QNX® Neutrino® RTOS
Technical Notes

BlackBerry | QNX

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 These Technotes

For this release of the QNX Neutrino RTOS, you'll find the following technotes here:

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 Compiling OpenVPN for QNX Neutrino

OpenVPN is an open-source application to securely tunnel IP networks over a single TCP/UDP port, with support for SSL/TLS-based session authentication and key exchange, packet encryption, packet authentication, and packet compression. Here's how to compile it on QNX Neutrino:

1. Download the OpenVPN source from *http://swupdate.openvpn.org/community/releases/openvpn-2.3.11.tar.gz*.
2. Untar the file, and then enter the created directory:

   ```
   tar -zxvf openvpn-2.3.11.tar.gz
   cd openvpn-2.3.11
   ```

3. Create a **build-hooks** script to set target-specific settings for the `addvariant` tool:
   - Add the **io-pkt** directory to include paths for **types_bsd.h**.
   - Set `fork_works` to `yes` for QNX targets.
   - Disable `auth-pam` for QNX targets.

   For example:

   ```
   echo '#!/bin/ksh
   function hook_preconfigure {
        CPPFLAGS="$CPPFLAGS -I$QNX_TARGET/usr/include/io-pkt"
        configure_opts="${configure_opts} --disable-plugin-auth-pam"
        configure_opts="${configure_opts} ac_cv_func_fork_works=yes"
   }
   ' > build-hooks
   chmod 777 build-hooks
   ```

4. Add the QNX Neutrino target type to the **configure** script. Edit this file, search for the `*-*-linux*)` line, and add the following above it:

   ```
       *-*-qnx*)

   $as_echo "#define TARGET_QNXNTO 1" >>confdefs.h


   cat >>confdefs.h <<_ACEOF
   #define TARGET_PREFIX "Q"
   _ACEOF

           ;;
   ```

5. Add QNX Neutrino target to the **config.h.in** file. This change defines TARGET_NETBSD, so that QNX Neutrino is handled like NetBSD. Search for `Are we running on`, and add the following:

```
/* Are we running on QNX Neutrino? */
#undef TARGET_QNXNTO

#ifdef TARGET_QNXNTO
/* QNX mirrors the NetBSD implementation */
#define TARGET_NETBSD
#endif
```

6. Update the source to include the NetBSD headers; edit **src/openvpn/tun.c**, search for `PR 32944`, and add the following after `#elif defined (TARGET_NETBSD)`:

```
#ifdef TARGET_QNXNTO
#include <sys/types_bsd.h>
#endif
```

7. Update IV_PLAT to indicate QNX Neutrino; edit **src/openvpn/ssl.c**, search for `IV_PLAT=linux`, and add the following after that line:

```
#elif defined(TARGET_QNXNTO)
      buf_printf (&out, "IV_PLAT=qnx\n");
```

8. Update the source to use IP_RECVDSTADDR instead of IP_PKTINFO, because `struct in_pktinfo` isn't exactly the same as for other targets. Edit **src/openvpn/socket.h**, and add the following after the declaration of `struct openvpn_sockaddr`:

```
#ifdef TARGET_QNXNTO
 #undef HAVE_IN_PKTINFO
 #undef IP_PKTINFO
#endif
```

9. Set up the QNX SDP build environment:

```
source base_directory/qnxsdp-env.sh
```

where *base_directory* is where you installed QNX SDP.

10. Add the supported targets:

```
addvariant nto arm le-v7
addvariant nto x86 o
addvariant nto aarch64 le
addvariant nto x86_64 o
```

11. Build the targets:

```
make
```

The binaries are located in the specific directory for each target. For example, the **openvpn** binary is placed in the following:

- **./nto-arm-le-v7/src/openvpn/openvpn**
- **./nto-x86_64-o/src/openvpn/openvpn**
- **./nto-x86-o/src/openvpn/openvpn**
- **./nto-aarch64-le/src/openvpn/openvpn**

Chapter 2
IP Tunneling (Generic Routing Encapsulation)

GRE (Generic Routing Encapsulation) or IP tunneling (IP encapsulation) is a technique that encapsulates IP datagrams within IP datagrams. GRE is a technique that allows datagrams to be encapsulated into IP packets and then redirected to an intermediate host. At this intermediate destination, the datagrams are decapsulated and then routed to the next leg. In doing so, the trip to the intermediate host appears to the inner datagrams as one hop. The general outline of GRE can be found in *RFC 1701*.

In the current stack, GRE is performed via GRE pseudo interfaces which simulate point-to-point connections. You can use the `ifconfig` utility to create GRE interfaces:

```
# ifconfig gre0 create
```

Each GRE interface supports the following modes of operation:

GRE encapsulation (default)

Outgoing datagrams are encapsulated by an IP header of protocol type 47, and a GRE header specifying the type of the encapsulated datagram (currently only IP). This mode is described in *RFC 1702*. It's also the default mode on Cisco routers.

MOBILE encapsulation (IP protocol number 55)

Applicable for IP encapsulation only. Outgoing IP datagrams are encapsulated by a smaller header, and the original IP header is modified. This mode is described in *RFC 2004*.

How do I use it?

Let's illustrate by use of an example:

Suppose you have the following two machines at your disposal on the ubiquitous LAN:

- machine A with IP address 10.0.0.25
- machine B with IP address 10.0.0.163

We'll assume that they're connected by a “real” interface, and that they can communicate through generic methods. Here's how you create a tunnel between “pseudo” addresses 12.0.0.1 and 12.0.0.2:

On machine A (10.0.0.25), enter:

```
# ifconfig gre0 create                <- create the pseudo device
# ifconfig gre0 12.1 12.2 link1       <- set the local (inner) addrs
# ifconfig gre0 tunnel 10.25 10.163   <- set the encapsulating (outer) addrs
# ifconfig gre0                       <- print the current state
```

The IP packet that's sent from machine A looks like this:

```
+------------+----------+------+
| dst 10.163 | dst 12.2 | data |
+------------+----------+------+
\_IP address_/\______data______/
```

On machine B (10.0.0.163), do the reverse:

```
# ifconfig gre0 12.2 12.1 link1
# ifconfig gre0 tunnel 10.163 10.25
# ifconfig gre0
```

Since the IP packet is decapsulated on machine B, it now looks like this:

```
+------------+------------------+
|  dst 12.2  |    data          |
+------------+------------------+
\_IP address_/\______data______/
```

From machine A, you should now be able to ping 12.0.0.2. What happens under the covers is an IP packet with source 12.0.0.1 and destination 12.0.0.2 is encapsulated with the "real" source 10.0.0.25 and destination 10.0.0.163.

That's interesting but not very useful since you would have accomplished the same thing if you went to 10.0.0.163 directly. Let's take a look at a more typical example.

A more typical example

Let's say you usually work at the office on machine A that has an IP address 10.0.0.25 (we'll use the same IP addresses as in the above example). You use the corporate LAN and you have access to all the machines on your subnet. You've just been sent on a business trip and you'll be able to access the corporate gateway only using the Internet. Let's call the gateway machine C, and assume it has forwarding enabled between the following two "real" interfaces:

- 10.0.0.1 — attached to the corporate subnet.
- C.C.C.C — some address on the Internet.

From your remote location, you'll set up the tunnel in similar way to the example above. Assuming that A.A.A.A is machine A's Internet address:

```
# ifconfig gre0 10.25 10.1 link1
# ifconfig gre0 tunnel A.A.A.A C.C.C.C
```

Then set up a route for all of network 10 to the other end of the tunnel:

```
# route add -net 10 10.1
```

Now on to machine C. Either before you leave for your trip, or from your remote location, call your colleague in the IS department and get set up with a proxy `arp` entry on the gateway. When this is done, the machines behind the gateway will think that you're still at the office. Assuming interface `en0` is attached to the local subnet (network 10), they'd enter the following:

```
# arp -s 10.0.0.25 $(netstat -in | grep en0 | grep Link | cut -c 27-43) pub
```

The command in the brackets cuts out machine C's `en0` MAC address and passes it on to the `arp` command. These commands set up the tunnel:

```
# ifconfig gre0 10.1 10.25 link1
# ifconfig gre0 tunnel C.C.C.C A.A.A.A
```

You should now have transparent remote access to your corporate LAN—just as if you were sitting at your desk!

Final tidbits

What's that `link1` stuff in the `ifconfig` command for?

It's a flag to the stack to suppress the creation of a nonspecific route that attempts to prevent loops when the inner destination address and real (tunnel) destination address are the same. The algorithm currently is to toggle the last bit in the tunnel destination.

This currently also applies to ipip tunneling, but there's no way to disable it as the inner and tunnel addresses are always the same for an ipip interface (ipip doesn't support the `tunnel` keyword to `ifconfig`).

For example, using an ipip interface to create a tunnel to from 10.1 to 10.2:

```
# ifconfig en0 10.1
# route add default 10.3
# ifconfig ipip0 11.1 11.2
```

The last command tries to create an implicit route to 11.3 (last bit of 11.2 is toggled) that resolves the default to 10.3, not the desired 10.2. The following shows how to reach the desired 10.2:

```
# ifconfig en0 10.1
# route add default 10.3
# route add -net 11 10.2
# ifconfig ipip0 11.1 11.2
```

The implicit route to 11.3 now resolves to the route to network 11 (gateway 10.2) instead of the default route.

If GRE encapsulation is the default, how do I enable MOBILE encapsulation?

Use `ifconfig` to clear the `link0` flag on the **gre** device. For example:

```
# ifconfig gre0 10.1 10.25 link1 -link0
```

Chapter 3
PPPOE and Path MTU Discovery

If you find that your PPPOE interface has problems with certain sites, it may be due to the general state of disarray of path MTU (Maximum Transmission Unit) discovery on the Internet.

PPPOE interfaces generally have an MTU of 1492 bytes, whereas the straight Ethernet interface that they sit on have an MTU of 1500 bytes.

When the stack initiates a TCP connection, it advertises the largest MTU of all its interfaces, as per RFC 1193, via the TCP `mss` option. This is done because, although the outgoing packets are going out on the PPPOE interface, there's a slight chance that packets coming back from the peer will actually come in on another interface, depending on the routing that occurs between you and the peer.

If a packet that's larger than 1492 comes back on the PPPOE interface, the router needs to handle it via fragmentation or path MTU discovery.

If the peer (in our example) sends a packet back to us with an MTU of 1500 (which we advertised as per the above) that happens to be routed back to our PPPOE interface of MTU 1492, the router before the PPPOE has two options:

- If the peer isn't using path MTU discovery, the router can fragment the packet into two smaller ones and send them on their way. This results in more overhead, but things should continue transparently.
- If the peer is using path MTU discovery in an effort to avoid the overhead of fragmentation, it has to set the DF (don't fragment bit) in the IP header. In doing this, it expects the router to drop the packet and send back an “ICMP fragmentation required” packet containing the new MTU (1492 in this example). This is where the problem often arises.

  Some routers appear not to generate the “ICMP fragmentation required” packet. More often, these packets are filtered out by intermediate firewalls. In either case, the peer has no indication that anything is wrong, and continues to generate packets of MTU 1500 with the DF bit set, which in turn are dropped by an intermediate router.

What can you do about this?

Setting the `mss_ifmtu` option on makes the stack use the MTU of the outgoing interface in the initial advertisement, rather than the maximum of all its interfaces. In accordance with RFC 1193, this option is off by default.

You can query and set the `mss_ifmtu` option by using the `sysctl` utility:

```
# sysctl net.inet.tcp.mss_ifmtu
net.inet.tcp.mss_ifmtu = 0
# sysctl -w net.inet.tcp.mss_ifmtu=1
net.inet.tcp.mss_ifmtu: 0 -> 1
```

For more information about `sysctl`, see the *Utilities Reference*.

Chapter 4
Making Multiple Images

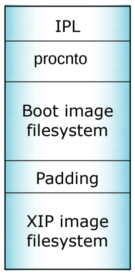
You can generate images as described in `mkifs` in the *Utilities Reference* and in OS Images in *Building Embedded Systems*.

When you use `mkifs` to build an OS filesystem, you can specify that particular executables will either execute in place (normally in flash) or be copied to RAM and executed there (see "Notes on UIP versus copy," in the `mkifs` entry in *Utilities Reference*). Executing in place saves a bit of RAM by avoiding the copying of the code and/or data segment of an object from one DRAM location to another.

But what if you want some executables to run from flash and others from RAM? That's what multiple images are used for.

Multiple image filesystems are typically used in execute-in-place (XIP) systems to separate executables that must run directly from flash (to conserve RAM) and those that should be copied and then executed from RAM (for performance).

Simply put, you create two separate images and then stitch them together. This is what the OS image will look like:



The boot image filesystem will be run from RAM to improve performance and the XIP image filesystem will be run from flash to conserve RAM.

