QNX SOFTWARE SYSTEMS

Meeting Early Boot Requirements with the QNX Neutrino RTOS

Andy Gryc
Worldwide Automotive FAE
QNX Software Systems
Email contact: paull@qnx.com

Introduction

Many embedded applications must perform a set of actions within a strict timeframe after the system boots. Meeting these deadlines can be a challenge, and the system architect must design the system with early boot requirements in mind. From a software perspective, the boot process consists of several stages. First, the RTOS must load from nonvolatile storage, either flash or hard disk. Next, the RTOS must initialize itself, as well as any device drivers and peripherals. Then, the application software must load, initialize, and start running. All of this takes time, and the system architect or designer must think through each of these stages to ensure that software and hardware components are initialized and ready when needed.

Early boot requirements are especially prevalent in automotive. Typically, an automotive system must receive and respond to vehicle bus messages within 50 milliseconds after power is applied. Radio systems must also meet “early audio” requirements, which consist of the time between “ignition on” (which is when the system boots) and when the passengers hear the radio. These times can vary, but normally range from 1 to 4 seconds after boot. A more sophisticated radio may have to meet multiple early audio requirements; for instance, playing FM radio within 1 second, playing satellite radio within 3 seconds, and playing MP3s off of a USB stick within 4 seconds. Systems with screens have early video requirements to support backup cameras, navigation applications, or splash screens.

This paper addresses longer timelines on the order of one second or more. Application-level code typically handles these events, and therefore must run after the RTOS has booted. The emphasis, then, is on optimizing your RTOS and application startup to boot as quickly and efficiently as possible. System architects can use this paper as a springboard, since it provides a number of techniques and strategies. These techniques focus on the QNX® Neutrino® RTOS, though some apply to other operating systems as well.

Typically, a full-featured RTOS cannot load and initialize quickly enough to handle early boot deadlines of 100 milliseconds or less. For such deadlines, system designers need a solution such as the QNX Instant Device Activation Technology Development Kit (TDK). The paper doesn’t cover this technology, except in areas where it impacts the bigger picture.

Startup sequence

Like most operating systems, the QNX Neutrino RTOS boots in several stages. See Figure 1.

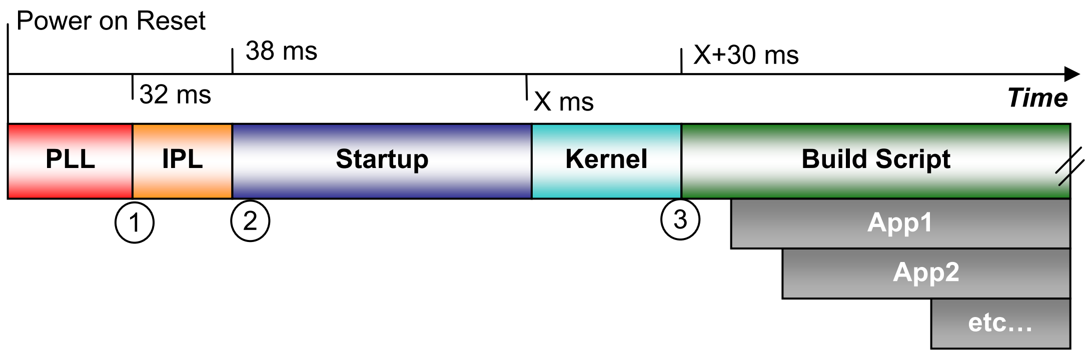


Figure 1 — QNX Neutrino boot process (durations not to scale).

Here is a brief description of each stage:

- **PLL (phase locked loop)** — In this paper, 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 load)** — The bootloader. QNX provides a standard, bare-bones IPL that performs the fewest steps necessary to set up memory, initialize chip selects, and configure other required CPU settings. Once these steps are complete, the IPL copies Startup into RAM and jumps to it to continue execution.

