Boot Optimization Guide

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: Tuesday, February 25, 2014

Table of Contents

Preface
About This Guide

The *Boot Optimization Guide* gives an overview of how the QNX CAR platform meets the boot time requirements of automotive systems. The guide describes the sequence of events from the initial power on to a fully functional system, including the optimization actions you can take at different points in this sequence.

To find out about:	See:
An introduction to boot time optimization	*Overview of Boot Optimization* (p. 9)
An overview of the system's boot sequence	*System Startup Sequence* (p. 11)
Preparing your target for boot optimization	*Configuring the Target for Boot Optimization* (p. 15)
Boot optimization in QNX CAR and best practices for optimizing boot times	*Optimizing Boot Times* (p. 17)
Measuring times in the boot sequence	*Measuring Boot Times* (p. 35)

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`
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.

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

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 use a forward slash (/) as a delimiter in all 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
Overview of Boot Optimization

Like other embedded systems, the QNX CAR Platform for Infotainment boots in several stages, each involving a number of interdependent tasks.

These tasks all take time. To ensure that software and hardware components are initialized and ready when needed, the system architect or designer must think deliberately through each of these stages. The QNX CAR platform provides a number of mechanisms to help meet your particular bootup requirements. This document will walk through the entire bootup sequence, offering techniques you can use at each stage to optimize the bootup sequence for the particular requirements of your system.

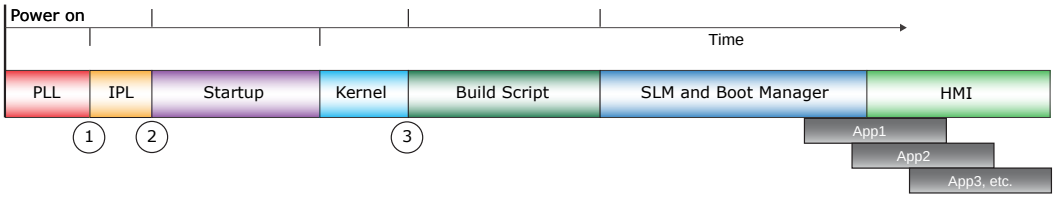
The boot process consists of several stages:

1. The operating system must load from nonvolatile storage.
2. The operating system must initialize itself, as well as any device drivers and peripherals.
3. The HMI (also called the Navigator) must load, initialize, and start running.
4. The application software must load, initialize, and start running.

These stages are discussed in detail in the sections that follow.

Chapter 2
System Startup Sequence

The QNX CAR platform boots in several stages, as illustrated in the following diagram:



The boot stages are as follows:

- **PLL (phase locked loop)**—PLL refers to how long it takes for the first instruction to begin executing after power is applied to the processor. Most CPUs have a PLL that divides the main crystal frequency into all the timers used by the chip. The time that the PLL takes to settle to the desired frequencies often represents the largest portion of the chip's startup time. The PLL stage is independent of any OS and varies from CPU to CPU; in some cases, it takes as long as 32 milliseconds. Consult your CPU user manual for the exact timing.
- **IPL (initial program loader)**—QNX provides a standard, bare-bones IPL that performs the fewest steps necessary to configure the memory controller, initialize the chip selects and/or PCI controller, and configure other required CPU settings. Once these steps are complete, the IPL copies the startup program from the image filesystem (IFS) into RAM and jumps to it to continue execution.

  The IFS contains the OS image, which consists of the startup program, the kernel, the build scripts, and any other drivers, applications, and binaries that the system requires. Because you can control what the IFS contains, the time for the copying stage varies, but it typically constitutes the longest part of the kernel boot process. In extreme cases where the system contains a very large image and has no filesystem other than the IFS, this stage can take a long time (10 seconds or more).

  That said, you can exercise a great deal of control over the length of this phase, albeit indirectly, by reducing the size of the IFS. To add, remove, or configure files stored in the IFS, you can edit the build script or use the system builder tool in the IDE. You can also compress the image to make the IFS smaller (with the additional overhead of decompression, which you can speed up by enabling the cache in the IPL).

  Typically, the bootloader executes for at least 6 milliseconds before it starts to load the OS image. The actual amount of time depends on the CPU architecture, on what the board requires for minimal configuration, and on what the chosen bootloader does before it passes control to the startup program.

Some boards come with another bootloader, such as U-boot. These bootloaders aren't as fast as the QNX IPL, since the IPL has been specifically tuned for QNX systems. We recommend that you replace your bootloader with the IPL.

For more information on the IPL and how to modify it for your purposes, see "Writing an IPL Program" in the *Building Embedded Systems* guide.

- **Startup program**—The first program in a bootable OS image is a startup program whose purpose is to initialize the hardware, initialize the system page, initialize callouts, and then load and transfer control to the kernel (`procnto` or `procnto-smp`). If the OS image isn't in its final destination in RAM, the startup program copies it there and decompresses it, if required.

  During bootup, the kernel initializes the memory management unit (MMU); creates structures to handle paging, processes and exceptions; and enables interrupts. Once this phase is complete, the kernel is fully operational and can begin to load and run user processes from the build scripts.

- **Build scripts**—Each board has a different set of build scripts to support different configurations. The build scripts let you specify which drivers and applications to start, and in what order.