Each of the three sections must begin at a memory page boundary (typically 4 KB). The IPL code will execute the OS in the first image.

- If you prefer to run the process manager XIP and other executables from RAM, you can adapt the following procedures by altering the buildfiles so that code from the boot image is used in place and code from the second image is copied into RAM.
- For another approach to mounting a secondary image filesystem, see the *Reloadable Image Filesystems* technote.

Restrictions on XIP image filesystems

There are some restrictions in XIP image filesystems:

- The IFS must be uncompressed.
- The IFS may not reside in NOR flash where there is a flash filesystem driver running for that same flash array (since the flash driver will occasionally put the flash in a mode where it can't be read as data).

- XIP might not work on all boards.
- In QNX Neutrino 6.4.0 or later, secondary image filesystems can be anywhere in memory; in earlier versions of the OS, they had to reside in the first 256 MB.

Mounting an IFS

Once the kernel is running, you can mount image filesystems. The syntax for this is:

`mount -tifs -o offset=`*XXXXXXXX*`,size=`*YYYYYYYY* `/dev/mem` */mountpoint*

The variables are:

XXXXXXXX

The physical address of an area of contiguous memory (linear) that contains a binary image filesystem.

YYYYYYYY

The size of the image, rounded up to the next highest page boundary.

/mountpoint

The name of the mountpoint for the image filesystem.

For example, suppose an image filesystem starts at physical address 0x600000 and is a size of 0x23A5C. You can use the `mkifs -v` command to determine the size when you build the image. The next highest 4-KB page boundary is 0x24000. Suppose we want to mount this image as **/ifs2**. The syntax would be:

```
mount -tifs -o offset=0x600000,size=0x24000 /dev/mem /ifs2
```

Using a second IFS

There are several options for using a second image filesystem:

- You can have a readable flash/PROM device that's directly accessible and has a unique memory address (e.g. flash at 0xD0000000).

  You can't use this flash device for an image filesystem and a flash filesystem (`devf-*`) at the same time.

- After booting the kernel, you can dynamically map contiguous RAM (*mmap_device_memory()*). Get the physical pointer to this memory, copy an image filesystem to it, and then mount the ifs.
- At boot time (in startup code), copy an area of flash that contains a prebuilt image filesystem in to a known area of reserved memory. Mount the second IFS after the kernel has booted.

In this technical note, we'll use the third option.

Example: mounting an IFS on a board

We'll use a fictitious board in this example. This board has a flash part that's 32 MB in size and sits in memory space from 0xFE000000 to 0xFFFFFFFF. The reset vector jumps to 0xFE000000.

You can design the flash any way you wish. Here's the layout that we'll use in this example:

From	To	Contains	Size
0xFE000000	0xFE03FFFF	IPL	256 KB
0xFE040000	0xFE2FFFFF	Boot image 1 (**os1**)	3 MB – 256 KB
0xFE300000	0xFE3FFFFF	Boot image 2 (**os2**)	1 MB
0xFE400000	0xFEFFFFFF	Flash filesystem mounted as **/**	28 MB

The IPL has been changed to start searching for **os1** at 0xFE040000. By default, the IPL will search from 0xFE010000 to 0xFE030000.

Sample buildfiles

First we need to make the **os1** and **os2** boot images. Let's look at the buildfiles.

The **os1.build** file looks like this:

```
[image=0x20000]
[virtual=armle-v7,binary +compress] .bootstrap = {

# reserve 4M of ram at the 6M physical address point. and zero it out (0 flag)

    startup-my_board-dual -vvv -r 0x600000,0x400000,0

   #######################################################################
   ## PATH set here is the *safe* path for executables.
   ## LD_LIBRARY_PATH set here is the *safe* path for libraries.
   ##     i.e. These are the paths searched by setuid/setgid binaries.
   ##          (confstr(_CS_PATH...) and confstr(_CS_LIBPATH...))
   #######################################################################
   PATH=:/proc/boot:/bin:/usr/bin LD_LIBRARY_PATH=:/proc/boot:/lib:/usr/lib:/lib/dll procnto-600
}
[+script] .script = {
    procmgr_symlink ../../proc/boot/libc.so.2 /usr/lib/ldqnx.so.2

    #######################################################################
    ## my_board
    #######################################################################
    display_msg Welcome to QNX Neutrino on my_board (OS1)
    devc-ser8250 -c 132000000 -u 3 -e -F -S -b115200 0xf0002400,67
    waitfor /dev/ser3
    reopen /dev/ser3

    SYSNAME=nto
    TERM=qansi
    HOME=/
    PATH=:/proc/boot:/bin:/usr/bin:/opt/bin
    TERM=qansi
    LD_LIBRARY_PATH=:/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib
    [+session] ksh &
}

[type=link] /bin/sh=/proc/boot/ksh
[type=link] /dev/console=/dev/ser1
[type=link] /tmp=/dev/shmem

[perms=+r,+x]
libc.so
fpemu.so.2

[data=c]
devc-serpsc

ls
ksh
pipe
pidin
uname
slogger2
slog2info
slay
mount
cp
mv
hd
spatch
#/usr/lib/terminfo=/usr/lib/terminfo
dumpmem=./dumpmem/dumpmem
umount
devf-my_board=/root/workspace/bsp-my_board_devf-my_board/my_board/arm/le/devf-my_board
flashctl
```

The **os2.build** buildfile is given below. Note that this image filesystem is always read-only.

```
# set search path for files
[search=/usr/qnx630/target/qnx6/armle-v7/bin:/usr/qnx630/target/qnx6/armle-v7/sbin:/usr/qnx630/ta

# code=uip means that the code will be run from the image filesystem
# (Use In Place).
#
# +raw means to not strip the programs in any way

[data=copy code=uip]
[perms=+r,+x]

# my files to include

# these files will reside directly under the mountpoint as specified
# with the mount command

[+raw] /raw/hd=hd
       /unraw/hd=hd
```

```
devc-ser8250
io-pkt-v4-hc

# these files will be placed

/bin/use=use
```

Unlike the build script for a bootable OS image, the XIP script doesn't have a boot section, nor does it have a boot script. If you run `mkifs -v` on this buildfile, you'll see that the first entry is the image filesystem header rather than the startup code normally found in a bootable image.

Programming into flash

In **os1.build**, the startup code reserves 4 MB of RAM at a physical address of 0x600000. This will be where the **os2** image filesystem will reside. The operating system won't use this memory range for other programs.

You'll need to program the srec files that will be created into flash. We'll assume that the best way to do this with our fictitious board is to use the dBUG tool and the `fp` command.

To program **os1.srec**:

1. Make sure your `tftpd` server is working.
2. Change the `filename` parameter in dBUG to look for this filename. For example:

   ```
   dBUG> set filename /xfer/os1.srec
   ```

   You can store this setting using the `store` command.
3. Download the S-records:

   ```
   dn
   ```
4. Program the S-records:

   ```
   fp 0 fe040000 fe2fffff 20000
   ```

Both the **os1.srec** and **os2.srec** files are created with offsets of 0x20000. This allows them to be downloaded in to RAM using the dBUG tool and then programmed. Be sure to include the offset (20000) when you flash program (`fp`).

Use the same programming technique for **os2.srec**, but change the start and end address:

```
fp 0 fe300000 fe3fffff 20000
```

Putting the images together

Note the following about the **os1.build** file:

- The startup code is doing the copy of flash from 0xFE300000 to RAM at 0x600000. This new startup was called **startup-my_board-dual**.

  In order to make this copy, edit the BSP's **bsp-startup-my_board/my_board/main.c** and modify the *main()* function:

  ```
  ...

  add_typed_string(_CS_MACHINE, "My board");

  /* Load the bootstrap executables in the image filesystem and
  ```

```
       initialize various syspage pointers. This must be the *last*
       initialization done before transferring to the next program. */

    init_system_private();

    /* At this point, copy the second image to RAM:
        - The source will be 0xFE300000  (physical flash address)
        - The destination will be 0x600000  (physical RAM location)

            Make sure the destination is the same physical address XXXXXX that
            is specified in the 'startup-my_board -r XXXXXX,YYYYYY,Z' in the
            buildfile.
        - The size to copy is 0x50000. The size to copy must be at least as
          large as the os2.ifs file. It is recommended to erase the flash area
          for os2.ifs so that any extra bytes at the end are zero.

       This copy must be done after the init_system_private() call. */

    copy_memory( 0xFE300000, 0x600000, 0x50000 );

    /* Dump the system page if verbose level 3 (-vvv) */
    if (debug_flag > 2)
    {
        print_syspage();
    }

    return 0;
```

- By default, the entire contents of SDRAM are zeroed when the board is initialized. If you don't want this to happen, then you will need to modify the startup code.
- When **startup-my_board-dual** is given the `-r` option to reserve system memory, the third argument is a flag that indicates if memory will be cleared. If you are warm-booting your CPU with memory still active, then you won't want to clear the memory. Set the flag to 0 to clear, or 1 to not clear.

It's possible to mount multiple image filesystems at the same mountpoint. For example:

```
mount -tifs -o offset=0x600000,size=0x24000 /dev/mem /
mount -tifs -o offset=0xa00000,size=0x31000 /dev/mem /
```

This mounts two additional images at the root (**/**) mountpoint. QNX Neutrino supports union filesystems. If there are duplicate paths to the same file, then the last image to mount will be the one that's used.

Test program

Here's a testing program called **dumpmem.c** that you can include in your first boot image, so that you can look at physical memory locations:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <inttypes.h>
#include <string.h>
#include <sys/neutrino.h>

int
```

```
main( int argc, char **argv )
{

    char *ptr;
    size_t len;
    uint64_t addr;
    long int ltmp;
    char c;

    int i;

    if ( argc < 3 ) {
        fprintf(stderr,"enter addr and size\n");
        exit(1);
    }

    addr = strtoull( argv[1], NULL, 0 );

    ltmp = strtol( argv[2], NULL, 0 );
    len  = ltmp;

    fprintf(stderr,"Dumping %d (0x%x) bytes at addr 0x%llx\n",
        len, len, addr );


    ThreadCtl( _NTO_TCTL_IO, 0);

    ptr = mmap_device_memory( 0, len, PROT_READ|PROT_WRITE|PROT_NOCACHE, 0, addr );
    if ( ptr == MAP_FAILED ) {
        perror( "mmap_device_memory for physical address failed" );
        exit( EXIT_FAILURE );
    }

    for ( i=0; i < len; i++ ) {
        c =(*(ptr+i) & 0xff);
        if ( isprint(c) )
            fprintf(stderr, "%c", c );
        fprintf(stderr, "[%x] ", c );
        if ( ( i % 20 ) == 0 )
            fprintf(stderr,"\n");
    }
}
```

For example, to print the first 100 bytes at address 0x600000 (to look for the “imagefs” signature), type:

```
dumpmem 0x600000 100
```

See also

`mkifs`, `mkimage`, and `mkrec` in the *Utilities Reference*

OS Images in *Building Embedded Systems*

Chapter 5
POSIX Message Queues: Two Implementations

As described in the "Interprocess Communication (IPC)" chapter of the *System Architecture* guide, QNX Neutrino supports POSIX message queues. The OS includes two managers for message queues:

`mqueue`

The traditional implementation.

`mq`

An alternate implementation.

The `mq` implementation uses a queue within the kernel to buffer the messages and eliminates the (context-switching) overheads of using an external server (i.e., `mqueue`) in each message-queue operation, thus improving performance.

- Although the `mq` implementation provides better performance, it isn't as secure as the traditional implementation.
- In QNX Neutrino 7.0 or later, `mq` and `mqueue` can coexist, but we recommend that you use only one of them on your system.

When you create a queue with `mq`, it appears in the pathname space under **/dev/mq**. Note that it's different from the pathname space for `mqueue`; the queue appears in the pathname space under **/dev/mqueue**. (You can change this directory to union over the directory exported by the `mqueue` server by using the `mq -N/dev/mqueue` option, but we don't recommend this, because it may cause some user-namespace confusion.)

Although the `mq` server isn't involved in each *mq_send()*/*mq_receive()*/*mq_notify()* operation, the server is necessary to maintain the queue names and create the corresponding kernel message queues. You can also use `ls` and `rm` for administrative purposes, but you can't manipulate the queue contents by using shell utilities.

The client functions communicate with `mqueue` or `mq`:

mq_open()

Open or create a queue

mq_close()

Close a queue

mq_unlink()

Remove a message queue

mq_getattr()

Get attributes on a queue

mq_setattr()

Set attributes on a queue

mq_notify()

Request notification when a message arrives

mq_send()

Send a message

mq_receive()

Receive a message

For more information about these functions, see the QNX Neutrino *C Library Reference*. The implementation of the *mq_*()* routines in **libc** is the traditional style, using the `mqueue` server to broker each transaction. In order to use the `mq` implementation, you must link your application(s) against the **libmq** library. In a manual build, specify the `-l mq` option; in automatic/recursive builds, use this setting in your **common.mk** file:

```
LIBS += mq
```

When relinking applications to use the alternative implementation, be sure to change *all* affected components. We require such explicit intervention to use the alternate implementation because of potential incompatibilities if your code isn't strictly conforming to POSIX.

Here are some other differences between these two servers:

- Message queue descriptors (`mqd_t`) aren't file descriptors in the alternate implementation. The POSIX 1003.1 standard makes no claim as to the underlying type of an `mqd_t`, and provides a specific set of functions to manipulate a message queue and its attributes based on an abstract `mqd_t` type.

  For example, you should use *mq_setattr()* instead of *fcntl()* to modify the nonblocking attribute of an open message queue. See the following code:

  ```
  mq_getattr(mq, &attr);
  attr.mq_flags |= O_NONBLOCK;
  mq_setattr(mq, &attr, NULL);
  ```

  The following code attempts to obtain the number of messages in a queue (don't use *fstat()*):

  ```
  mq_getattr(mq, &attr);
  nmsg = attr.mq_curmsgs;
  ```

- Historically, you could interchange *mq_send()* with *write()*, and *mq_receive()* with *read()*. This isn't possible with the alternate implementation.
- In the alternate implementation, there's no direct counterpart to *select()* when used on a mixed set of queues and file descriptors or for the transition from a full queue, although you can use *mq_notify()* to register an event against a message queue on transition from empty.
- The alternate implementation doesn't support the DCMD_MISC_MQSETCLOSEMSG Neutrino extension to inject a canned message when a queue descriptor is closed.

- In the traditional implementation, each call to *mq_open()* uses one file descriptor; in the alternate implementation, each call to *mq_open()* uses two.
- The default configuration of a message queue created by `mq_open(O_CREAT)` with NULL for the *mq_attr* argument is different in the alternate implementation, because kernel buffers for the messages are created up front rather than on demand. To create a queue with a specific configuration, simply provide a non-NULL *mq_attr* argument. To force the NULL default to match that of the traditional implementation, use the `mq -m1024 -s4096` options (which we don't recommend, because this consumes 4 MB of system memory for each such queue).
- The alternate implementation allows for message queues to be manipulated only from the local machine.
- In QNX Neutrino 7.0 or later, `mq` registers for files of type _FTYPE_MQ, and `mqueue` registers for files of type _FTYPE_MQUEUE.

The following table summarizes the main differences between the implementations:

Style	Server	Library	Pathname	`mqd_t` implementation	Qnet	File type
Traditional	`mqueue`	**libc**	**/dev/mqueue/**	`int` (file descriptor)	Yes	_FTYPE_MQUEUE
Alternative	`mq`	**libmq**	**/dev/mq/**	`int` (internal index)	No	_FTYPE_MQ

Chapter 6
Choosing the Correct MTD Routine for the Flash Filesystem

This technote explains how to choose the correct Memory Technology Driver (MTD) routine for your flash filesystem. Before you begin, you should first review the descriptions of all available MTD callouts for a flash driver.

The MTD callouts are as follows:

Callout	Description
ident()	Identify the flash chip(s)
reset()	Return the flash to the read state after an error
read()	Read data from flash (if NULL, an internal *memcpy()* is used)
write()	Write data to flash
erase()	Erase a flash block (also known as a "unit")
sync()	Poll for the completion of an erasure
suspend()	Suspend an erasure for a read/write operation (if supported)
resume()	Resume a suspended erasure after a read/write operation (if supported)
islock()	Determine whether a block/unit is write protected
lock()	Write-protect a block
unlock()	Disable write protection
unlockall()	Invoke a single command to unlock the whole chip (if supported)

Although we refer to the callouts in this technote using the general notation above, the source code for the MTD libraries contains different API versions. The exact callouts available for the two versions of the library (MTDv1 and MTDv2) are:

Generic callout	MTDv1	MTDv2
ident()	*f3s_ident()*	*f3s_ident()*
reset()	*f3s_reset()*	*f3s_reset()*
read()	*f3s_read()*	*f3s_v2read()*

Generic callout	MTDv1	MTDv2
write()	*f3s_write()*	*f3s_v2write()*
erase()	*f3s_erase()*	*f3s_v2erase()*
sync()	*f3s_sync()*	*f3s_v2sync()*
suspend()	*f3s_suspend()*	*f3s_v2suspend()*
resume()	*f3s_resume()*	*f3s_v2resume()*
islock()	—	*f3s_v2islock()*
lock()	—	*f3s_v2lock()*
unlock()	—	*f3s_v2unlock()*
unlockall()	—	*f3s_v2unlockall()*

Ideally, there would be a table that describes every combination of board and flash in existence and tells you which callout to use. Since this is unrealistic, this technote focuses mainly on describing a general method for choosing the right combination of MTD callouts. You will need to:

- have a general understanding of the board's flash interface
- look at the descriptions/comments in the MTD source code in our BSPs
- look at the board's datasheet for the flash part

Unusual flash configurations

As a first step, you should determine whether your board has an unusual flash configuration. In a standard flash configuration, the flash should be located at a contiguous physical memory address that can be directly mapped, read, and written to by software. Usually, the chips are allowed to be interleaved for performance/bus-width reasons.

Here are some examples of nonstandard flash configurations:

Special alignment restrictions

Some CPUs and/or boards require that the memory be read or written to at certain sizes and alignments. These boards cause unpredictable behavior (corruption, SIGBUS, or SIGSEGV) if the default *memcpy()*-based MTD read function is used.

Miswired endian

Some boards have improperly wired flash that cause bytes to be in the wrong endian. This requires significant custom modifications to many MTD callouts in order to write commands and read CFI data in the right endian.

PCMCIA

Under rare situations, some NOR flash-based PC Card storage devices are supported. These require special interactions with the PCMCIA driver.

SDRAM controller based

Most Micron flash chips use an interface based on SDRAM bus commands. While they can be made to work, special code must be written to access the board's memory controller.

NAND flash

CompactFlash, SD Card, SmartMedia, Sony MemoryStick, etc. These devices use NAND flash and are not supported by the `devf-*` flash drivers. NAND flash is totally different from NOR flash, but is often confused with the NOR flash.

SRAM

The `devf-ram` driver may work with some battery-backed SRAM. They are to be assessed on a case-by-case basis. Keep reading if you want to give it a try, but we can't guarantee the driver's resilience to power failure. This is because SRAM has a completely different "failure mode" from that of NOR flash.

If your board has any such unusual configuration, don't hesitate to consult your QNX Software Systems representative.

MTD source code

You can find the complete source to the MTD driver library under *bsp_root*`/libs/src/hardware/flash/mtd-flash`.

The callouts are grouped by manufacturer (e.g., `intel/iCFI_write.c`). Each of these files contains enough comments to detail the type of flash they work with. These comment-blocks provide the most up-to-date information on selecting the right MTD callout. Don't be afraid to look at these files.

Manufacturer

Since the MTD source code is organized by manufacturer, your next step is to determine the manufacturer. Look at the board itself or read the flash's datasheet. Flash components made by different manufacturers are usually not compatible with each other.

To date, we support flash from these manufacturers: Intel and FASL LLC (Spansion/AMD/Fujitsu).

There are also groups for SRAM and ROM; in this case there's only one choice for each callout, so we won't discuss them here.

Choosing the callouts

read()

In most cases, setting the callout pointer to NULL is sufficient. This causes the MTD to use *memcpy()* to read directly from flash. You need to write a custom read callout if your board has special read restrictions.

Here's an example:

```
#include <sys/f3s_mtd.h>

int32_t f3s_mtd_read(f3s_dbase_t *dbase,
                     f3s_access_t *access,
                     uint32_t flags,
                     uint32_t offset,
                     int32_t size,
                     uint8_t *buffer)
{
   uint8_t *memory;
   /* Set proper page on socket */
   memory = (uint8_t *)access->service->page(&access->socket,
                                             F3S_POWER_ALL,
                                             offset,
                                             NULL);
   if (memory == NULL)
   {
       fprintf(stderr,
               "%s: %d page() returns NULL\n",
               __func__,
               __LINE__);
       return (-1);
   }
   /* Replace this memcpy with your special handling code */
   memcpy(buffer, memory, size);
   return (size);
}
```

ident()

There are currently two main choices available for *ident()* callouts: one for CFI and another for non-CFI. Most modern flash chips support the common flash interface (CFI) specification, a common method of identifying a flash chip and its capabilities. If your flash chip supports CFI, then you should always use the CFI-specific *ident()* callout. The alternative is to use a hard-coded table of recognized flash IDs. These non-CFI-specific *ident()* callouts usually require some customization and should be used as a last resort.

write()

Flash can be written to either a word (8 or 16 bits) or a buffer (several words at a time). All chips support single-word writes, so it is always a conservative choice. More sophisticated chips such as Intel StrataFlash and AMD MirrorBit support buffered writes, which are significantly faster.

When consulting AMD's datasheets, don't confuse their “Unlock Bypass” write mode with their buffered write mode. The unlock bypass mode eliminates only the extra handshake between each word write. Real buffered write modes collect several words into an internal buffer and programs them in parallel.

erase()

Choosing an *erase()* callout is very straightforward; there's usually just one to choose from.

sync()

For MTDv1, there are usually two main types of *f3s_sync()* callouts: one for boot-block flash and one for regular flash. The boot-block flash have two different sizes for erase blocks. The boot-block *f3s_sync()* callout needs to know the size of the block it wants to erase. The regular *f3s_sync()* callout doesn't need this extra logic. If you aren't sure, pick the *f3s_sync()* for boot-block flash; it's slightly slower, but works for any kind of flash.

If you need to use a *f3s_v2sync()* callout, there's usually only one to choose from.

The *f3s_v2sync()* callout doesn't need to know the erase block size. The size is determined by the filesystem via a different API.

suspend() and *resume()*

Flash chips either support *suspend()* and *resume()* callouts or they don't. This is evident from the datasheet. Like the *erase()* callout, there's usually only one choice.

islock(), *lock()*, *unlock()*, and *unlockall()*

Locking is new to the MTD version 2 library (MTDv2). Support for locking is evident from the datasheet. For chips that do support block-level write protection, there are two different implementations: persistent and volatile.

- Persistent write protection uses flash cells to remember which block is locked. Just like normal flash cells, these cells can be arbitrarily locked, but must be unlocked at the same time. Persistent implementations use the *islock()*, *lock()*, and *unlockall()* callouts.
- Volatile implementations always default to being completely locked after every reset. This implementation has the advantage that blocks can be locked and unlocked at will. These flash chips use the *islock()*, *lock()*, and *unlock()* callouts.

Chapter 7 Reading a Kernel Dump

If your application crashes with a kernel fault, the output tells you what happened at the time of the crash. Here's a sample:

```
Shutdown[0,0] S/C/F=11/1/11 C/D=f001517d/f00571ac state(c0)= now lock
QNX Version 6.6.0 Release 2014/02/22-18:29:37EST KSB:fe3f6000
[0]PID-TID= 1-1? P/T FL=00019001/08800000 "proc/boot/procnto-instr"
[0]ASPACE PID=7 PF=00001010 "proc/boot/devb-eide"
x86 context[efffcc28]:
0000: 08088cc8 b0359320 efff2c3c efffcc48 b0357f14 08088d10 efff2c10 000000f8
0020: b0323948 0000001d 00011296 efff2c24 00000099
instruction[b0323948]:
ff 08 75 0e 8b 02 83 c4 f4 83 c0 08 50 e8 8e f5 fe ff 8b 5d e8 c9 c3 90
55 89
stack[efff2c24]:
0000:>b0357f14 00000003 08088cc8 b0317d3d b0357f14 b0359320 efff2c6c b033f692
0000: 8088d10 b033f49c efff2c5c b033f678 b0357f14 00000003 00100102 00000003
```

Here's what each part means:

S/C/F=11/1/11

Signal, code, and fault codes; see these files:

- signal: **/usr/include/signal.h**
- code: **/usr/include/sys/siginfo.h**
- fault: **/usr/include/sys/fault.h**

To find out what happened, search **signal.h** for the signal code. This tells you the name of the signal. Then, look in **siginfo.h** for the signal name. In this example, code 11 in **signal.h** is a SIGSEGV; in **siginfo.h**, code 1 in the SIGSEGV section is:

```
SEGV_MAPERR 1  // Address not mapped
```

See also the entry for `siginfo_t` in the QNX Neutrino *C Library Reference*.

C/D

Location of the kernel's code and data.

state

The state of the kernel:

- `now` — in the kernel
- `lock` — nonpreemptible
- `exit` — leaving kernel

- `specret` — special return processing
- any number — the interrupt nesting level.

`QNX Version`

The version of the OS, followed by the date and time at which it was built.

`KSB`

The kernel stack base.

`[`*x*`]PID-TID=`*y*`-`*z*

The process ID and thread ID. On CPU *x* (think SMP), process *y* was running thread *z* when the crash occurred.

`P/T FL`

Process and thread flags. The process flags are in the form `_NTO_PF_*`, and the thread flags are in the form `_NTO_TF_*`. For more information, see **<sys/neutrino.h>** and the entry for `pidin` in the *Utilities Reference*.

`[`*x*`]ASPACE PID=`*y*

On CPU *x*, the address space for process *y* was active. This line appears only when the process is different from the one in the `PID-TID` line.

`PF`

The process flags for the `ASPACE PID`. In the sample above, `devb-eide` wasn't running, but its address space was active.

`context`

The register set. You can find the list of registers in **/usr/nto/include/*cpu*/context.h**, where *cpu* is the appropriate CPU-specific directory.

`instruction`

The instruction on which the error occurred.

`stack`

The contents of the stack. The location of the stack pointer is indicated by a greater-than sign (>) in the output; you can use the `-S` option for `procnto` to specify how many bytes of data before and after the stack pointer to include in the dump.

Chapter 8
SPI (Serial Peripheral Interface) Framework

Hardware interface

This is the interface to the code that implements the hardware-specific low-level functionality of an SPI master.

Function table

The `spi_funcs_t` structure is a table of pointers to functions that you can provide for your hardware-specific low-level module. The high-level code calls these functions.

