module on the command line is if you need to pass it optional command-line parameters. For example:

```
devi-hirun ps2 kb -2 rel -G2
```

This tells the relative filter module to multiply *X* and *Y* coordinates passed in by 2, effectively providing a gain factor (a faster-moving mouse).

Interface to the system

After data has passed from the input device up the event bus line to the filter module, it's passed to the system. There are currently two interfaces to the system:

Photon interface

> This requires that the Photon server is running. It passes data from the input to Photon via raw system events. Keyboard data is given by raw keyboard events, while relative and absolute data is given by raw pointer events. See the Photon docs for more on Photon events.

Resource manager interface

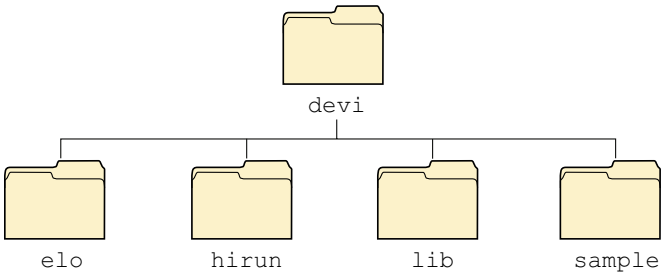
> This interface establishes a pathname under the **/dev** directory, which can be read by applications to get input data. For example, a relative event bus line would be represented by the file **/dev/mouse0**. Reading from **/dev/mouse0** would provide pointer packets, as defined in **<sys/dcmd_input.h>**. Multiple opens are allowed, and device files can be opened in blocking or non-blocking mode, with I/O notification (i.e. *select()*, *ionotify()*) supported.

The default interface started by the input system is the Photon interface. If you want to run input drivers without Photon, then you'd use the resource manager interface.

You start the resource manager interface by passing the **-r** option to the **devi-*** driver. You can disable the Photon interface by passing the **-P** option to the **devi-*** driver.

Source file organization for devi-*

The input (or **devi-***) source base is organized as follows:



How the input SDK source base is organized.

The **lib** directory contains "glue" code used by all drivers. It contains the command-line parsing code, the code used to manipulate modules and event bus lines, the code for the Photon and resource manager interfaces, as well as the filter modules (**rel**, **abs**, and **keyboard**). In addition, the **lib** directory contains functions used by modules to request services of the input system (e.g. for attaching interrupts and pulse handlers, mapping device I/O space, etc.)

We don't recommend changing anything in the **lib** directory. The source code is there simply to aid in understanding and debugging.

The **hirun** directory is an example of an actual input driver, **devi-hirun**. In this directory, you'll find various device and protocol modules.

The **elo** directory contains source for the "ELO" touchscreen drivers.

The **sample** directory contains sample code with lots of comments detailing the steps required to initialize your module, and what to put in your module's functions.

When writing your own input driver, you would create your own directory and put your new input modules there.

Chapter 2

Writing an Input Device Driver

In this chapter. . .

Creating an input module

To write an input driver, you must first create your own input module. The **sample** directory contains a sample skeleton for creating a module. We recommend that you use this as a starting point.

A module is represented by a data type called **input_module_t**. It contains various data fields and function pointers representing its interface.

input_module_t data type

Writing an input module consists of simply creating an **input_module_t** representing your module and filling in the relevant interface functions.

```
struct _input_module {

    input_module_t   *up;        // Up and down modules in bus line -
    input_module_t   *down;      // for internal use only
    struct Line      *line;      // driver bus line - for internal use only
    int               flags;     // module flags
    int               type;      // type of module
    char              name[12];  // module name (used in devi-* commands)
    char              date[12];  // date of compilation
    const char       *args;      // list of module args (used in devi-* commands)
    void             *data;      // private module data
    // pointers to user-supplied module functions
    int             (*init)(input_module_t *);
    int             (*reset)(input_module_t *);
    int             (*input)(input_module_t *, int, void *);
    int             (*output)(input_module_t *, void *, int);
    int             (*pulse)(message_context_t *, int, unsigned, void *);
    int             (*parm)(input_module_t *, int, char *);
    int             (*devctrl)(input_module_t *, int, void *);
    int             (*shutdown)(input_module_t *, int);
};
```

flags Only one flag has been defined — MODULE_FLAG_INUSE, which indicates a valid module.