  You can use the build scripts to launch services or utilities that need to be running very early (for example, audio chime and backup camera) or that need extra time to load (for example, PPS or disk drivers). Wherever possible, these processes should be started in the background to optimize parallelsim and maintain the highest possible utilization of the CPU until the HMI is fully operational.

  It's also important to limit what goes into the build script because the build script is included in the IFS, and everything that's added to it increases the IFS size and the time it takes to load. Furthermore, the System Launch and Monitor (SLM) is more efficient at launching services, with the added benefit that it allows you to monitor and restart services as required.

  In the QNX CAR platform, the build scripts start the following:

  - screen and camera services
  - audio service and the early chime utility
  - disk drivers (and then mount the disks)
  - PPS service
  - debugging utilities, such as `slogger` and `dumper`
  - BSP drivers, like the serial driver, realtime clock, and other hardware utilities
  - SLM and the system debug console

- **SLM**—SLM is a service that starts any processes that are necessary for the HMI (`io-pkt`, for example), and then starts the Boot Manager and the HMI itself. At this point, SLM waits for further instructions from the Boot Manager. SLM is controlled by a set of configuration files (`slm-config-all.xml`,

`slm-config-modules.xml`, and `slm-config-platform.xml`) that tell it what modules to start and whether there are dependencies within or between those modules. The dependencies of the HMI are defined in the `car2-init` module of the file `slm-config-all.xml`. For more information, see the entry for SLM in the *System Services Reference*.

- **Boot Manager**—The Boot Manager drives SLM by sending it commands to start up sets of components (modules) that in general comprise the dependencies for each core application (tab) of the HMI, but could also allow you to launch other sets of functionality at a particular point in the boot sequence (for example, Bluetooth services). Each tab in the HMI is defined in the file `slm-config-modules.xml` as the list of dependencies it requires.

  The Boot Manager publishes PPS objects to `/pps/services/bootmgr/modules_ready/` to signal to the HMI that a particular tab's dependencies are ready and that tab can be launched. If the `/pps/services/bootmgr/last_tab` object is present (representing the tab that was active when the system was last shut down), the Boot Manager launches that tab first. Otherwise, it launches tabs in the order they are listed in `slm-config-modules.xml`. You can change the order in which tabs are launched as priority dictates by changing the order they are listed in `slm-config-modules.xml`

Chapter 3
Configuring the Target for Boot Optimization

Before you can perform any of the boot optimization procedures described in the sections that follow, you first need to have a target system running an SD-only image. Follow the instructions for "Installing a boot-optimized image" in the installation note included with your evaluation image.

Chapter 4
Optimizing Boot Times

Each system has its own set of boot time requirements to meet. Depending on your goals, there are a number of ways you can optimize the startup of the system. By implementing some simple techniques at various points in the boot sequence you can make the OS and applications load, initialize, and launch more quickly. For the QNX CAR platform, you can optimize startup times in three distinct areas:

To optimize:	See these sections in this guide for details:
The loading and launching of OS itself	• *Optimize the bootloader* (p. 19) • *Reduce the size of the startup program* (p. 20) • *Remove unnecessary debug printing* (p. 21) • *Reduce the size of the IFS* (p. 22) • *Generate the IPL to skip the image scan* (p. 24) • *Enable fast reading in the SD card* (p. 25) • *Use compression strategies* (p. 26)
The platform application stack	• *Make careful use of the default build script* (p. 27) • *Consider the placement of `waitfor` statements* (p. 28) • *Reorder the startup program* (p. 31)
The HMI	• *Optimize the HMI* (p. 32) • *Create modular applications* (p. 33) • *Statically link libraries* (p. 34)

In the QNX CAR platform, boot time optimization has been done using many of the techniques discussed in this chapter. These optimizations were focused on the following goals:

- Early splash screen and camera—meeting this goal required loading the IFS and getting to the build script as soon as possible, and then running the Screen Graphics Subsystem and the graphical app as early as possible (see *Optimize the bootloader* (p. 19), *Reduce the size of the IFS* (p. 22), and *Reorder the startup program* (p. 31)).
- Early audio—used the same techniques as early splash screen and camera, but starting audio as early as possible.
- Early HMI display (within 10 seconds)—used and benefited from the same techniques as above, but required additional work to reduce the HMI's dependencies

down to what was strictly necessary. This led to the development of Boot Manager, which allows the HMI to come up before all the apps are instantiated.

- Last audio playing within 10 seconds—required the invention of a new multimedia service that saves its state at shutdown and restores it at power-up. This also required careful management of this service's dependencies and placement in the SLM configuration.