```
typedef struct {
    size_t  size;   /* size of this structure */
    void*   (*init)( void *hdl, char *options );
    void    (*fini)( void *hdl );
    int     (*drvinfo)( void *hdl, spi_drvinfo_t *info );
    int     (*devinfo)( void *hdl, uint32_t device,
                        spi_devinfo_t *info );
    int     (*setcfg)( void *hdl, uint16_t device,
                       spi_cfg_t *cfg );
    void*   (*xfer)( void *hdl, uint32_t device, uint8_t *buf,
                     int *len );
    int     (*dma_xfer)( void *hdl, uint32_t device,
                        spi_dma_paddr_t *paddr, int len );
} spi_funcs_t;
```

There has to be a function table entry in the low-level module, and it has to be named `spi_drv_entry`. High-level code looks for this symbol name to find the function table for the low-level module.

Low-level module handle structure

The `SPIDEV` structure is the handle that the low-level module has to return to the high-level code. You can extend the structure, but `SPIDEV` has to be at the top. This handle is also passed to the low-level driver when the high-level code calls low-level functions.

```
typedef struct _spidev_entry {
    iofunc_attr_t   attr;
    void        *hdl;       /* Pointer to high-level handle */
    void        *lock;      /* Pointer to lock list */
} SPIDEV;
```

init function

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

```
void *init( void *hdl,
            char *options );
```

The arguments are:

hdl

The handle of the high-level code.

options

A pointer to the command-line arguments for the low-level module.

The function must return either a handle, which is the pointer to the `SPIDEV` of the low-level module, or NULL if an error occurred.

fini function

The *fini* function cleans up the low-level 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 of the low-level module that the *init* function returned.

drvinfo function

The *drvinfo* function gets driver information from the low-level module. The prototype for this function is:

```
int drvinfo( void *hdl,
             spi_drvinfo_t *info );
```

The arguments are:

hdl

The handle of the low-level module that the *init* function returned.

info

A pointer to the driver information structure. This structure is defined as:

```
typedef struct {
        uint32_t    version;
        char        name[16];   /* Driver name */
        uint32_t    feature;
#define SPI_FEATURE_DMA         (1 << 31)
#define SPI_FEATURE_DMA_ALIGN   0xFF
      } spi_drvinfo_t;
```

The function must return EOK if it successfully obtained the driver information.

devinfo function

The *devinfo* function obtains the information from specific devices that are connected to the SPI bus. The prototype for this function is:

```
int devinfo( void *hdl,
             uint32_t device,
             spi_devinfo_t *info );
```

The arguments are:

hdl

The handle of the low-level module that the *init* function returned.

device

The device ID. You can OR it with SPI_DEV_DEFAULT to select the current device; otherwise the next device is selected.

info

A pointer to the device information structure. This structure is defined as:

```
typedef struct {
    uint32_t    device;     /* Device ID */
    char        name[16];   /* Device description */
    spi_cfg_t   cfg;        /* Device configuration */
} spi_devinfo_t;
```

This function must return EOK, or EINVAL if the device ID is invalid.

setcfg function

The *setcfg* function changes the configuration of specific devices on the SPI bus. The prototype for this function is:

```
int setcfg( void *hdl,
            uint16_t device,
            spi_cfg_t *cfg );
```

The arguments are:

hdl

The handle of the low-level module that the *init* function returned.

device

The device ID.

cfg

A pointer to the configuration structure. This structure is defined as:

```
typedef struct {
    uint32_t    mode;
```

```
        uint32_t    clock_rate;
    } spi_cfg_t;
```

This function must return EOK, or EINVAL if either the device ID or the configuration is invalid.

xfer function

The *xfer* function initiates a transmit, receive, or exchange transaction. The prototype for this function is:

```
void *xfer( void *hdl,
            uint32_t device,
            uint8_t *buf,
            int *len );
```

The arguments are:

hdl

The handle of the low-level module that the *init* function returned.

device

The device ID.

buf

A pointer to the data buffer for this transaction.

len

A pointer to length, in bytes, of the data for this transaction.

The function must return a pointer to the receive/exchange buffer, and store, in the location that *len* points to, the byte length of the data that has been transmitted, received, or exchanged by the low-level module. The high-level code checks the length to determine whether the transaction was successful.

The buffer is not DMA-safe, so if the low-level module needs to use DMA, it must allocate its own DMA-safe buffer and copy the data over, if necessary.

dma_xfer function

The *dma_xfer* function initiates a DMA transmit, receive, or exchange transaction. The prototype for this function is:

```
int dma_xfer( void *hdl,
              uint32_t device,
              spi_dma_paddr_t *paddr,
              int len );
```

The arguments are:

hdl

The handle of the low-level module that the *init* function returned.

device

The device ID.

paddr

A pointer to the DMA buffer address, which is defined as:

```
typedef struct {
    uint64_t    rpaddr;
    uint64_t    wpaddr;
} spi_dma_paddr_t;
```

The *rpaddr* and *wpaddr* are physical addresses.

len

The length, in bytes, of the data for this DMA transaction.

This function must return the number of bytes that have been successfully transferred by DMA, or -1 if an error occurred. It's the responsibility of the application to manage the DMA buffer.

API library

The **libspi-master** library provides an interface to mediate access to the SPI master. The resource manager layer registers a device name (usually **/dev/spi0**). Applications access the SPI master by using the functions declared in **<hw/spi-master.h>**.

spi_open()

The *spi_open()* function lets the application connect to the SPI resource manager. The prototype for this function is:

```
int spi_open( const char *path );
```

The argument is:

path

The path name of the SPI device, usually **/dev/spi0**.

This function returns a file descriptor, or -1 if the open failed.

spi_close()

The *spi_close()* function disconnects the application from the SPI resource manager. The prototype for this function is:

```
int spi_close( int fd );
```

The argument is:

fd

The file descriptor that the *spi_open()* function returned.

spi_setcfg()

The *spi_setcfg()* function sets the configuration for a specific device on the SPI bus. The prototype for this function is:

```
int spi_setcfg( int fd,
                uint32_t device,
                spi_cfg_t *cfg );
```

The arguments are:

fd

The file descriptor obtained by calling *spi_open()*.

device

The device ID. You can OR it with SPI_DEV_DEFAULT to set the device as the default for this file descriptor.

cfg

A pointer to the configuration structure. This structure is:

```
typedef struct {
    uint32_t    mode;
    uint32_t    clock_rate;
} spi_cfg_t;
```

The possible mode settings are defined in **<hw/spi-master.h>**.

This function returns EOK if the configuration is successful.

spi_getdevinfo()

The *spi_getdevinfo()* function gets the information for a specific device on the SPI bus. The prototype for this function is:

```
int spi_getdevinfo( int fd,
                    uint32_t device,
                    spi_devinfo_t *devinfo );
```

The arguments are:

fd

The file descriptor returned by *spi_open()*.

device

The device you want to get information about:

Specify:	To get information about:
SPI_DEV_ID_NONE	The first device
A device ID ORed with SPI_DEV_DEFAULT	The specified device
A device ID	The next device

devinfo

A pointer to the device information structure:

```
typedef struct {
    uint32_t    device;     /* Device ID */
    char        name[16];   /* Device description */
    spi_cfg_t   cfg;        /* Device configuration */
} spi_devinfo_t;
```

This function returns EOK if the device information is obtained successfully.

spi_getdrvinfo()

The *spi_getdrvinfo()* function gets the driver information for the low-level module. The prototype for this function is:

```
int spi_getdrvinfo( int fd,
                    spi_drvinfo_t *drvinfo );
```

The arguments are:

fd

The file descriptor returned by *spi_open()*.

devinfo

A pointer to device information structure:

```
typedef struct {
    uint32_t    version;
    char        name[16];   /* Driver name */
    uint32_t    feature;
} spi_drvinfo_t;
```

The function returns EOK if the driver information is obtained successfully.

spi_read()

The *spi_read()* function reads data from a specific device on the SPI bus. The prototype for this function is:

```
int spi_read( int fd,
              uint32_t device,
              void *buf,
              int len );
```

The arguments are:

fd

The file descriptor returned by *spi_open()*.

device

The device ID with at most one of the following flags optionally ORed in:

- SPI_DEV_LOCK
- SPI_DEV_UNLOCK

buf

A pointer to the read buffer.

len

The length, in bytes, of the data to be read.

The function returns the number of bytes of data that it successfully read from the device. If an error occurred, the function returns -1 and sets *errno*:

EIO

The read from the device failed, or a hardware error occurred.

EINVAL

The device ID is invalid, or you're trying to unlock a device that isn't locked.

ENOMEM

Insufficient memory.

EPERM

The device is locked by another connection.

An SPI driver typically considers it to be an error if the number of bytes returned by this function isn't the same as the number of bytes it asked the function to read.

spi_write()

The *spi_write()* function writes data to a specific device on the SPI bus. The prototype for this function is:

```
int spi_write( int fd,
               uint32_t device,
               void *buf,
               int len );
```

The arguments are:

fd

The file descriptor returned by *spi_open()*.

device

The device ID with at most one of the following flags optionally ORed in:

- SPI_DEV_LOCK
- SPI_DEV_UNLOCK

buf

A pointer to the write buffer.

len

The length, in bytes, of the data to be written.

The function returns the number of bytes of data that it successfully wrote to the device. If an error occurred, the function returns -1 and sets *errno*:

EIO

The write to the device failed, or a hardware error occurred.

EINVAL

The device ID is invalid, or you're trying to unlock a device that isn't locked.

ENOMEM

Insufficient memory.

EPERM

The device is locked by another connection.

An SPI driver typically considers it to be an error if the number of bytes returned by this function isn't the same as the number of bytes it asked the function to write.

spi_xchange()

The *spi_xchange()* function exchanges data between a specific device and the SPI master. The prototype for this function is:

```
int spi_xchange( int fd,
                 uint32_t device,
                 void *wbuf,
                 void *rbuf,
                 int len );
```

The arguments are:

fd

The file descriptor returned by *spi_open()*.

device

The device ID with at most one of the following flags optionally ORed in:

- SPI_DEV_LOCK
- SPI_DEV_UNLOCK

wbuf

A pointer to the send buffer.

rbuf

A pointer to the receive buffer.

len

The length, in bytes, of the data to be exchanged.

The function returns the number of bytes of data that it successfully exchanged between the device and the SPI master. If an error occurred, the function returns -1 and sets *errno*:

EIO

The write to the device failed, or a hardware error occurred.

EINVAL

The device ID is invalid, or you're trying to unlock a device that isn't locked.

ENOMEM

Insufficient memory.

EPERM

The device is locked by another connection.

An SPI driver typically considers it to be an error if the number of bytes returned by this function isn't the same as the number of bytes it asked the function to exchange.

spi_cmdread()

The *spi_cmdread()* function sends a command to, and then reads data from, a specific device on SPI bus. The prototype for this function is:

```
int spi_cmdread( int fd,
                 uint32_t device,
                 void *cbuf,
                 int16_t clen,
                 void *rbuf,
                 int rlen );
```

The arguments are:

fd

The file descriptor returned by *spi_open()*.

device

The device ID with at most one of the following flags optionally ORed in:

- SPI_DEV_LOCK
- SPI_DEV_UNLOCK

cbuf

A pointer to the command buffer.

clen

The command length, in bytes.

rbuf

A pointer to the receive buffer.

rlen

The read length, in bytes.

The function returns the number of bytes of data that it successfully read. If an error occurred, the function returns -1 and sets *errno*:

EIO

The write to the device failed, or a hardware error occurred.

EINVAL

The device ID is invalid, or you're trying to unlock a device that isn't locked.

ENOMEM

Insufficient memory.

EPERM

The device is locked by another connection.

An SPI driver typically considers it to be an error if the number of bytes returned by this function isn't the same as the number of bytes it asked the function to read.

You can achieve the same results by calling *spi_xchange()*.

spi_dma_xchange()

The *spi_dma_xchange()* function uses DMA to exchange data between the SPI master and an SPI device. The prototype for this function is:

```
int spi_dma_xchange( int fd,
                     uint32_t device,
                     void *wbuf,
                     void *rbuf,
                     int len );
```

The arguments are:

fd

The file descriptor returned by *spi_open()*.

device

The device ID with at most one of the following flags optionally ORed in:

- SPI_DEV_LOCK
- SPI_DEV_UNLOCK

wbuf

A pointer to the send buffer, or NULL if there's no data to send.

rbuf

A pointer to the receive buffer, or NULL if there's no data to receive.

len

The exchange length, in bytes.

This function calls *spi_dma_xfer()* to do the actual DMA. If you're using the same buffer repeatedly, it's more efficient to call *spi_dma_xfer()* directly.

The *spi_dma_xchange()* function returns the number of bytes of data that it successfully exchanged. If an error occurred, the function returns -1 and sets *errno*:

EIO

The write to the device failed, or a hardware error occurred.

EINVAL

The device ID is invalid, or you're trying to unlock a device that isn't locked, or the buffer address is invalid.

ENOMEM

Insufficient memory.

EPERM

The device is locked by another connection.

ENOTSUP

DMA isn't supported.

An SPI driver typically considers it to be an error if the number of bytes returned by this function isn't the same as the number of bytes it asked the function to exchange.

The application is responsible for allocating and managing the DMA buffer. The application can call *spi_getdrvinfo()* to determine if the driver supports DMA, and whether or not the DMA buffer requires alignment.

spi_dma_xfer()

The *spi_dma_xfer()* function uses DMA to exchange data between the SPI master and an SPI device. The prototype for this function is:

```
int spi_dma_xfer( int fd,
                  uint32_t device,
                  void *paddr,
                  int len);
```

The arguments are:

fd

The file descriptor returned by *spi_open()*.

device

The device ID with at most one of the following flags optionally ORed in:

- SPI_DEV_LOCK
- SPI_DEV_UNLOCK

paddr

A pointer to the DMA buffer address, which is defined as:

```
typedef struct {
    uint64_t    rpaddr;
    uint64_t    wpaddr;
} spi_dma_paddr_t;
```

The *rpaddr* and *wpaddr* are physical addresses.

The *spi_dma_xfer()* function returns the number of bytes of data that it successfully exchanged. If an error occurred, the function returns -1 and sets *errno*:

EIO

The write to the device failed, or a hardware error occurred.

ENOMEM

Insufficient memory.

EPERM

The device is locked by another connection.

ENOTSUP

DMA isn't supported.

An SPI driver typically considers it to be an error if the number of bytes returned by this function isn't the same as the number of bytes it asked the function to exchange.

The application is responsible for allocating and managing the DMA buffer. The application can call *spi_getdrvinfo()* to determine if the driver supports DMA, and whether or not the DMA buffer requires alignment.

Chapter 9
Fine-tuning your network drivers

This technical note is intended to help you tune your network drivers for increased performance or reduced memory footprint.

First, we need to talk about network driver interface hardware chips—ASICs—which are sometimes referred to as *NICs* (network interface controllers, or network interface chips).

At the risk of oversimplifying, we can categorize NICs into two groups: high-performance and low-performance. We aren't talking about the media bit rate (10, 100 or 1000 Mbit) but rather the ability of the complete system, when using the NIC, to avoid packet loss.

Hardware engineers work very hard to design media that rarely lose a packet. And there is a gain, or amplifying effect: when you lose 1% of your packets, you don't lose 1% of your throughput via a protocol, you lose around 50% of your throughput, due to the incurred software-level protocol timeouts and/or retransmissions.

High-performance NICs

What we call “high-performance” NICs have the ability, in a loaded system, to not lose any packets. They generally do this by using transmit and receive descriptor rings in main memory, which in turn point to packet buffers also in main memory.

High-performance NICs use bus-master DMA (direct memory access) to transfer packet data to and from main memory entirely independent of the CPU, using the descriptor rings as laundry lists of packet transmit and receive requests to carry out.

Thus, large scheduling latencies in software that service the NIC (e.g., `io-pkt*`) can be tolerated.

Low-performance NICs

What we call “low-performance” NICs have been observed by users, in loaded systems, to consistently lose packets, with corresponding poor data throughput performance. These NICs *don't* use descriptor rings and DMA, but for simplicity, instead attempt to buffer the entire packet in a (usually limited) on-chip buffer area.

Unfortunately, these low-performance NICs, because of their low cost and size, are very attractive to board designers.

On a fast (e.g., 2 GHz) lightly loaded machine, these low-performance NICs can function adequately, without packet loss.

However on a slower (e.g., 100 MHz) machine that's CPU-bound with applications that may increase the scheduling latency of `io-pkt*`, packet loss during receive can often result because the limited hardware buffer overflows.

You shouldn't use these NICs where you need high-performance data throughput. You should use them only for low-cost debug and diagnostic ports, which are often removed for production versions of boards.

If you're using NFS used with one of these low-performance NICs, you can get a great improvement by using the `-B4096` or even `-B2048` option to `fs-nfs`. Qnet in QNX Neutrino 6.3 and later generally automatically goes into “windowed mode” with these NICs to try to avoid packet loss.

Tuning high-performance NIC drivers

High-performance NICs are all at least 10/100 Mbit, and some are gigabit, but what makes them high-performance is their ability to function independently of the CPU and use the large CPU main memory for packet buffering, which the low performance NICs by design can't.

There are two critical data-transfer interfaces to a high-performance NIC, which you must tune correctly to avoid packet loss under load:

- The first is the (usually PCI) bus itself. The high-performance NIC will have some FIFO memory, used to buffer data for immediate transmit and receive. As the FIFO drains for transmit, or fills up for receive, the NIC must request to become the bus master to burst data to the FIFO for transmit, or from the FIFO for receive.

  If the latency to schedule the NIC as bus master is excessive, the FIFO will drain for transmit or will overflow for receive. Either will cause a packet to be lost.

  Excessive bus master scheduling latency used to be more of a problem in QNX 4, where other devices (e.g., disk) were programmed with excessive DMA burst length; they would “park” themselves on the bus. This doesn't appear to be as much of a problem in QNX Neutrino, but you should be aware that it can be a problem if you're suffering from mysterious packet loss. The `nicinfo` output can often give you a clue here.

- Far more likely, when you're encountering packet loss at the driver/hardware level, is that the transmit and receive descriptor rings are overflowing.

  For receive, this usually happens when a high (e.g. greater than 21) priority thread runs READY and hogs the CPU for an extended period of time. This causes `io-pkt*` to not be scheduled, and the receive descriptor eventually fills up as packets arrive, and the NIC bus-masters the received packets into main memory.

  For transmit, this usually happens when there's an extremely large burst of transmit activity (e.g. server) and possibly some kind of backup or congestion (e.g. PAUSE frames) which simply fills up the transmit descriptor ring faster than the NIC can get it out onto the wire.

  The drivers for the high-performance NICs are generally configured with a default 64 transmit descriptors and 128 receive descriptors. You can change them using the `transmit=`*XXXX* and `receive=`*XXXX* command-line options to the drivers. Generally, the minimum allowed is 16, and the maximum is 2048. Due to the hardware design, stick to a power of 2, such as 16, 32, 64, 128, 256, 512, 1024, or 2048.

  Transmit buffer descriptors are generally quite small, generally in the range of 8 to 64 bytes. So, the cost of increasing the `transmit=`*XXXX* value to, say, 1024 for a server (which sees large bursts of transmitted data) is quite small:

  (1024 - 64) x 32 = 30,720 bytes

  for a transmit descriptor of 32 bytes.

  Receive buffer descriptors are similarly quite small, however there's a catch. For each receive descriptor, the driver must allocate a 1,500 byte Ethernet packet buffer. Because the packet buffers must be aligned, they aren't permitted to cross a 4 KB page boundary, so in reality, `io-pkt*` allocates a 2 KB buffer for each 1,500 byte Ethernet packet.

So, the cost of increasing the receive descriptor to 1024 from the default 128, with an almost insignificant 32-byte-sized receive descriptor is:

(1024 - 128) x (32 + 2048) = 1,863,680 bytes

or almost 2 Megabytes, which is nowhere near as much as the filesystem grabs by default for its cache, but still not an insignificant amount of memory for a memory-constrained embedded system.

For a memory-constrained system, you should carefully select the sizes of the transmit and receive descriptor rings so that they're minimum-sized, yet no packets are lost under load, with the scheduling latency for `io-pkt*` on your system.

Obviously, reducing the receive descriptor ring has more of an effect than reducing the transmit descriptor ring.

In an application where memory is of no concern, but maximum performance is, generally transmit and receive descriptor rings of 1024 or even 2048 are used.

Most of the time, bigger is better. There is, however, a potential catch: for some benchmarks, such as RFC2544 (fast forwarding), we've observed that excessively large descriptor rings decrease performance because of cache thrashing.

However, that's really getting out there. Most of the time, you simply need to configure the transmit and receive descriptor ring size to suit your application so that minimum memory is consumed, and no packets are lost.

PHY probing

Almost all of the network drivers have been optimized for performance with respect to PHY probing.

Formerly, network drivers would periodically (e.g., every two or three seconds) communicate via the MII to the PHY chip connected to the NIC, to determine the speed and duplex of the current media connection.

The problem is that, while the PHY is being probed, packet loss can occur. The drivers now contain an optimization to not probe the PHY, as long as there have recently been some packets received. This gives maximum performance for most users.

However, there is a nasty scenario: the NIC is connected to a 100 Mbit full-duplex link. The cable is rapidly unplugged and immediately replugged into a 10 Mbit half-duplex hub, which also has a steady stream of (e.g. broadcasted) received packets. In this scenario, because of the steady stream of received packets, the network driver *won't* probe the PHY, and will still think it's in 100 Mbit full-duplex. This is a problem, because the NIC isn't listening before it transmits; it's still full-duplex, on a half-duplex link. Excessive collisions and out-of-window collisions will result in packet loss.

However, if you leave the cable unplugged for three seconds before plugging it into another hub, the driver probes the PHY and relearns the media parameters and reprograms the NIC with the appropriate duplex.

If you need to rapidly unplug and replug the cable into network boxes with different duplexes, you should specify `probe_phy=1` to the network driver, to force it to always periodically probe the PHY. Packet loss may result during this probing, but you will know that the driver is always in sync with the PHY with respect to the media.

For maximum performance, the default is `probe_phy=0`.

Speed and duplex

Most (but not all) of the drivers support more than one Ethernet speed and duplex. The most common are 10 and 100 Mbit, and half and full duplex, though the newest NICs support 1000 Mbit (gigabit) Ethernet as well.

All of the drivers let you specify `speed=`*XX* and `duplex=`*Z*, where *XX* is 10, 100 or 1000, and *Z* is 0 (zero) for half-duplex, and 1 (one) for full-duplex. Generally most 10 Mbit links are half-duplex (to old hubs or repeaters, which is the original Xerox blue-book Ethernet) and most 100 Mbit links are full-duplex (switches with point-to-point connections). However, for maximum confusion, you will occasionally see 10 Mbit/full-duplex, and 100 Mbit/half-duplex, but not very often.

If you don't specify speed and duplex, the driver attempts to auto-negotiate the speed and duplex to the fastest possible, by an IEEE specification. Most Ethernet hardware produced in the last few years is compliant with the IEEE specification, but there is some older hardware around that isn't.

In the absence of auto-negotiation, the PHY can figure out the speed pretty easily. However, the duplex is another matter. If auto-negotiation isn't supported, the remote device is assumed to be older and thus half-duplex, not full-duplex.

The moral of the story is that 99% of the time you shouldn't specify speed and duplex; the auto-negotiation should automatically figure it out for you. If you run the `nicinfo` utility, it will tell you what the auto-negotiated speed and duplex is, and most of the time, it will be correct.

It's crucial that both devices, at both ends of the link, use the same speed and duplex, otherwise heavy packet loss can occur (see above).

So, you should specify speed and duplex to the network driver *only* if you have older, perhaps broken or nonstandard Ethernet hardware, *and* you manually control *both ends* of the Ethernet link. For example, a managed hub or a cross-over cable.

Chapter 10
Backtraces

The **libbacktrace** library gives you a way to programmatically backtrace a running process from within itself. You can use backtracing for debugging, as well as for diagnostics or logging. Most of the time, you should use `gdb` for debugging.

The backtrace library lets you:

- backtrace the calling thread
- backtrace a thread within the same process
- backtrace a thread in another process
- backtrace C code
- backtrace C++ code

The backtrace library is compatible with QNX Neutrino 6.3.0 SP2 or later.

Note the following:

- Backtracing is a best effort, and may at times be inaccurate due to the nature of backtracing (e.g. optimized code can confuse the backtracer).
- You can't currently backtrace a thread on a remote node (i.e. over Qnet).
- Backtracing a corrupt stack could cause a fatal SIGSEGV.
- The backtrace library needs to trap SIGSEGV and SIGBUS, but `gdb` normally catches these signals. If you're using `gdb` on a program that uses **libbacktrace**, you need to tell `gdb` to pass these signals to the program for correct operation:

  ```
  handle SIGSEGV pass
  handle SIGBUS pass
  ```

The backtrace library is an unsupported feature, due to its fragility. The functionality it offers is available via `gdb` in a safer and more stable way; `gdb` uses debugging information to unwind, while backtrace uses guessing — it decodes instructions/stack/registers to figure out the return address, and this is prone to errors. GDB uses such guessing only as a fallback when symbols aren't available.

Due to multiple `gcc` versions that can be used, and in each of them probable different optimizations, backtrace accuracy is directly affected; it is almost certain the backtrace won't work in many cases.

API

The **libbacktrace** library defines the following data types and variables:

`bt_accessor_t`

An opaque structure that holds the identity of the thread to backtrace.

`bt_memmap_t`

An opaque structure that holds the memory map of a given process. A memory map is made of a list of all of the object files (executable and shared libraries) in process memory, and its text segments' location and size.

`bt_accessor_t` *bt_acc_self*

A preinitialized accessor used to backtrace BT_SELF.

Don't call *bt_init_accessor()* or *bt_release_accessor()* for this global variable.

The library also defines the following functions, described in more detail in the QNX Neutrino *C Library Reference*:

bt_get_backtrace()

Collect a backtrace

bt_init_accessor()

Initialize a backtrace accessor

bt_load_memmap()

Load a memory map associated with a backtrace

bt_release_accessor()

Release an accessor for a backtrace

bt_set_flags()

Set or clear the flags for backtracing

bt_sprn_memmap()

Format the memory map information for a backtrace

bt_sprnf_addrs()

Format the addresses from a backtrace

bt_translate_addrs()

Translate the addresses from a backtrace

bt_unload_memmap()

Unload a memory map associated with a backtrace

In general, here's how you use these functions:

1. Call *bt_init_accessor()* to set up the backtrace.
2. Optionally call *bt_set_flags()* if you want to do a live backtrace. By default, *bt_get_backtrace()* freezes a thread before gathering the backtrace.
3. Call *bt_get_backtrace()* to collect the backtrace addresses.
4. Optionally load the memory map for the process by calling *bt_load_memmap()*. You need to do this if you want to format the information in certain ways.
5. Optionally call *bt_sprn_memmap()* to produce a string of the memory map's contents.
6. Call *bt_sprnf_addrs()* or *bt_translate_addrs()* to format the backtrace addresses.
7. Call *bt_unload_memmap()* to unload the memory map.
8. Call *bt_release_accessor()* to release the accessor.

Examples

Let's look at some examples of using the backtrace library.

Obtaining and printing a memory map

Use the following sample code segment to obtain and then print the contents of a memory map:

```
#include <stdio.h>
#include <backtrace.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    char out[1024];

    bt_accessor_t acc;
    bt_memmap_t memmap;

    if (bt_init_accessor(&acc, BT_SELF) == -1)
    {
        fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
                 "bt_init_accessor", errno, strerror(errno));
        return EXIT_FAILURE;
    }

    if (bt_load_memmap( &acc, &memmap) == -1)
    {
        fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
                 "bt_load_memmap", errno, strerror(errno));
        return EXIT_FAILURE;
    }

    if (bt_sprn_memmap(&memmap, out, sizeof(out)) == -1)
    {
        fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
                 "bt_sprn_memmap", errno, strerror(errno));
        return EXIT_FAILURE;
    }

    /* Make sure that the string is null-terminated. */
    out[sizeof(out) - 1] = '\0';
    puts(out);

    bt_unload_memmap( &memmap );

    if (bt_release_accessor(&acc) == -1)
    {
        fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
                 "bt_release_accessor", errno, strerror(errno));
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Additional notes about memory:

- The formula for calculating memory used by a memory map is roughly the following:

  ```
  (16 + strlen(filename)) * num_files
  ```

- Some memory is temporarily allocated while the memory map is read. For example, an executable with three shared libraries loaded (assuming an average file name length of 40 characters, and excluding overhead from *malloc()*), would consume 224 bytes.

There are no explicit links between the memory map and a backtrace. Consequently, you're responsible for ensuring that the memory map is reread to account for the proper handling of the removal of the *dlopen()* and *dlclose()* processes, as well as the recycling of process IDs.

Backtracing a thread in another process

Use the following sample code to repeatedly backtrace a thread in another process by using the remote process's pid and thread tid:

```
#include <stdio.h>
#include <backtrace.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    char out[1024];
    bt_addr_t pc[16];
    bt_accessor_t acc;
    int remotepid = -1;
    int remotetid = -1;
    char opt;
    bt_memmap_t memmap;

    while((opt = getopt(argc, argv, "p:t:")) != -1)
    {
        switch(opt) {
        case 'p':
            remotepid = strtol(optarg, 0, 0);
            break;
        case 't':
            remotetid = strtol(optarg, 0, 0);
            break;
        default:
```

```
            exit(EXIT_FAILURE);
        }
    }

    if ((remotepid == -1) || (remotetid == -1)) {
        fprintf(stderr, "Invalid parameters. Use -p <pid> -t <tid>\n");
        exit(EXIT_FAILURE);
    }

    if (bt_init_accessor( &acc, BT_PROCESS, remotepid, remotetid) == -1)
    {
        fprintf(stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
                "bt_init_accessor", errno, strerror(errno));
  return EXIT_FAILURE;
    }

    if (bt_load_memmap( &acc, &memmap ) ==-1)
    {
        fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
                 "bt_load_memmap", errno, strerror(errno));
        return EXIT_FAILURE;
    }

    if (bt_sprn_memmap(&memmap, out, sizeof(out)) == -1)
    {
        fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
                 "bt_sprn_memmap", errno, strerror(errno));
        return EXIT_FAILURE;
    }

    /* Make sure that the string is null-terminated. */
    out[sizeof(out) - 1] = '\0';
    puts (out);

    for (int i=0; i < 10; i++)
    {
        int count = bt_get_backtrace( &acc, pc, sizeof(pc) / sizeof(bt_addr_t));
        int w = 0;
        while (count > 0)
        {
            w = bt_sprnf_addrs(&memmap, pc + w, count, "%a\n", out, sizeof(out), 0);
            if (w == -1)
            {
                fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
                         "bt_sprnf_addrs", errno, strerror(errno));
                return EXIT_FAILURE;
            }
            count -= w;
            puts (out);
        }
    }

    bt_unload_memmap( &memmap );
```

```
        if (bt_release_accessor( &acc ) == -1)
        {
            fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
                     "bt_release_accessor", errno, strerror(errno));
            return EXIT_FAILURE;
        }
    }
```

Backtracing another thread within the same process

Backtracing a different thread in the current process is similar to backtracing a thread in a different process (see above). The only difference is how you initialize the accessor structure. The flags passed to *bt_init_accessor()* define which process to backtrace. If you specify BT_THREAD, a different thread within the same process is set up for backtracing:

```
if (bt_init_accessor(&acc, BT_THREAD, tid) == -1)
{
  fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
           "bt_init_accessor", errno, strerror(errno));
  return -1;
}
```

Backtracing the current thread

To backtrace the currently running process and thread, you can do one of the following:

- Use the BT_PROCESS accessor flag, as described in "*Backtracing a thread in another process*," but specify the current pid and tid.
- Use the BT_THREAD accessor flag and select the currently running thread.
- Use the BT_SELF accessor flag. This is the best way to insure that the currently active thread is backtraced:

  ```
  if (bt_init_accessor(&acc, BT_SELF) == -1)
  {
    fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
             "bt_init_accessor", errno, strerror(errno));
    return -1;
  }
  ```

You can also use a preinitialized accessor called *bt_acc_self*.

Doing a BT_SELF backtrace in a signal handler

Providing backtracing that's signal-handler-safe is a special case. From within a signal handler, you can use only *bt_get_backtrace()*, and you must ensure access to memory is without conflict. For example:

```
bt_accessor_t acc_sighandler1;
bt_addr_t pc_sighandler1[10];
int count_sighandler1;
bt_accessor_t acc;
```

```
bt_memmap_t memmap;

void sighandler1(int sig)
{
    count_sighandler1 =
        bt_get_backtrace(&acc_sighandler1, pc_sighandler1,
           sizeof(pc_sighandler1) / sizeof(bt_addr_t));
}

thread_func()
{
    char out[512];
    if (bt_init_accessor(&acc_sighandler1, BT_SELF) == -1)
    {
      fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
               "bt_init_accessor", errno, strerror(errno));
      ...
    }

    signal(SIGUSR2, sighandler1);
    ...
    if (bt_sprnf_addrs( &memmap, pc_sighandler1, count_sighandler1, "%a",
                        out, sizeof(out), 0) == -1)
    {
       fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
               "bt_sprnf_addrs", errno, strerror(errno));
       ...
    }
    ...

    if (bt_release_accessor(&acc) == -1)
    {
       fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
               "bt_release_accessor", errno, strerror(errno));
       ...
    }
    ...
}
```

Backtracing a collection of threads

Occasionally, it may be useful to backtrace a collection of threads in a coherent manner. You can accomplish this by freezing all the threads, backtracing the threads, and then unfreezing them. For example:

```
bt_accessor_t acc;
int count;

hold_all_threads();

for (i=0; i < max_thread; i++) {
    if (bt_init_accessor(&acc, BT_PROCESS, pid, i) == -1)
    {
      fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
               "bt_init_accessor", errno, strerror(errno));
      return -1;
    }

    if (bt_set_flags(&acc, BTF_LIVE_BACKTRACE, 1) == -1)
    {
      fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
               "bt_set_flags", errno, strerror(errno));
      return -1;
    }

    count = bt_get_backtrace(&acc, addrs, len);
    ...

    if (bt_release_accessor(&acc) == -1)
    {
      fprintf( stderr, "%s:%i %s (%i)%s\n", __FUNCTION__, __LINE__,
               "bt_release_accessor", errno, strerror(errno));
      return -1;
    }

    save_addrs(addrs);
}

cont_all_threads();
```

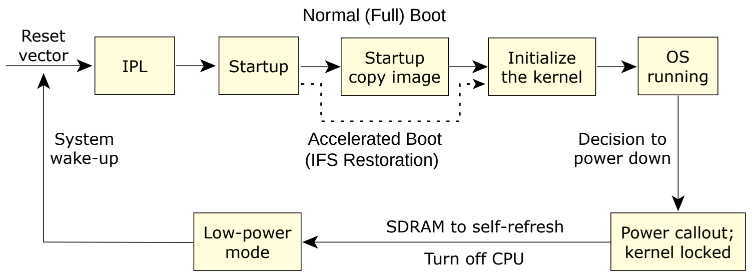
Chapter 11
Reloadable Image Filesystems

The IFS Restoration feature applies to systems that require an ultra-low power mode in which the main CPU is completely turned off while the SDRAM is left in a self-refresh state. This provides the ability to create a low-power mode with power consumption ranging from 1-3 mA (depending on the power draw of the SDRAM in self-refresh) with an accelerated wake-up/reboot.

IFS Restoration makes use of the fact that the image filesystem (IFS) is already in RAM on wake-up from this state and avoids the slow process of copying the image from FLASH to RAM. Note that when booting for the first time (SDRAM turned off), there are no time savings—savings are only on subsequent boots:

- On initial power-up, the system performs a full boot.
- On subsequent boots in which the SDRAM was left in self-refresh, the system performs an accelerated boot via IFS restoration. A full boot is still performed; only the copying of the image from flash to RAM is avoided.
- The boot time saved depends on the size of the image in flash; the larger the image, the more time is saved.
- The CPU and OS context aren't saved or restored. Although this would be faster, it requires device-driver-, CPU-, and board-specific support. The IFS Restoration feature is generic and doesn't require platform-specific development (other than the code to put the SDRAM to self-refresh).