type A combination (**OR**) of two descriptors:

- driver class:

  - DEVI_CLASS_KBD — keyboard
  - DEVI_CLASS_REL — relative
  - DEVI_CLASS_ABS — absolute
- driver layer that this module represents:
  - DEVI_MODULE_TYPE_FILTER — filter
  - DEVI_MODULE_TYPE_PROTO — protocol
  - DEVI_MODULE_TYPE_DEVICE — device

args List of module parameters where each parameter is represented by a single character. If there's an optional argument, the parameter has the format *x*`:` (the `:` means that the optional argument is expected).

data Usually a pointer to a module's local data. This can be assigned in the *init()* module function.

In the `sample` directory

The code in the **`sample`** directory provides lots of comments detailing the steps required to initialize your module, and what to put in your module's functions.

You'll also find two modules:

- **`samp_dev`** — an example of a device module.
- **`samp_proto`** — the MS-mouse protocol code with lots of comments.

You'll also find a **`README`** file that provides further background info on how the system processes data from **`keyboard`** and **`absolute`** devices.

In many embedded systems, a combination *device/protocol module* is called for. For details, see the section on "Writing a combination device/protocol module" in this chapter.

Data format

Device modules can pass data in any format they want up to protocol modules. But protocol modules must pass data *in a specific format* to filter modules.

This protocol module:	Must format data into a:
Keyboard	**`struct packet_kbd`**
Relative	**`struct packet_rel`**
Absolute	**`struct packet_abs`**

See the header **`<devi.h>`** for the format of these structures. All these structures have a timestamp field; you fill them in using the library call *clk_get()*.

Keyboard devices

When writing keyboard device modules, keep in mind that the protocol/filter layers will expect make-and-break scan codes indicating when a key is pressed down and released. The easiest thing to do is to map the scan codes your device sends to the standard PC scan codes. This way you won't have to make any filter-layer changes — it will all just work like a normal PC keyboard. Standard PC scan codes are available in any PC hardware book.

When passing up a **`struct packet_kbd`** to the filter layer, all you need to do is:

1. Fill in the *key_scan* field of the **`struct _keyboard_data`** with the scan code.
2. Fill in the *flags* field with KEY_SCAN_VALID.

The keyboard filter layer will read in a *keyboard definition file* and interpret the scan codes it receives based on the contents of this file.

The keyboard definition files are typically kept in the location $PHOTON_PATH/**keyboard**, where **$PHOTON_PATH** depends on you system configuration (e.g. this might be **/usr/photon** on your machine). In this directory there's a file called **sample.kdef**, which provides a sample definition file. The **.kdef** files are compiled into **.kbd** files using the utilities **kbcvt** and **mkkbd**.

Both of these utilities are shipped with Photon for QNX 4.

You shouldn't have to play around with these mapping files very much if you map your scan codes appropriately. The only place where you might need to modify these files is if your keyboard has special keys. In this case, you would start with a standard definition file (e.g. **en_US_101.kdef**), and add your unique scan codes.

When the driver starts up and initializes the keyboard filter module, the module will try to load in a mapping definition file. It uses the following algorithm to look for the file:

1. The module tries to open the keyboard configuration file **/etc/system/trap/.KEYBOARD.***hostname*. If this file exists, the module just reads the keyboard filename from it.
2. If the keyboard mapping filename is empty, the module tries to take it from the **KBD** environment variable.
3. If the keyboard mapping filename is still empty, the module assigns the standard US keyboard definition file (**en_US_101.kbd**) to it.
4. The module tries to find this file in the **%PHOTON%/keyboard** directory.
5. If the **PHOTON** environment variable isn't defined, the module tries to open it in the **/usr/photon/keyboard** directory.

Absolute devices

The **elo** directory contains an example of a touchscreen protocol module.

Absolute devices (e.g. touchscreens) need to be calibrated. They typically generate "raw" coordinates that must be translated into actual screen coordinates. The screen coordinates they're translated into depend on the screen resolution.

The device/protocol layer module receives raw coordinates from the touchscreen device, formats a **packet_abs** structure, and passes it up to the absolute filter.