Optimize the bootloader

Once developers get the system to boot for the first time, bootloader development often goes on the back burner. Here are a few techniques that sometimes get overlooked (all these optimizations are present in the QNX CAR platform):

- Enable data and instruction cache as early as possible. This sounds obvious, but some of the tight copy loops used in the bootloader benefit immensely from having the instruction cache enabled.
- Minimize or eliminate the boot script timeout. Bootloaders like RedBoot and U-Boot, which run a script, typically contain an automatic timeout that lets you abort the loading of one OS load and then load another OS. Also, the bootloader might print messages (for instance, help or welcome messages) to the serial port; you can suppress these. To modify the timeout in U-Boot, use the *bootdelay*, *bootcmd*, and *preboot* environment variables. For RedBoot, use `fconfig` to change the value for Boot script timeout. This step applies only if you're using a bootloader other than the IPL (which is not recommended for production systems).
- Don't scan for the OS image. If the system uses a default QNX IPL, you should look at the code in *main()* within `main.c` and remove anything unnecessary. In particular, look for code that calls *image_scan()* and replace it with the OS image's hardcoded address. You can also turn off the scan option when you generate the IPL (see *Generate the IPL to skip the image scan* (p. 24)).

  If you pad the IPL to a fixed size, you will always know where the OS image begins.

- Eliminate the bootup checksum. In most cases, the system has a single OS image. Consequently, performing a checksum to ensure the image's validity has little value, since you can't perform a recovery if the image has failed. Also, the checksum takes time; removing it allows your important code to start running sooner.

Reduce the size of the startup program

Startup is small (roughly 45K), so it's difficult to trim much fat from it. If you use the QNX Instant Device Activation technology, your minidrivers will be linked to the startup program and will consequently add to its load time. So make sure that your minidrivers are as small as possible—don't clutter them up with lots of unused debug or *kprintf()* calls.

For more information about minidrivers, see the *Instant Device Activation* guide.

Remove unnecessary debug printing

Callouts in either the IPL or the startup program handle any debug printing that happens early in the system boot (before the serial driver is loaded). These callout routines normally write directly to the registers of the first UART. But before the kernel has initialized, no interrupts are available. So, if the UART FIFO is full, the callouts can't insert a character until another character leaves the FIFO. With a standard UART, a blazingly fast startup can slow to a crawl if you burden the boot process with too many messages.

- Comment out unneeded *kprintf()* statements—In IPL or Startup, look for unneeded *kprintf()* statements in *main()* and comment them out.
- Reduce `-v` options—In the build script, find the line that launches the kernel (`procnto`) and reduce the `-v` options. For instance, if the line looks like this:

  ```
  PATH=:/proc/boot:/bin:/usr/bin
  LD_LIBRARY_PATH=:/proc/boot:/lib:/usr/lib:/lib/dll procnto –vvvv
  ```

  you can replace `-vvvv` with `-v` or simply remove the option altogether.
- Remove `display_msg` calls—In the build script, remove any `display_msg` calls that use the startup callouts. These include all `display_msg` statements that occur before the following sequence:

  ```
  waitfor /dev/ser1
  reopen /dev/ser1
  ```

  These statements redirect the serial output to the newly loaded serial driver (typically right above the `waitfor`), which will be interrupt driven and won't need to wait.
- Avoid a slow baud rate—Don't use a console baud rate less than 115,200 unless you absolutely must. Otherwise, you'll potentially spin longer in a loop in the kernel *printf()*, waiting for the UART FIFO to have enough space to send characters out. Chances are, you won't do this, for the simple reason that it's inconveniently slow. But in systems with few UARTs, it's tempting to share a 9600-baud GPS device with the default serial console. If you do this and still have some serial debug output in the kernel or startup, you could end up severely throttling back the code to keep pace with the slow baud rate.

Reduce the size of the IFS

The IPL copies the IFS from flash into RAM. The kernel and the applications can begin running only after this copy operation is complete. So the smaller you make the IFS, the sooner those components can run.

- Remove unused executables—Remove any unused executables from IFS, starting with the larger ones. Before you cut to the bone and remove anything that could help debug the target, you should measure your target's flash-to-RAM copy speed (see *Measuring the time to copy from flash to RAM* (p. 41) for more information). Remove executables from the image only if the benefits of doing so outweigh the loss of useful tools.

  Note that you don't have to manually strip executables of their debug information; `mkifs` takes care of that automatically. Note that `mkefs` doesn't automatically strip binaries—you should do this in your makefile.

- Use symbolic links—Shared libraries in POSIX systems, including the QNX OS, typically have two representations in the filesystem: a regular filename (with a version number) and a symbolic link (without a version number). For instance, `libc.so.2` and `libc.so`. The target system should contain both representations; thus, code that requires a specific version of the shared library can link to that version, and code that doesn't care can link to the generic version. Under Windows, which doesn't support true symbolic links, the QNX development installation creates duplicates of linked files, instead of symbolic links.

  If you use both versioned and nonversioned representations of shared objects on your target, take the time to make one a symbolic link to the other, either in the IDE or in the build script. Otherwise, you risk ending up with two distinct copies of the executable in the IFS. Since many shared libraries can be rather large (`libc.so`, for instance, ranges from 600K to 700K), taking this step can reduce the IFS significantly.

- Move selected files into an external filesystem—If any file doesn't need to start early in the boot process, move it into a flash external filesystem. The smallest IFS consists of the kernel, `libc`, a UART driver, a flash driver, and little else. After the flash driver loads, it can automount the external filesystem partitions, and you can start running the remainder of your drivers or applications files from there.

  There is a tradeoff here, of course. The IFS is completely loaded from flash into RAM as one big chunk. Once loaded into the IFS, any executables that you run out of IFS will load from RAM into RAM. For external filesystems, the files are loaded out of flash into RAM each time they're needed. So if you need to load an executable multiple times during bootup, it may be better to leave it in IFS since you pay the flash-copying penalty only the first time.