The following diagram shows the difference between a normal boot and an accelerated boot using IFS Restoration:



There are two mechanisms to achieve the IFS restoration:

Kernel restoration

The OS is booted using a single image including QNX Neutrino startup, kernel, and the IFS (containing libraries, drivers, and applications). On wake-up, all of the startup and only part of the kernel (the data section of the `procnto` binary) are copied to RAM. The remainder of the IFS is reused without having to copy.

Secondary IFS restoration

The OS is booted using two images: a primary IFS containing only the QNX Neutrino startup and kernel, and a secondary IFS containing all other libraries, drivers, and applications. The secondary IFS is automatically mounted.

On wake-up, the entire primary IFS is copied, while the secondary IFS can be reused without having to copy. Note that additional binaries can be added to the primary IFS, but these will be copied along with the rest of the primary IFS.

The advantage of kernel restoration is that it can be enabled and used with very little intervention by the user. Secondary IFS restoration may be desired for projects that wish to use multiple image filesystems, but requires more management by the user, such as maintaining multiple buildfiles and specifying the secondary IFS size and location.

It's possible to enable both kernel and secondary IFS restoration at the same time (i.e., the startup and kernel from the primary IFS are restored, and the secondary IFS is reused and automatically mounted).

IFS Restoration was designed to support the following features:

- compressed and uncompressed images

  Compressed secondary image filesystems aren't currently supported.

- IFS checksum — both primary and secondary IFSs can be verified using an optimized checksum algorithm (checksum can be enabled or disabled)
- compatibility with minidrivers
- automatic secondary IFS source and destination address selection

Command-line options

The features are enabled via command-line options to the board-specific startup. Since all of the support is built into the startup library, the features are generic across all platforms.

Kernel restoration

To enable kernel restoration, use the `-I` option to the startup in the buildfile:

`startup-`*boardname* `-I` *flag*

where *flag* is 0 to disable checksum verification, or 1 to enable it.

For debugging output, specify at least two levels of debugging via the `-v` option:

`startup-`*boardname* `-I` *flag* `-vv`

If checksum verification is enabled and fails, the entire image is reloaded.

Even if the IFS checksum verification is disabled, a checksum is still performed on the IFS Restoration internal data structure (approximately 32 bytes) to ensure at least some data integrity.

Secondary IFS restoration

To enable secondary IFS restoration, use the `-i` option to the startup in the buildfile:

`startup-`*boardname* `-i` *ifs2_size* `[,` *flags* `] [,` *paddr_src* `] [,` *paddr_dst* `]`

The arguments are:

ifs2_size

The size of the secondary IFS (note: this can be larger than the actual size).

flags

- Not specified — load the secondary IFS but don't try to restore on wake-up
- `R` — load the secondary IFS and restore
- `K` or `RK` — load the secondary IFS and restore with a checksum

paddr_src

- Not specified — the secondary IFS is located in flash after the primary IFS
- Specified — the secondary IFS is located at the physical address specified

paddr_dst

- Not specified — the secondary IFS will be copied to a default location in RAM
- Specified — the secondary IFS will be copied to the physical address specified (choose an address in a “safe” place, such as at the end of RAM away from where the primary image is copied)

For debugging output, specify at least two levels of debugging via the `-v` option:

`startup-`*boardname* `-i` *ifs2_size* `[,` *flags* `][,` *paddr_src* `][,` *paddr_dst* `] -vv`

If the checksum is enabled and fails, the entire secondary IFS is reloaded.

Even if the secondary IFS checksum is disabled, a checksum is still performed on the IFS Restoration internal data structure (approximately 16 bytes) to ensure at least some data integrity.

CAUTION: Kernel and secondary IFS restoration aren't guaranteed to work if the image is downloaded serially. This is because the IPL may copy the serially downloaded image to a location in RAM that overwrites the secondary IFS or data structures used by the restore features.

In practice, this isn't an issue since serial downloading won't be used other than for testing. If serial download is required, try manually setting the destination location of the secondary IFS to be somewhere away from where the IPL downloads the image.

Examples

The following examples are for the EDOSK7780 reference platform, based on an SH4A 400 MHz CPU.

Kernel restoration

The sample buildfile for kernel restoration on the EDOSK7780 reference platform is as follows:

```
#################################################################
## START OF BUILD SCRIPT for Renesas EDOSK7780 Board
#################################################################
[image=0x88010000]
[virtual=shle/binary +compress] .bootstrap = {
    startup-edosk7780 -Dscif..115200.1843200.16 -f400000000 -vv -I1

    PATH=/proc/boot LD_LIBRARY_PATH=/proc/boot procnto -vvvv
}

[+script] .script = {
    procmgr_symlink ../../proc/boot/libc.so.2 /usr/lib/ldqnx.so.2

    display_msg Welcome to QNX Neutrino on the Renesas EDOSK-7780

    devc-sersci -e -F -x -b115200 -c1843200/16 scif0 scif1 &
    reopen /dev/ser1

    SYSNAME=nto
    TERM=qansi

    [+session] PATH=:/proc/boot LD_LIBRARY_PATH=/proc/boot ksh &
}

[type=link] /dev/console=/dev/ser1
[type=link] /tmp=/dev/shmem
libc.so

[data=c]
devc-sersci
ls
ksh
pidin
###################################################################
## END OF BUILD SCRIPT
###################################################################
```

To build the image:

```
# mkifs -v edosk7780.build edosk7780.ifs
```

The sample output is as follows:

```
QNX Neutrino IPL for the EDOSK-7780
Press 'd' to download an OS image serially
Press any other key to boot from flash
```

Press a key other than **d**; the output continues:

```
Downloading from Flash
Restore IFS searching for valid IFS in RAM...

rifs_info PADDR = 0x08008000
rifs_info ADDR = 0x88008000
INVALID IFS signature
Restore IFS failed - Reload entire IFS.

PT_LOAD RW: 0004b000 size is 000030b0
Found procnto Elf header
bootable exec data: offset 0004b000, size 000030b0
Compressed image, store data
Calculate restore info checksum

System page at phys:081d1000 user:081d1000 kern:881d1000
Starting next program at v8803c258
Welcome to QNX Neutrino on the Renesas EDOSK-7780
```

The board has booted; run an application:

```
#
# ls
dev     proc    tmp     usr
```

Press the reset button to simulate wake-up:

```
# QNX Neutrino IPL for the EDOSK-7780
Press 'd' to download an OS image serially
Press any other key to boot from flash
```

Press a key other than **d**. The output continues:

```
Downloading from Flash
Restore IFS searching for valid IFS in RAM...

rifs_info PADDR = 0x08008000
rifs_info ADDR = 0x88008000
FOUND valid IFS signature
FOUND valid RIFS signature
FOUND valid RIFS info
FOUND valid IFS signature and RIFS info in RAM.

IFS pre checksum = 0x7bbb7898 (shouldn't be 0x0)
bootable exec 0 offset: 0x0004bef8
bootable exec 0 size: 0x000030b0
Compressed image src = 0x0800a000
IFS post checksum = 0x00000000 (should be 0x0)
PT_LOAD RW: 0004b000 size is 000030b0
Found procnto Elf header
bootable exec data: offset 0004b000, size 000030b0
Compressed image, store data
Calculate restore info checksum

System page at phys:081d1000 user:081d1000 kern:881d1000
```

```
Starting next program at v8803c258
Welcome to QNX Neutrino on the Renesas EDOSK-7780
# ls
dev     proc    tmp     usr
```

To disable the checksum verification on the IFS after it has been restored, modify the buildfile to:

```
startup-edosk7780 -Dscif..115200.1843200.16 -f400000000 -vv —I0
```

On subsequent boots, you will see the additional output (with debug enabled):

```
WARNING: Skipped image checksum verification
```

With debugging enabled, the checksum value is still calculated so that it can be displayed. However, the checksum value isn't used to determine if the IFS has been restored properly (it's assumed that it was).

If you're working with uncompressed images, the buildfile looks like:

```
[virtual=shle/binary] .bootstrap = {
    startup-edosk7780 -Dscif..115200.1843200.16 -f400000000 -vv -I1

    PATH=/proc/boot LD_LIBRARY_PATH=/proc/boot procnto -vvvv
}
```

The sample output looks something like:

```
Restore IFS searching for valid IFS in RAM...

rifs_info PADDR = 0x08008000
rifs_info ADDR = 0x88008000
INVALID IFS signature
Restore IFS failed - Reload entire IFS.

PT_LOAD RW: 0004b000 size is 000030b0
Found procnto Elf header
bootable exec data: offset 0004b000, size 000030b0
Calculate restore info checksum
```

Press the reset button to simulate wake-up. The output continues:

```
Restore IFS searching for valid IFS in RAM...

rifs_info PADDR = 0x08008000
rifs_info ADDR = 0x88008000
FOUND valid IFS signature
FOUND valid RIFS signature
FOUND valid RIFS info
FOUND valid IFS signature and RIFS info in RAM.

IFS pre checksum = 0x7a82a8f2 (shouldn't be 0x0)
bootable exec 0 offset: 0x0004bef8
bootable exec 0 size: 0x000030b0
Uncompressed image paddr_src = 0x00059000
IFS post checksum = 0x00000000 (should be 0x0)
PT_LOAD RW: 0004b000 size is 000030b0
```

```
Found procnto Elf header
bootable exec data: offset 0004b000, size 000030b0
Calculate restore info checksum
```

Secondary IFS restoration

To build an image with multiple IFSs, you must create a primary and a secondary IFS. The primary IFS includes only the startup and `procnto` (kernel). The secondary IFS includes all other libraries and binaries including **libc**, drivers, and applications.

Here's a sample primary buildfile:

```
#################################################################
## START OF PRIMARY IFS BUILD SCRIPT for Renesas EDOSK7780 Board
#################################################################
[image=0x88010000]
[virtual=shle/binary +compress] .bootstrap = {
startup-edosk7780 -Dscif..115200.1843200.16 -f400000000 -i0x700000 -vv

PATH=/proc/boot:/ifs2 LD_LIBRARY_PATH=/proc/boot:/ifs2 procnto -vvvv
}

[+script] .script = {
# Default libc symbolic link
#procmgr_symlink ../../proc/boot/libc.so.2 /usr/lib/ldqnx.so.2
# Symbolic link to libc in secondary IFS
    procmgr_symlink ../../ifs2/libc.so.2 /usr/lib/ldqnx.so.2

    display_msg Welcome to QNX Neutrino on the Renesas EDOSK-7780
    devc-sersci -e -F -x -b115200 -c1843200/16 scif0 scif1 &
    reopen /dev/ser1

    SYSNAME=nto
    TERM=qansi

    [+session] PATH=:/proc/boot:/ifs2 LD_LIBRARY_PATH=/proc/boot:/ifs2 ksh &
}

[type=link] /dev/console=/dev/ser1
[type=link] /tmp=/dev/shmem
##################################################################
## END OF PRIMARY IFS BUILD SCRIPT
##################################################################
```

The size `0x700000` passed to the `-i` option was chosen to be much larger than the size of the secondary IFS. Don't make it too large, since memory will be reserved according to this size, and a copy will be made from flash.

The sample secondary buildfile is as follows:

```
#################################################################
## START OF SECONDARY IFS BUILD SCRIPT for Renesas EDOSK7780 Board
#################################################################
# Specify the search path, otherwise defaults to x86
```

```
[search=${QNX_TARGET}/shle/bin:${QNX_TARGET}/shle/usr/bin:${QNX_TARGET}/shle/sbin:${QNX_TARGET}/shle/usr/sbin:${Q

# Windows mkifs needs to be reminded of permissions
[perms=+x]

# Where the files will be mounted at boot time
[prefix=/ifs2]

# Libraries to include
[data=copy]
libc.so

# Binaries to include
[+raw data=copy]
cat
devc-sersci
ls
pidin
ksh
echo
###################################################################
## END OF SECONDARY IFS BUILD SCRIPT
###################################################################
```

Unless the physical address for the source location for the secondary image is specified, IFS Restoration will automatically look for the secondary IFS in flash directly following the primary IFS.