The absolute filter module takes care of translating raw coordinates into screen coordinates. To do this, the module tries to locate and read in a *calibration file* on startup via:

1. Command-line option to the absolute filter (**-f** *filename*)
2. **ABSF** environment variable
3. **/etc/system/trap/calib.***hostname*

Calibration file format

The format of this file is as follows:

XL*x***YL:XH***x***YH:XRL XRH YRL YRH SWAP**

where:

XL	X screen coordinate of upper left side (typically 0).
YL	Y screen coordinate of upper left side (typically 0).
XH	X screen coordinate of lower right side (typically X screen resolution – 1).
YH	Y screen coordinate of lower right side (typically Y screen resolution – 1).

XRL	Raw touchscreen X coordinate at upper left side.
XRH	Raw touchscreen X coordinate at lower right side.
YRL	Raw touchscreen Y coordinate at upper left side.
YRH	Raw touchscreen Y coordinate at lower right size.
SWAP	Whether to swap X or Y axes (0 is no, 1 is yes.) It's safe to leave this as 0.

This calibration file is typically generated by the Photon touchscreen calibration application, **`calib`**. When the utility starts, it sends a message to the **`devi-`** driver asking it to switch to raw mode, and then solicits coordinate info by asking the user to touch the screen at all four corners and the middle. After doing this, **`calib`** formats the **`absf`** file, sends a calibration message to the **`devi-*`** driver, and writes the file.

Relative devices

The **`hirun`** directory contains examples of a mouse device (**`kb.c`**) and protocol (**`msoft.c`**, **`ps2.s`**, **`msys.c`**) modules.

Since these modules cover all the main types of relative devices, you probably won't need to develop something new from scratch. If you need to implement support for any device that's not completely supported by this driver, you can simply copy the files from this directory into a new one and modify them.

Note that Microsoft and Mouse Systems class devices don't have a device module — they just use **`/dev/ser`***N* to get raw data from a serial communication port. A PS/2 mouse shares the 8042 controller device driver (**`kb.c`**) with a standard keyboard.

The protocol layer module receives raw coordinates from the mouse, formats a **`packet_rel`** structure, and then passes it up to the relative filter.

The relative filter module implements an acceleration algorithm, converts raw data received from the protocol level according to the

current speed parameter, and emits this data in the form of events to Photon.

Callbacks in your module

The main part of developing a new module involves implementing several standard callback functions, combined "under the roof" of the module's instance of the **`input_module_t`** structure.

Consider implementing the following callbacks:

init()	Should be called for a one-time initialization of a module's state after it's loaded.
reset()	Used to reset a module's and/or device's state. You would call it when the module is linked into an event bus line; if necessary, it could be called from your code as a reaction to any sort of device trouble.
input()	You usually implement this callback function in protocol modules as part of the device-to-interface data channel.
output()	Usually called by higher-layer modules asking for data to be sent to the device. You can use this callback for passing commands to control an input device.
pulse()	Usually implemented in device class modules. This callback is automatically activated each time that a registered interrupt handler wants to notify a device module about input activity.
parm()	Called by the Input Runtime System to parse any command-line parameters given to the module.
devctrl()	Used by modules in an event bus line to send commands to each other. This callback may also be called as a response to the external *devctl()* call. You can use this callback for reconfiguring a driver on the fly.

shutdown() Called when the Input Runtime System is shutting down.

Which callbacks are required?

To decide which callback functions should be implemented in a module, you'll need to consider the module's purpose. In general, a *device module* must have the following functions:

- *pulse()* (if it doesn't use an interrupt handler)
- *init()*
- *parm()*
- *devctrl()*

A protocol module, in turn, must have at least the *input()* function (and optionally *init()*, *parm()*, and *devctrl()*).

Callback sequence

At startup, the Input Runtime System always calls a module's callback functions in the following sequence:

init() → *parm()* → *reset()*

Writing a combination device/protocol module

If you're writing a driver for a custom type of device where it doesn't make sense to split up the functionality of *device* and *protocol*, you can write a combination module.

To do this, you simply proceed as you would when writing a "normal" driver: fill in your callbacks, talk to your device, interpret its protocol, etc.