- Use the system optimizer to remove unreferenced libraries and functions—In many cases, you can shrink the IFS significantly by using the system optimizer (aka dietician) in the QNX Momentics system builder. The system optimizer finds any nonreferenced libraries and removes them completely. It can also remove functions from shared objects if those functions aren't referenced anywhere in the IFS. The system optimizer creates special reduced versions of the shared objects that the IDE builds for the target. The IDE places these smaller libraries in the `Reductions` subfolder of your system builder project.

  Some caveats:

  - You can use this tool only from within the QNX Momentics IDE; there is no command-line equivalent if you build your IFS outside of the IDE.
  - The reduced versions of the shared objects will contain only the functions required to run the files within your IFS. If you subsequently add a binary outside of the IFS, that binary will fail to load if it relies on any of the removed functions.
  - The system optimizer won't find code that uses *dlsym()* to dynamically load function addresses. To work around this, you can: a) create a stub library that references the required functions, thereby forcing them to be included, or b) skip running the system optimizer on a shared object if you will be dynamically loading the object with *dlopen()*.
  - You will generate new versions of the shared objects every time you run the system optimizer. This may require more configuration management for your project to keep track of the extra, reduced copies of the libraries.
  - You won't be using the "QNX-blessed" versions of the libraries.

  Despite these caveats, the system optimizer offers a very useful and relatively effortless way to shrink the IFS. The savings will directly translate into shorter boot times.

Generate the IPL to skip the image scan

The IPL normally scans for a valid system image so that it can load that image into RAM. In a production system, however, the image is a known size, so this step is unecessary. To reduce the time it takes for the image to boot, you can configure the IPL to skip the memory scan.

To build the IPL with the image scan disabled, follow the instructions to "Build a Target Image" in the *Building and Customizing Target Images* guide, with the following modifications:

1. Build the BSP with the following options:
   ```
   make CCOPTS="-DSKIP_IMAGE_SCAN -DBTMODE=BTMODE_SD"
   ```
2. Run `mkflashimage`:
   ```
   cd images/
   sh mkflashimage
   ```
3. Rename and copy the generated IPL as follows, depending on your target (*install_location* is the location where you installed QNX SDP 6.6):

   OMAP5 5432uevm

   Rename the file `sd-ipl-omap5-uevm5432.bin` to `MLO` and copy it to *`install_location`*`/deployment/qnx-car/boards/omap5uevm/sd-boot/`.

   i.MX6q SABRELite

   Rename the file `ipl-mx6q-sabrelite.bin` to `ipl-mx6q-sabrelite.binqnx660` and copy it to *`install_location`*`/deployment/qnx-car/boards/imx61sabre/sd-boot/`.

   Jacinto 5 Eco

   Rename the file `nand-ipl-ti-j5eco_dra62x-evm811x.bin` to `MLO` and copy it to *`install_location`*`/deployment/qnx-car/boards/jacinto5eco/sd-boot/`.

4. Generate a an image by running `mksysimage.bat` or `mksysimage.sh`.
5. Follow the instructions for copying the image to an SD card and booting the image.

Enable fast reading in the SD card

Reading from an SD card is much slower than reading from disk. Modify the SD card driver to enable fast reading to reduce the time it takes to read the image from the card.

Use compression strategies

You can either compress the entire IFS or compress individual files in the EFS. (If you're using the QNX Instant Device Activation TDK, you cannot compress the IFS.) Besides saving flash memory, compression can also speed up boot time. In systems with very slow flash access, it often takes less time to decompress files out of flash than to do a straight copy of the larger uncompressed file (enable the cache in the IPL to reduce decompression time). If your system's flash timing is on the slow side, try using compression; the decompression code might be able to run completely out of the CPU instruction cache. Of course, this depends on what else the system is doing during the boot; you'll need to try both approaches and measure which is quicker.

Make careful use of the default build script

The build scripts that QNX provides normally have many components commented out for a minimal system. Uncomment these components as required, but first determine what you actually need:

- `slogger`—The system logger, which allows QNX components to report errors, is useful during development. However, your production systems may not have any way to access the errors reported. If so, you don't need `slogger` (or `sloginfo` for that matter) in the final build. You can also remove `slogger` if you use your own logging subsystem.
- `pipe`—Supports the POSIX pipe facility (for instance, `ls | more`). You can also use pipes programmatically, without resorting to scripting. Many embedded systems don't use pipes, so you might be able to remove this.
- `devc-pty` and `qconn`—Also needed for debugging and development, these could be removed for production systems.