To create an image consisting of both primary and secondary IFS:

```
# mkifs -v primary.build primary.ifs
# mkifs -v secondary.build secondary.ifs
# cat primary.ifs secondary.ifs > edosk7780.ifs
# mkflashimage
```

The above example loads a primary IFS and a secondary IFS mounted at **/ifs2**. The output will look something like:

```
Press 'd' to download an OS image serially
Press any other key to boot from flash
```

Press a key other than **d**.

```
Downloading from Flash
ifs2_paddr_dst: 0x0806d000
ifs2_paddr_src: 0x0003a9b0
ifs2_paddr_src (auto): 0x0003a9b0

System page at phys:0800a000 user:0800a000 kern:8800a000
Starting next program at v8803c258
Welcome to QNX Neutrino on the Renesas EDOSK-7780
# ls /proc/boot
.script    procnto

# ls /ifs2
cat     ksh         ls
```

```
echo        libc.so         pidin
devc-sersci     libc.so.2
```

To restore the secondary IFS, modify the options to startup in the primary buildfile (enable secondary IFS restoration, enable checksum):

```
startup-edosk7780 -Dscif..115200.1843200.16 -f400000000 -i0x700000,K -vv
```

After rebuilding and burning the updated multiple IFS image to flash, the output will look like:

```
QNX Neutrino IPL for the EDOSK-7780
Press 'd' to download an OS image serially
Press any other key to boot from flash
```

Press a key other than **d**.

```
Downloading from Flash
ifs2_paddr_dst: 0x0806d000
Restore IFS2 searching for valid IFS in RAM...

rifs2_info PADDR = 0x08008000
rifs2_info ADDR = 0x88008000
INVALID IFS signature
Restore IFS2 failed - Reload entire IFS2.

ifs2_paddr_src: 0x0003a9b0
ifs2_paddr_src (auto): 0x0003a9b0

System page at phys:0800b000 user:0800b000 kern:8800b000
Starting next program at v8803c258
Welcome to QNX Neutrino on the Renesas EDOSK-7780
#
```

Press the reset button to simulate wake-up:

```
# QNX Neutrino IPL for the EDOSK-7780
Press 'd' to download an OS image serially
Press any other key to boot from flash
```

Press a key other than **d**:

```
Downloading from Flash
ifs2_paddr_dst: 0x0806d000
Restore IFS2 searching for valid IFS in RAM...

rifs2_info PADDR = 0x08008000
rifs2_info ADDR = 0x88008000
FOUND valid IFS signature
FOUND valid IFS2 signature and RIFS2 info in RAM.


System page at phys:0800b000 user:0800b000 kern:8800b000
Starting next program at v8803c258
Welcome to QNX Neutrino on the Renesas EDOSK-7780
#
```

To disable the checksum verification on the secondary IFS after it has been restored, modify the buildfile to:

```
startup-edosk7780 -Dscif..115200.1843200.16 -f400000000 -i0x700000,R -vv
```

On subsequent boots, you will see the additional output (with debug enabled):

```
WARNING: Skipped IFS2 checksum verification
```

If you don't want to put the primary and secondary IFS together in flash (by using the `cat` utility), you can put the secondary IFS anywhere in the flash you wish and manually specify its location. Additionally, you can manually specify the location to copy the secondary IFS to in RAM if you don't wish to use the default location. Use the optional arguments to specify the physical address for the secondary IFS source and destination:

- *paddr_src* is the physical location where the secondary IFS is located in flash
- *paddr_dst* is the physical location where the secondary IFS will be copied to in RAM

To calculate the destination in RAM:

paddr_dst = *paddr_start_of_RAM* + *paddr_location_in_RAM*

For example, on the EDOSK7780, there is 128 MB of RAM starting at the address 0x08000000. To put the secondary IFS at the 120 MB location in RAM:

paddr_dst = 0x08000000 + 120 * 1024 * 1024 = 0x0F800000

The destination location in RAM must lie on a 4 KB page boundary. The address passed in is automatically adjusted to ensure this:

paddr_dst = *paddr_dst* & 0xFFFFF000.

To manually specify the source of the secondary IFS located in flash at 0x00040000:

```
startup-edosk7780 -Dscif..115200.1843200.16 -f400000000 -i0x700000,0x00040000
```

To manually specify the source of the secondary IFS located in flash at 0x00040000, and the destination of the secondary IFS in RAM at 0x0F800000:

```
startup-edosk7780 -Dscif..115200.1843200.16 -f400000000 -i0x700000,0x00040000,0x0F800000
```

To manually specify the source of the secondary IFS located in flash at 0x00040000, the destination of the secondary IFS in RAM at 0xF8000000, and enable restoring with checksum:

```
startup-edosk7780 -Dscif..115200.1843200.16 -f400000000 -i0x700000,K,0x00040000,0x0F800000
```

Minidriver support

The IFS Restoration was designed to be compatible with minidrivers. Any copies of data made during IFS restoration will include periodic polls to the minidriver handler (if enabled) based on the value specified by the *mdriver_max* variable:

```
unsigned mdriver_max = KILO(16);
```

This value specifies how many bytes will be copied from flash to RAM before the minidriver handler will be polled. For more information, see the Instant Device Activation *User's Guide*.

Similarly, IFS Restoration will perform the potentially lengthy operation of performing a checksum on the entire IFS (if enabled). To ensure that the minidriver handler is periodically polled, a polling value is defined:

```
unsigned mdriver_cksum_max = KILO(500);
```

This value specifies how many bytes of the IFS checksum will be performed before the minidriver handler will be polled. The default value is set to be much larger than the *mdriver_max* variable since it's assumed that the checksum (adding bytes in RAM) will be much faster than copying bytes from flash to RAM.

Performance measurements

The following table shows the boot time savings using IFS Restoration, as measured on an EDOSK7780 at 400 MHz. Note that debug output is disabled.

Restoration type	Image size	IFS compression	IFS checksum	Initial boot time	Subsequent boot time	Savings
Kernel	32.8 MB	UCL	No	8 s	< 0.5 s	7.5 s
Kernel	32.8 MB	UCL	Yes	8 s	< 1.0 s	7.0 s
Kernel	32.8 MB	None	No	7 s	< 0.5 s	6.5 s
Kernel	32.8 MB	None	Yes	7 s	< 1.0 s	6.0 s
Secondary IFS	32.0 MB (0.2 MB, 31.8 MB)	UCL (primary) None (secondary)	No	7 s	< 0.5 s	6.5 s
Secondary IFS	32.0 MB (0.2 MB, 31.8 MB)	UCL (primary) None (secondary)	Yes	7 s	< 1.0 s	6.0 s

Note the following details regarding these performance numbers:

- Boot times don't include the time from power-on/reset vector but from the point where the IPL scans and loads the image (add 10–50 ms to the times above).
- During initial boot, images are manually copied from flash to RAM (i.e. no DMA).
- Boot times include the time to:
  - copy the image from flash to RAM (on initial boot)
  - run the startup and initialize the kernel
  - start the serial driver and the shell
- Boot times don't include the time to start other drivers or user applications.
- Time measurements are made to approximately 0.5 second granularity (subsequent boot times are likely faster than reported).
- Boot time savings for uncompressed images are less because the initial booting is faster than UCL compressed images (due to the time it takes to uncompress the UCL image).

Manually mounting an IFS

To manually mount a 10 MB IFS located in memory at 0x0F800000:

```
mount -tifs -ooffset=0x0F800000,size=0xA00000 /dev/mem /ifs2
```

Sample script to combine IPL with boot image for the EDOSK7780

```
#!/bin/sh
# script to build a binary IPL and boot image for the EDOSK7780 board
set -v

#convert IPL into an S-Record
${QNX_HOST}/usr/bin/ntosh-objcopy -Osrec ipl-edosk7780 ipl-tmp.srec

#convert S-Record IPL to binary
${QNX_HOST}/usr/bin/ntosh-objcopy -Isrec -Obinary ipl-tmp.srec ipl.tmp
mkrec -r -ffull -s0x2000 ipl.tmp > ipl-tmp.bin

#combine ipl with boot image
cat ipl-tmp.bin edosk7780.ifs > edosk7780.bin
echo "done!!!!!!!"
```

Commands to burn a new IPL/Boot image for the EDOSK7780

```
io-pkt -d abc100 ioport=0x15800000,irq=6 -pttcpip if=en0:x.x.x.x
qconn (use to transfer new image to /dev/shmem)
devf-edosk7780 -s0x0,64M   (Overwrite the boot image in current bank)
flashctl -p/dev/fs0 -l2M -ev
cp -V fb.ifs /dev/fs0
```

To burn the image in the other bank (i.e., not the bank used to boot), use the following instead:

```
devf-edosk7780 -s0x04000000,64M
```

Chapter 12
Customizing language sort orders for `libqdb_cldr.so`

Different locales, organizations and implementations may require different sort orders, even for the same language. This section describes how you can customize sort orders for your MME implementation.

Standard language sort order files

The CLDR (Common Locale Data Repository) tables shipped with the MME use the standard Unicode ordering produced from the CLDR project. These tables are converted from the standard in CLDR POSIX format and shipped as binary files suitable for the **libqdb_cdlr.so**. They are located at **etc/cldr/**:

- **cs_CZ** — Czech
- **da_DK** — Danish
- **de_DE** — German (Germany)
- **en_US** — English (U.S.)
- **es_ES** — Spanish (Spain)
- **fr_FR** — French (France)
- **it_IT** — Italian
- **ja_JP** — Japanese
- **ko_KR** — Korean
- **nb_NO** — Norwegian (Bokmål)
- **sv_SE** — Swedish
- **zh_CN** — Chinese (simplified)

For information about converting CLDR POSIX tables, see the **mkcldr** page in the *Neutrino Utilities Reference*. For information about the CLDR project, see *www.unicode.org/cldr/*.

Sort order algorithm

The language collation (or sorting) DLL, **libqdb_cldr.so**, implements the Unicode Collation Algorithm. This algorithm is a multi-level comparison algorithm, where each character has a number of weights.

These character weights typically correspond to the base character, accents, case, and punctuation. The primary weight has the most relevance, while lower-level weights provide a tie-breaking role; that is, they determine sort order when two or more different elements are assigned the same primary weight.

For example, depending on specific language, locale and even institutional conventions, the characters "a" and "á", and "A" and "Á" might all be assigned the same primary weight, but require differentiation at the secondary and tertiary levels respectively.

For more information about the Unicode Collation Algorithm, see *www.unicode.org/reports/tr10/*.

Contractions and expansions

The Unicode Collation Algorithm supports the character contractions and expansions required for correct sorting in some languages. For example, in traditional Spanish "ch" is considered a single letter that sorts after "c" and before "d", and in German, "æ" is sorted as "a" followed by "e". Correct sorting in these languages requires, respectively, contraction and expansion of the characters being sorted.

Locale data files

The Unicode Common Locale Data Repository includes sort order files for more than 200 locales. Each locale has a table of character weights, and contraction and expansion sequences that describes the sort ordering for that locale.

The filenames for these files take the form *language_LOCALE*.UTF-8.src. For example, the file for French in Canada is **fr_CA.UTF-8.src**.

Adding a new sort order locale

To add a new locale sort order:

1. Download the required CLDR locale file from *cldr.unicode.org*.
2. Convert the downloaded sort order table to the binary data format used by **libqdb_cldr.so**. See “Converting CLDR POSIX files” below.
3. Configure your application to use the new sort order.

Converting CLDR POSIX files

Use the **mkcldr** utility to convert CLDR POSIX files to binary files suitable for the **libqdb_cdlr.so**. For example, the following example converts the file for German used in Switzerland:

```
$ cd cldr-1.4.1/posix
$ mkcldr -c UTF-8.cm de_CH.UTF-8.src /etc/cldr/de_CH
```

The **UTF-8.cm** file is simply a database that maps textual character descriptions to their Unicode value; it is used in parsing the collation information.

Tailoring a sort order algorithm

If none of the standard CLDR locale files defines exactly the sort order you require, you can tailor the character weights in an existing file, which you can then save and convert to use in your custom implementation.

The POSIX format for sort order files is a processed output with explicit weights already assigned to the sort order. It may be simplest, therefore, to tailor a sort order by modifying the XML file from which a sort order file was generated.

That is, to tailor a sort order:

1. Download from *unicode.org/repos/cldr/trunk/docs/web/repository_access.html* and unzip the files for the latest release:
   - the XML/LDML files (**core.zip**)
   - the Java tools for generating POSIX files from the XML files (**tools.zip**)
2. Open the XML file for the sort order you want to change and tailor the character weights as required.
3. Use the **GeneratePOSIX** utility from the Java tools download to generate a POSIX file with the tailored sort order.
4. Use the QNX **mkcldr** utility to convert the new POSIX file to a binary file suitable for the **libqdb_cdlr.so**, as explained above in "*Converting CLDR POSIX files*".

To use your new (*language_LOCALE*.UTF-8.src) file, it must be in the **/etc/cldr** directory or the directory specified by *$QDB_CLDR_PATH*, as required by your system configuration. Ensure, then, that the new file is copied to this location, either as part of the XML to POSIX conversion with **mkcldr**, or by a simple file copy afterwards.