In addition, there are two things you have to do:

1. In the type field, put in DEVI_MODULE_TYPE_DEVICE | DEVI_MODULE_TYPE_PROTO in addition to the DEVI_CLASS_ manifest.
2. When you've interpreted the data from your device, package up a **`struct packet_*`** (depending on your class of device) and send it up.

A note about reentrancy

Because the **`devi-*`** framework is multithreaded, you should be aware of a possible reentrancy issue. When a **`devi-*`** driver is invoked, a module may be specified multiple times, where each invocation will belong to a separate event bus line.

An example is the keyboard controller device module (**`kb`**). This module can communicate with a keyboard and with a PS/2 mouse. We would invoke the driver as follows:

```
devi-hirun kbd kb ps2 kb -2
```

Here we'll have two event bus lines: one for the keyboard and one for the mouse. Upon initialization, the input framework will use the static **`kb`** data structure (**`input_module_t`**) for one of the bus lines and dynamically allocate/copy another one for the other bus line.

If you keep your module-specific data confined to the private data member of the module structure, you won't have any problems with reentrancy. But if your module contains global variables, then you'll have to use some sort of mutual exclusion mechanism for protection.

Note that you don't have to ensure that the *init()*, *reset()*, and *parm()* callbacks are reentrant, because they're always called from a single thread upon initialization. (However, if for some reason you need to call them when the runtime system is up, then you'd have to ensure that they're reentrant.) The callbacks used at runtime (e.g. the *pulse()* callback) are the ones at risk.

For more information, see the keyboard controller module code (**`hirun/kb.c`**).

Chapter 3

Testing and Debugging Your Driver

You can use the standard debugger to debug your driver code or you can use the old *printf()* method.

If you're using *printf()*, make sure you specify at least one **-v** command-line option to your driver. Otherwise, the **devi** lib will close *stdout*.

Debugging a keyboard/mouse driver

If you're going to use the standard debugger for a keyboard/mouse driver, perhaps the most convenient method is to use remote debugging via **telnet**.

This approach helps you avoid eventual problems caused by the contamination of test data with debug input activity.

Note that you can run your driver without graphics being started simply by starting the Photon server first.

If you're writing a touchscreen driver and are just testing out getting raw coordinates, then you can use the **-G** option to **devi-*** to tell it *not* to search for a graphics region when it starts up.

PS/2 mouse

While testing a PS/2-type of mouse, you can use the **-d** *filename* parameter of the device-level module (see **kb.c**) in order to collect and analyze raw data received from the device.

Keyboard filter module

In most cases, you don't need to redesign the keyboard filter module. But if you need to, you can use the **-p** *filename* command-line option to **devi-hirun** in order to debug your program separately without affecting the existing input system.

The protocol module **kbd** uses the supplied filename to create and open a FIFO file and then duplicates to this file all data passed to the standard filter module.

To access this data, your application should always be READ-blocked on this file. Alternatively, you can use the **`-f`** *filename* option to **`devi-hirun`** to simply create a separate file with the same data and then use this data for debug purposes.

Chapter 4

Module Functions

You can create a module by providing the functions listed here and by saving pointers to them in the **`input_module_t`** structure.

Function	Summary
devctrl()	Allow for configuration of a module by an external source
init()	Initialize a module's private data
input()	Pass data to a higher layer module
output()	Pass data to a lower layer module
parm()	Process the command-line arguments to the module
pulse()	Process data gathered by the interrupt handler
reset()	Reset the module to its initial state
shutdown()	Clean up when the input manager is about to terminate

devctrl()

Allow for the configuration of a module by an external source

Synopsis:

```
static int devctrl( inout_module_t *pModule,
                    int event,
                    void *ptr );
```

Description:

This function lets an external source — typically another module — configure a module. It's invoked by an output module at arbitrary points of execution.

The *pModule* parameter holds a pointer to a module descriptor.

What this function should do depends on which layer it's in:

Device	If *event* is meaningful, assign type to *ptr* and process. Read and update the pointer as appropriate. If *event* isn't meaningful, return -1 and set *errno* to EINVAL.
Protocol	If *event* is meaningful, assign type to *ptr* and process. Read and update the pointer as appropriate. If *event* isn't meaningful and there's a linked input module, invoke the input module's *devctrl()* callback.