Consider the placement of `waitfor` statments

The build script contains multiple calls to `waitfor`, which ensure that a resource manager is loaded before any of the programs that might use it. This is a very good practice, since the programs that follow may fail if they don't find the resource they require. However, in the default build script, these `waitfor` statements are grouped to make sense, rather than to ensure maximum performance. For example, consider the following (simplified) example code:

```
...

# I2C driver
display_msg starting I2C driver...

# I2C0 interface
i2c-omap35xx-j5 -i 70 -p0x48028000 --u0
waitfor /dev/i2c0

# I2C1 interface
i2c-omap35xx-j5 -i 71 -p0x4802A000 --u1
waitfor /dev/i2c1

# I2C2 interface
i2c-omap35xx-j5 -i 30 -p0x4819C000 --u2
waitfor /dev/i2c2

# I2C3 interface
i2c-omap35xx-j5 -i 31 -p0x4819E000 --u3
waitfor /dev/i2c3

# USB OTG Host Controller driver
io-usb -vvv -d dm816x-mg ioport=0x47401400,irq=18
waitfor /dev/io-usb/io-usb 4
devb-umass cam pnp

# AUDIO Driver - I2C must be running
display_msg Starting Audio driver...
# MCASP2
io-audio -vv -d mcasp-j5_aic3106 mcasp=2
waitfor /dev/snd/pcmC0D0p

# SPI driver
display_msg starting SPI driver...
# SPI 0
spi-master -u0 -d dm816x base=0x48030100,irq=65,somi=0,edma=1,edmairq=529,edmachannel=17

# PCIe server
display_msg Starting PCI server...
pci-dm814x
waitfor /dev/pci 4

 ...
```

This script does the reasonable thing of starting each driver, then waiting for it to finish loading before continuing. Some of these drivers require hardware initialization. If a driver is waiting on the hardware, then `waitfor` can prevent the next program from loading prematurely.

The behavior of `waitfor` is very simple: it polls the device, and if the device isn't found, it sleeps for 100 milliseconds and tries again. It terminates when either the device is found or the timeout is reached, whichever happens first. As a result, each `waitfor` might do nothing except poll and hold up the rest of the show. You want the CPU 100% utilized during the boot—any idle time adds to the total boot duration. Ideally, then, each `waitfor` would do a single device check that succeeds and then

continues. An ordering that breaks the logical grouping can minimize unwanted sleeps by using other program loads to introduce any required delay.

For instance, let's say you need to start an IDE driver in your boot process. That driver must wait for the hardware to initialize, an operation that always takes 100 milliseconds. That's what `waitfor` does: it waits until your driver has the hardware initialized before proceeding. But why waste that 100 milliseconds? After starting the IDE driver, start your USB driver (or any other software) that can effectively utilize that time. If your USB driver takes 100 milliseconds to prepare the hardware, you've gotten some extra time "for free." Then, when you actually need the IDE device, the `waitfor` test will succeed immediately. And you've managed to shorten the total boot time.

See the following code for an example of modifying the script in this way:

```
...

# I2C driver
# We won't wait for any of these, since nothing needs them yet
display_msg starting I2C driver...

# I2C0 interface
i2c-omap35xx-j5 -i 70 -p0x48028000 --u0

# I2C1 interface
i2c-omap35xx-j5 -i 71 -p0x4802A000 --u1

# I2C2 interface
i2c-omap35xx-j5 -i 30 -p0x4819C000 --u2

# I2C3 interface
i2c-omap35xx-j5 -i 31 -p0x4819E000 --u3

# USB OTG Host Controller driver
display_msg Starting USB OTG Host driver...
io-usb -vvv -d dm816x-mg ioport=0x47401400,irq=18

# Start the SPI driver before checking on USB, since SPI doesn't rely on io-usb

# SPI driver
display_msg starting SPI driver...
# SPI 0
spi-master -u0 -d dm816x base=0x48030100,irq=65,somi=0,edma=1,edmairq=529,edmachannel=17

# Check on USB relocated from above
waitfor /dev/io-usb/io-usb 4
devb-umass cam pnp

# PCIe server
display_msg Starting PCI server...
pci-dm814x
waitfor /dev/pci 4

# I2C driver should be up by now, and we need it for audio
waitfor /dev/i2c0
waitfor /dev/i2c1
waitfor /dev/i2c2
waitfor /dev/i2c3

# The audio driver requires I2C, so we've moved it later in the build file
# (after SPI and PCIe), to allow more time for the I2C drivers to initialize

# AUDIO Driver - I2C must be running
# McASP2
io-audio -vv -d mcasp-j5_aic3106 mcasp=2
waitfor /dev/snd/pcmC0D0p

...
```

These examples illustrate the benefits of optimized `waitfor` placement. This technique has a potential drawback, however: the driver might not be waiting on the hardware, but rather using the processor to do real work. In that case, the reordering

will cause all the drivers to load at once, which will make the task scheduler continually switch between all the active threads. This can be less efficient than the first method.

To determine whether reordering will improve boot performance, use `tracelogger` to capture a system profiler snapshot during boot. If the snapshot shows blocks of time where the CPU is idle after a driver load and indicates that calls are being made into the kernel every 100 milliseconds, then that driver is a reasonable target for this technique.

Reorder the startup program

If you're used to working with a monolithic kernel like Linux or Windows, you might be inclined to start all your drivers before you start any applications. But with a microkernel OS, you have more flexibility and can reorder some of your startup program to take advantage of any idle time. That includes starting applications before starting drivers, wherever it makes sense. In addition, you can use SLM to manage utilitites and services that can be started later or as needed.

A good example of this is the network driver. While the HMI needs the network stack (`io-pkt`) to be up, it doesn't necessarily need the network device driver to be loaded or network connectivity to be established, so the HMI is launched without waiting for the network driver. Although some applicatiions will need this, only those apps, and not the whole HMI, should have to wait.

Optimize the HMI

The HTML5 HMI is large and can potentially take a long time to launch. The HTML5 HMI layer includes the Browser Engine (also called the Web Engine or Web Launcher), HTML5 application framework, the Navigator (also known as the Applications Window Manager), and the HMI Notification Manager.

A couple of key techniques help to speed up the launch of the HMI:

- To optimize the brower engine, all large browser libraries are loaded in a secondary IFS. Using the utility `mount-ifs`, the browser engine can read these large libraries from the disk much faster than from a regular filesystem.
- The browser engine can run as a zygote and applications can be forks of the zygote process, so can use the libraries that are already loaded in memory. The exception to this is the Navigator, which needs root access, so doesn't run as a zygote and gets its libraries from the IFS.

In addition, to speed the launch of applications, the HTML5 apps in the prebuilt images are “minified”. Minification makes the source code smaller by removing comments and white space, and possibly also shortening symbol names. The resulting code loads faster in the HMI. We recommend that you minify your HTML5 and JavaScript code for production, which you can do using any off-the-shelf minification tool.

Create modular applications

If you design a system with a single main application, none of the application logic can run until the entire application is loaded into memory. The larger the application, the more of a problem this becomes. Consequently, it often makes sense to break your software system into several logical modules that run as separate processes. Those processes can communicate via any number of interprocess communication (IPC) mechanisms. Having separate processes also gives you more flexibility in load order, provided they're not fully dependent on one another. As a side benefit, you gain protection from memory isolation between those processes.

Statically link libraries

Shared libraries take time to load. When an application is linked to a shared object, the process loader will first check whether that shared object is already loaded. If it isn't, the loader must load the object out of permanent storage first (IFS, EFS, or elsewhere). The process of loading the various ELF sections from the file can take time. Even if the shared object is already in memory, the application must have fixups applied. The dynamic linker must look up the symbol names to get the appropriate addresses.

For a large shared object, it can be significantly quicker to statically link the application with the biggest libraries. That way, you pay for the linker lookup penalties at compile time rather than at runtime. Of course, statically linking an executable will consume more flash memory if multiple applications call from that library. Also, this practice may introduce version incompatibilities between applications if the shared library changes and you don't rebuild everything it's linked against. But for some systems, the performance benefits will outweigh the drawbacks.

Chapter 5
Measuring Boot Times

Developers and system designers can employ many techniques to meet early boot requirements. However, before applying any of the techniques described here, always remember to get a stable baseline measurement of system boot speed. That way, when you start making changes, you can ensure that you're making real progress towards meeting your requirements.

To optimize any boot stage, you must measure its duration, modify the code, then measure again to see how much timing has improved. Some basic techniques exist for measuring time; their applicability depends on the starting point of the measurement. There are three key points where you can measure times:

- Before the IPL is loaded the CPU can't execute instructions, so time measurements at this point require hardware assistance. This point is labeled "1" in the diagram in the section *System startup sequence* (p. 11).
- Software can run between the startup driver launching (label "2") and the kernel being fully operational (label "3"), but not always with the same functions. For example, startup code cannot use most RTOS services, including POSIX timers. It supports only a limited subset of functions—such as *memcpy()*, *strcpy()*, *printf()*, and *kprintf()*—to perform rudimentary operations.
- When optimizing times after the kernel is running (label "3"), you can access any OS feature, run all programs, and connect to the IDE with its assortment of tools.

The table that follows describes some of the techniques that can be used to measure times at these points in the bootup sequence:

Start Time	Technique	Accuracy	Description	Pros and Cons
After the kernel is running	*TraceEvent()*	Microseconds	Uses the instrumented kernel (`procnto-instr`) and collects data with tracelogger or the QNX Momentics system profiler. Customer code is sprinkled with calls to the *TraceEvent()* function.	Can graphically display when your process is executing, as well as all other system activity. The developer must set up the instrumented kernel.
After the kernel is running	`time`	Milliseconds	Command-line utility gives approximate	Measurement is unavailable until the

Start Time	Technique	Accuracy	Description	Pros and Cons
			execution time of a process.	process in question terminates.
After the kernel is running	*ClockCycles()*	Nanoseconds	System function that uses a high-speed CPU counter to determine the number of clock cycles from power on to the point when *ClockCycles()* is called.	Measures absolute time. Doesn't necessarily reflect time spent in the measured process, since the kernel may have scheduled other threads during time of measurement.
After the kernel is running	*slogf()* / `sloginfo`	Seconds	System logger function, used with `slogger`.	Inaccurate timing; used mainly to determine sequence of events.
After the startup driver starts and before the kernel is running	*ClockCycles()* (macro)	Nanoseconds	Not a function, but a macro that reads the CPU's hardware counter directly. Gives the same result as the OS-level function of the same name, which is available after kernel boot.	Not supported on all architectures; works only if *ClockCycles()* is read directly from a hardware register, and not a derived value.
After the IPL starts and before the kernel is running	GPIO and scope	Nanoseconds	The customer code switches a GPIO pin on and off at various points in the code. A digital oscilloscope measures these level changes or pulses to determine the time between events.	Distinguishing different points is impossible. Requires a free GPIO in the hardware design, as well as a digital scope and significant setup.
Before the IPL starts	Hardware lines and scope	Nanoseconds	Measures hardware lines (like CPU reset) and GPIO.	Same as above.