To add a new DEVCTL command, make an entry in **`include/const.h`** and define any data sent with the DEVCTL in **`struct.h`**.

Returns:

EOK

Classification:

Your code

Safety	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Initialize a module's private data

Synopsis:

```
static int init( input_module_t *pModule );
```

Description:

This function initializes the data structure pointed to by *pModule*. It's invoked by the Input Runtime System during startup before the module is linked into an event bus.

This function should allocate and initialize any private data in the data structure.

Returns:

EOK	Success.
ENOMEM	There isn't enough memory to allocate private data member.

Examples:

See any module.

Classification:

Your code

Safety	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

input()

Pass data to a higher-layer module

Synopsis:

```
static int input( input_module_t *pModule,
                  int num,
                  void *ptr );
```

Description:

This function is used to pass data to a higher-layer module. It's called by another module from either its *input()* or *pulse()* callback.

This function should process *num* elements of data located at *ptr*.

Returns:

EOK

Examples:

See **protocol/msoft.c**.

Classification:

Your code

Safety	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

output()

Pass data to a lower-layer module

Synopsis:

```
static int output( input_module_t *pModule,
                   void *ptr,
                   int num );
```

Description:

This function is used to pass data to a lower-layer module. It's called by an output module at arbitrary points in execution.

This function should:

- Read *num* elements of data from *ptr* and send it to an external device.

  Or

- Call the next lower layer's *output()* callback.

Returns:

EOK

Examples:

See **`device/uart.c`**.

Classification:

Your code

Safety	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

parm()

Process the command-line arguments to the module

Synopsis:

```
static int parm( message_context_t *pContext,
                 int opt,
                 void *optarg );
```

Description:

This function processes the command-line arguments passed to the module. It's called by the Input Runtime System at startup.

This function is invoked once for each option letter or option-letter/option-argument pair. It's called only if *opt* is meaningful to *pContext* (i.e. the *args* member of *pContext* points to a string that contains the character *opt*).

This function should process the option letter, *opt*, along with its option arguments *optarg*, if applicable.

Returns:

EOK

Classification:

Your code

Safety	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

pulse()

Process data gathered by the interrupt handler

Synopsis:

```
static int pulse( message_context_t *pContext,
                  int code,
                  unsigned flags,
                  void *data );
```

Description:

This function processes data gathered by the interrupt handler. It's called by the Input Runtime System on receiving a pulse or proxy associated with this callback.

This function should handle the condition being signalled by the pulse. At some application-determined point, it should call the output module's *input()* callback.

The arguments are:

pContext	Pointer to message context structure.
code	Value that you specify as the third parameter of the *devi_register_interrupt()* call.
flags	This parameter isn't used.
data	Pointer to module descriptor.

Returns:

EOK

Examples:

See **`device/uart.c`**.

Classification:

Your code

Safety	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Reset the module to its initial state

Synopsis:

```
static int reset( input_module_t *pModule );
```

Description:

This function resets the module to its initial state. It's called by the Input Runtime System at startup. The module has already been linked into an event bus line by the time this function is called.

If your module includes an interrupt handler, *reset()* should call *devi_register_pulse()* and store the pulse code it returns in the module's private data.

Returns:

EOK

Examples:

See any module.

Classification:

Your code

Safety	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

shutdown()

Clean up when the input manager terminates

Synopsis:

```
static int shutdown( message_context_t *pContext,
                     int delay );
```

Description:

This function is used to perform any required cleanup before the **devi-*** program terminates. It's called by the Input Runtime System when a terminal signal has been caught.

This function should bring the state of the system to a point suitable for process termination.

The arguments are:

pContext	Pointer to the message context structure
delay	Reserved for future use. For now use 0.

Returns:

EOK

Examples:

See **device/kb.c**.