For the *TraceEvent()* technique, you must use the instrumented kernel and load `tracelogger` early in the boot script. For instance, to log the first ten seconds of boot time, you would use this command:

```
tracelogger –n0 -s10
```

See the `tracelogger` documentation for details on how to analyze the resulting `.kev` (kernel event trace) file.

To measure the absolute time since reset at various points in your boot script, simply print out the *ClockCycles()* value:

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

int main( int argc, char *argv[])
{
    uint64_t timesinceboot_ms;

    timesinceboot_ms = (ClockCycles() /
                        (SYSPAGE_ENTRY(qtime->cycles_per_sec/1000));

    printf( "ClockCycles()=%llu ms\n", timesinceboot_ms);

    return EXIT_SUCCESS;
}
```

This technique lets you measure how long it takes your code to execute the IPL and startup phases. Normally, you would use the *ClockCycles()* value to measure relative time: you record the value of *ClockCycles()* at two points, then subtract the first value from the second value to get the duration of an event. In this case, however, we're using *ClockCycles()* to measure the absolute time that has elapsed since the CPU power was applied.

This approach comes with some caveats:

- The high-speed counter counts very quickly and can wrap, so it's best to apply this technique during the first several seconds after the CPU has been reset.
- Depending on how the BSP implements reset, a `shutdown` command to reset the target may fail to clear the *ClockCycles()* value. If so, you might have to power-cycle the device.
- This technique applies only to systems that have a high-speed counter. Systems where the OS emulates *ClockCycles()* and where the CPU has no high-speed counter won't give an absolute time since reset.

The `boot_metrics.log` file

You can use the `boot_metrics.log` file to monitor startup times for your system.

The `boot_metrics.log` file contains time measurements from board reset to a particular system event (the time from board reset varies depending on your platform).

Although this log is purely for reference purposes, you can get important information from it that can help you tune your system startup.

Variation in timers

Each measurement in the log starts with either `(hw)` or `(sw)`:

- Times that start with `(hw)` are measured using a simple utility called `timestamp` that makes a call to the *ClockCycles()* kernel function. This function provides the `timestamp` utility with the number of clock cycles since the board was reset. Note that these measurements aren't entirely accurate because they're taken close to—but not simultaneously with—the event that's being measured. The `timestamp` utility is just a process that's run in the background at a normal priority, as closely as possible to the event being measured.
- Times that start with `(sw)` are doing some other measurement, like querying the system for the date.

On some boards, the time reported by the `timestamp` utility correlates very closely with the time elapsed since power was actually applied to the board (i.e., a board reset was done either in software or by pressing the reset button). This is the most desirable implementation. In some cases, however, the time reported by the `timestamp` utility could represent the time since the IPL started, or even the time since the startup driver started. This situation results in measurements that are much smaller than would be observed using a stopwatch from power on. These measurements are obviously less accurate, but can still be useful in comparing one software build to another.

The particular measurements you get depend on whether:

- the board provides a counter in the hardware that starts as close as possible to power-on reset (PoR). If the board doesn't provide a hardware counter, you can use the counter provided in the kernel, but it won't start counting until the kernel takes control, which is well after PoR.
- the IPL/startup driver can (or actually does) initialize this hardware counter to zero.
- the startup driver performs a “cold” or “warm” reset on shutdown (software reset). A cold reset generally initializes the hardware more thoroughly than a warm reset, so a cold reset would be more likely to set the hardware counter to 0.

Contents of boot_metrics.log

The boot_metrics.log is found in the /dev/shmem directory. Its contents look like this:

```
(hw) CAR_BOOT_METRICS: (IFS SCRIPT START) at 0.484901 seconds
(hw) CAR_BOOT_METRICS: (LAUNCHING EARLY-SPLASH) at 0.629556 seconds
(hw) CAR_BOOT_METRICS: (REARVIEW CAMERA DONE) at 0.632296 seconds
(hw) CAR_BOOT_METRICS: (LAUNCHING EARLY-CHIME) at 0.661630 seconds
(hw) CAR_BOOT_METRICS: (STARTING PPS) at 0.759833 seconds
(hw) CAR_BOOT_METRICS: (LAUNCHING PPS) at 0.798557 seconds
(hw) CAR_BOOT_METRICS: (LAUNCHING MOUNT-IFS2) at 0.815651 seconds
(hw) CAR_BOOT_METRICS: (LAUNCHING SLM) at 0.852900 seconds
(hw) CAR_BOOT_METRICS: (EARLY-SPLASH WINDOW CREATED) at 0.961214 seconds
(hw) CAR_BOOT_METRICS: (DONE MOUNT-IFS2) at 1.161844 seconds
(hw) CAR_BOOT_METRICS: (STARTING SLOGGER2) at 1.354558 seconds
(hw) CAR_BOOT_METRICS: (DONE PPS (/pps is available)) at 1.364052 seconds
(hw) CAR_BOOT_METRICS: (HMI LAUNCHED) at 1.837985 seconds
(hw) CAR_BOOT_METRICS: (DONE WEB-ZYGOTE) at 2.514162 seconds
(hw) CAR_BOOT_METRICS: (DONE EARLY-CHIME) at 2.746928 seconds
(hw) CAR_BOOT_METRICS: (MMPLAYER LAUNCHED) at 3.742285 seconds
(hw) CAR_BOOT_METRICS: (SET DATE) at 13.623120 seconds
(sw) CAR_BOOT_METRICS: (SYSTEM SECONDS) at 1391095010
(sw) CAR_BOOT_METRICS: (SYSTEM DATE) at Thu Jan 30 10:16:50 EST 2014
(hw) CAR_BOOT_METRICS: (HMI LOADED) at 14.947910 seconds
```

The events that appear in the log file are as shown in the following table:

This log entry:	Corresponds to this system event:
IFS SCRIPT START	The IFS build script has started.
LAUNCHING EARLY-SPLASH	The early splash screen has been launched.
REARVIEW CAMERA DONE	The rearview camera is ready.
LAUNCHING EARLY-CHIME	The early audio chime has been launched.
STARTING PPS	PPS has been launched.
LAUNCHING PPS	PPS has been launched.
LAUNCHING MOUNT-IFS2	The process to mount the secondary IFS has been launched.
LAUNCHING SLM	SLM has been launched.
EARLY-SPLASH WINDOW CREATED	The early splash screen window has been created. This is close to but not necessarily exactly the same time as when you see the splash screen on the display.
DONE MOUNT-IFS2.	The secondary IFS has been mounted
STARTING SLOGGER2	The slogger2 daemon has been launched.

This log entry:	Corresponds to this system event:
DONE PPS (/pps is available)	PPS is ready and the PPS filesystem has been mounted.
HMI LAUNCHED	The HMI is ready.
DONE WEB-ZYGOTE	The browser engine zygote is ready.
DONE EARLY-CHIME	The early audio chime has finished playing.
MMPLAYER LAUNCHED	The `mm-player` service is ready (so early audio is available).
SET DATE	The system date has been set.
SYSTEM SECONDS	The system time as reported by the POSIX-standard `date -t` utility.
SYSTEM DATE	The current system date and time.
HMI LOADED	The HMI is has loaded.

You can write additional events to the log by running the `timestamp` utility (e.g., `timestamp` *`event_name`*). The event information will be written to `/dev/shmem/boot_metrics.log`.

Measuring the time to copy from flash to RAM

In the IPL and Startup stages, code is copied from flash into RAM and then executed. How long this takes depends on the speed of the CPU and the speed of the flash chip. To measure the duration of the copy operation, you can use the following code:

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

#define MEGABYTE (1024*1024)
#define BLOCK_SIZE 16384
#define LOOP_RUNS 10

char *ram_destination;
char *ram_block;
char *flash_block;
unsigned long flash_addr;
uint64_t cycles_per_sec;

double CopyTest(const char *msg, char *source, char *dest)
{
    uint64_t accum = 0, start, stop;
    double t;
    int i;

    for (i=0; i<LOOP_RUNS; i++)
    {
        start = ClockCycles();
        memcpy(dest, source, BLOCK_SIZE);
        stop = ClockCycles();
        accum += (stop - start);
    }
    accum /= LOOP_RUNS;

    t = accum*(MEGABYTE/BLOCK_SIZE);  // t = cycles per MB
    t = t / cycles_per_sec;  // t = seconds per 1MB

     printf("\nTo copy %s to RAM takes:\n",msg);
     printf("   %llu clock cycles for %u bytes\n", accum, BLOCK_SIZE);
     printf("   %f milliseconds per 1MB bytes\n", t*1000);
     printf("   %f microseconds per 1KB bytes\n", t*1000);
     return t;
}

int main(int argc, char *argv[])
{
    double flashtime, ramtime;

    if (argc<1) {
    printf("%s requires address of flash (any 16K block will do)\n", argv[0]);
    return EXIT_FAILURE;
    }

    flash_addr = strtoul(argv[1], 0, 16);
    printf("Using flash physical address at %lx\n", flash_addr);

    ram_block = malloc(BLOCK_SIZE);
    ram_destination = malloc(BLOCK_SIZE);
    flash_block = mmap(0, BLOCK_SIZE, PROT_READ,MAP_PHYS|MAP_SHARED, NOFD,flash_addr);
    if (flash_block == MAP_FAILED) {
        printf("Unable to map block at %lx\n", flash_addr);
    }
    cycles_per_sec = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
    flashtime = CopyTest("flash", flash_block, ram_destination);
    ramtime = CopyTest("RAM", ram_block, ram_destination);
    printf("\nFlash is %f times slower than RAM\n", flashtime/ramtime);
```

```
    return EXIT_SUCCESS;
}
```

To get reasonably accurate results, you should run the preceding code either at a high priority (using the on -p command) or when little else is running in the system.

A key factor that affects flash copy time is the bus interface to the flash. Fast CPUs can lose their advantage to their slower competitors if the system has a slow bus architecture or too many wait states.

Index

T

W