Classification:

Your code

Safety	
Interrupt handler	Yes
Signal handler	Yes
Thread	Yes

Chapter 5

Input API Reference

The Input DDK includes the following interface functions:

Function	Summary
begin()	Initialize resource manager and activate driver bus line
buff_append()	Append bytes to circular buffer
buff_create()	Create a circular buffer
buff_delete()	Delete bytes from circular buffer
buff_flush()	Clear the circular buffer
buff_getc()	Get character from circular buffer
buff_putc()	Place character on circular buffer
buff_waiting()	Determine the number of bytes stored in circular buffer
clk_get()	Get the time from the OS Process Manager
devi_enqueue_packet()	Dispatch a completed packet
devi_register_interrupt()	Attach to an interrupt
devi_register_pulse()	Associate a pulse code with a function
devi_register_timer()	Create a timer
devi_request_iorange()	Map device registers into memory

begin()

Initialize resource manager and activate driver bus line

Synopsis:

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

Description:

This function initializes the resource manager and activates the driver bus line. The *begin()* function should be called at the *end* of your initialization process, because it never returns control until the program is closed.

The arguments are:

argc The number of elements in the *argv* array.

argv An array of command-line arguments.

Returns:

Always returns 0.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

buff_append()

Append bytes to circular buffer

Synopsis:

```
int buff_append( buffer_t *bptr,
                 char *dptr,
                 int n );
```

Description:

This function appends *n* bytes from *dptr* (pointer to data block) to the circular buffer *bptr*.

Returns:

The number of bytes currently stored in the buffer *bptr*.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

See also:

buff_create(), *buff_delete()*, *buff_flush()*, *buff_getc()*, *buff_putc()*, *buff_waiting()*

buff_create()

Create a circular buffer

Synopsis:

```
struct buffer *buff_create( unsigned size,
                            unsigned rsize );
```

Description:

This function creates a circular buffer made up of *size* number of *rsize* records.

Returns:

A pointer to an initialized buffer handle, or NULL if memory for the buffer data couldn't be allocated.

All the other *buff_*()* functions take as their first argument the handle returned by the successful execution of *buff_create()*.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

See also:

buff_append(), *buff_delete()*, *buff_flush()*, *buff_getc()*, *buff_putc()*, *buff_waiting()*

buff_delete()

Delete bytes from the circular buffer

Synopsis:

```
int buff_delete( buffer_t *bptr,
                 char *dptr,
                 int n );
```

Description:

This function removes *n* bytes from the circular buffer *bptr* and places them in *dptr* (pointer to data block).

Returns:

The number of bytes removed from the buffer *bptr*.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

See also:

buff_create(), *buff_append()*, *buff_flush()*, *buff_getc()*, *buff_putc()*, *buff_waiting()*

buff_flush()

Clear the circular buffer

Synopsis:

```
int buff_flush( buffer_t *bptr );
```

Description:

This function clears the circular buffer *bptr*.

Returns:

EOK.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

See also:

buff_append(), *buff_create()*, *buff_delete()*, *buff_getc()*, *buff_putc()*, *buff_waiting()*

buff_getc()

Get a character from the circular buffer

Synopsis:

```
int buff_getc( buffer_t *bptr );
```

Description:

This function retrieves a character from the circular buffer *bptr*, removing it from the buffer in the process.

Returns:

The character currently at the head of the circular buffer *bptr*, or -1 if the buffer is empty.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

See also:

buff_append(), *buff_create()*, *buff_delete()*, *buff_flush()*, *buff_putc()*, *buff_waiting()*

buff_putc()

Place a character on the circular buffer

Synopsis:

```
int buff_putc( buffer_t *bptr,
               char c );
```

Description:

This function places a character *c* on the circular buffer *bptr*.

Returns:

The number of bytes currently stored in the buffer *bptr*.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

See also:

buff_append(), *buff_create()*, *buff_delete()*, *buff_flush()*, *buff_getc()*, *buff_waiting()*

buff_waiting()

Determine the number of bytes stored in the circular buffer

Synopsis:

```
int buff_waiting( buffer_t *bptr );
```

Description:

This function determines the number of bytes stored in the circular buffer *bptr*.

Returns:

The number of bytes currently stored in the buffer *bptr*.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

See also:

buff_append(), *buff_create()*, *buff_delete()*, *buff_flush()*, *buff_getc()*, *buff_putc()*

clk_get()

Get the time from the OS Process Manager

Synopsis:

```
void clk_get( struct timespec *tspec );
```

Description:

This function loads the **`timespec`** structure pointed to by *tspec* with the current time from **`procnto`**'s internal clock.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

devi_enqueue_packet()

Dispatch a completed packet

Synopsis:

```
int devi_enqueue_packet( input_module_t *module,
                         char *dptr,
                         unsigned size ) ;
```

Description:

This function is used by filter-layer modules to dispatch a completed packet to the proper interface, to Photon, or to a resource manager.

The arguments are:

module	Name of module data block.
dptr	Pointer to data block.
size	Size of data block.

Returns:

0 on success, -1 on error.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

devi_register_interrupt()

Attach to an interrupt

Synopsis:

```
int devi_register_interrupt( int intr,
                             int prio,
                             int *pc,
                             input_module_t *module,
                             struct sigevent *evp,
                             unsigned flags );
```

Description:

This function lets you attach to an interrupt. The default method of attachment is to use *InterruptAttachEvent()* and send back a pulse when the IRQ triggers. But you can override this behavior by passing in your own event structure in the fifth parameter (*evp*).

For example if you wanted to spawn a separate interrupt-handling thread and process interrupts within it, you could set up the event structure to send back SIGEV_INTR.

The arguments are:

intr — Interrupt number (IRQ) you're going to register.

prio — Dispatch priority that will be assigned to the callback function that processes a pulse. The priority is generated by an interrupt handler.

pc — Pulse code. If the *pc* argument isn't NULL and isn't equal to DEVI_PULSE_ALLOC, then it will be used as the pulse code to associate the *pulse()* callback.

module — Name of module data block.

evp — Pointer to **sigevent** structure. If provided, *evp* is attached to the interrupt (see *InterruptAttachEvent()* in the QNX Neutrino *Library Reference*.)

If *evp* is NULL, a pulse will be allocated and the module's *pulse()* callback will be associated with it.

flags Can be 0 or DEVI_SHARE_RSRC. If the *flags* argument is set to DEVI_SHARE_RSRC, this will tell the resource database manager to allow this interrupt to be shared. If the *pc* parameter wasn't NULL, then the allocated pulse code will be returned in it.

Returns:

A valid interrupt ID, or -1 on error.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

devi_register_pulse()

Associate a pulse code with a function

Synopsis:

```
int devi_register_pulse( input_module_t *module,
                         int code,
                         int (*func)(message_context_t *,
                                     int,
                                     unsigned,
                                     void *));
```

Description:

This function associates a pulse code with a function. When the input runtime system receives a pulse with the specified code, it will call the associated function.

The arguments are:

module	Name of module data block.
code	Can be 0 or MSG_FLAG_ALLOC_PULSE, in which case a pulse code will be allocated for the caller.
func	The name of the function to associate. If it's NULL, then the module's *pulse()* callback is used.

Returns:

Valid pulse code used to associate the function, or -1 on error.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

devi_register_timer()

Create a timer

Synopsis:

```
timer_t devi_register_timer( input_module_t *module,
                             int prio,
                             int *pc,
                             struct sigevent *evp );
```

Description:

This function creates a timer. To arm the timer, the caller must call the OS library *time_settime()* function.

The default notification is a *pulse* when the timer expires. You can override this by passing a prebuilt event in *evp*.

The arguments are:

module	Name of module data block.
prio	Dispatch priority that will be assigned to the callback function that processes a pulse. The priority is generated by an interrupt handler.
pc	Pulse code. If the *pc* argument isn't NULL and isn't equal to DEVI_PULSE_ALLOC, then it will be used as the pulse code to associate the *pulse()* callback.
evp	Pointer to **sigevent** structure. If provided, *evp* is attached to the interrupt (see *InterruptAttachEvent()* in the QNX Neutrino *Library Reference*.)

Returns:

0 on success, -1 on error.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

devi_request_iorange()

Map device registers into memory

Synopsis:

```
uintptr_t devi_request_iorange( unsigned start,
                                unsigned len,
                                unsigned flags );
```

Description:

This function maps device registers into memory.

The arguments are:

start	Start of I/O port area to map.
len	Length of I/O port area.
flags	The *flags* parameter can be set to DEVI_SHARE_RSRCM to indicate that the I/O range can be shared.

Returns:

A pointer to the first location in the mapped-in range or MAP_FAILED on error.

Classification:

QNX

Safety	
Interrupt handler	Not applicable
Signal handler	Not applicable
Thread	Not applicable

Index

A

B

C

D

E

O

P

R

S

T