  You can replace the standard QNX IPL with another bootloader. Choices include full-featured products like U-Boot or RedBoot, which provide a TFTP client to transfer a new boot image and burn it into flash. Typically, the bootloader executes for at least 6 milli-seconds before it starts to load the OS image. The actual amount of time depends on the CPU architecture, what the board requires for minimal configuration, and what the chosen bootloader does before it passes control to the OS.

- **Startup** — Copies the OS image from flash into RAM and executes the image. This copying phase constitutes the longest part of the QNX Neutrino kernel boot process. That said, the system architect can exercise a great deal of control over the length of this phase, albeit indirectly.

  In QNX Neutrino, the image file system (IFS) contains both the OS image and Startup (which the IPL copies into RAM). The OS image consists of the kernel, the build script, and any other drivers, applications, and binaries that the system requires. Because you can control what the IFS contains, the time for this stage is variable and listed as *X* in Figure 1. The time can be as short as 200 milliseconds or as long as 2 to 3 seconds. In extreme cases where the system contains a very large image and has no file system other than the IFS, this stage may take much longer (10 seconds or more).

To add, remove, or configure files stored in the IFS, you can edit the build script or use the system builder tool in the QNX Momentics® IDE.

- **Kernel —** During startup, the kernel initializes the memory management unit (MMU); creates structures to handle paging, processes, 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 script.

- **Build script —** Lets you specify which drivers and applications to start, and in what order. Because the QNX Neutrino RTOS is a microkernel OS, every driver, server, and application (App1, App2, etc. in Figure 1) runs as a separately loadable user-level process. This approach offers a major benefit: you don't have to wait for the kernel and a full set of drivers to load and initialize before you get control. Once the microkernel is running, you can interleave drivers and applications to achieve the fastest possible system startup. More on this later.

Measuring bootup time

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, and their applicability depends on the starting point of the measurement. If you look at Figure 1, you'll see three key points represented by circled numbers:

- Times measured before ① occur before the CPU executes instructions and require hardware assistance.

- Software can run during points between ① and ③, but not always with the same APIs. For example, Startup code cannot use most RTOS services, including POSIX timers. It supports only a limited subset of functions — like *memcpy*(), *strcpy*(), *printf*(), and *kprintf*() — to perform rudimentary operations.

- When optimizing times after point ③, you can access any OS feature, run all programs, and connect to the QNX Momentics IDE with its assortment of tools.

Start Time	Technique	Accuracy	Description	Pros/Cons
After ③	*TraceEvent*()	Micro-seconds	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*() API.	Can graphically display when your process is executing, as well as all other system activity. Developer must set up the instrumented kernel.
After ③	**`time`**	Milli-seconds	Command-line utility gives approximate execution time of a process.	Measurement unavailable until the process in question terminates.
After ③	*ClockCycles*()	Nano-seconds	System API 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 kernel may have sche-duled other threads during time of measurement.
After ③	*slogf*() / **`sloginfo`**	Seconds	System logger API, used with **`slogger`**.	Inaccurate timing; used mainly to determine sequence of events.
Between ② & ③	*ClockCycles*() (macro)	Nano-seconds	Not an API, but a macro that reads the CPU's hardware counter directly. Gives the same result as the OS-level API 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.
Between ① & ③	GPIO and scope	Nano-seconds	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 ①	Hardware lines and scope	Nano-seconds	Measures hardware lines (like CPU reset) and GPIO	Same as above

Table 1 — Techniques for measuring bootup time.

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 QNX **`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; see the code in Listing 1. 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 and 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 the CPU has no high-speed counter won't give an absolute time since reset.

```
#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;
}
```

Listing 1 — Using *ClockCycles*() to measure time elapsed since reset.

Bootloader optimizations

Once developers get the system to boot for the first time, bootloader development often goes on the backburner. Here are a few techniques that sometimes get overlooked:

- **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 OS load and load a new OS. This timeout is convenient during development, but should be removed for production. Also, the bootloader might print messages (for instance, “help” messages 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`**.

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

- **Eliminate 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, as you can’t perform a recovery if the image has failed. Also, the checksum take time; removing it allows your important code to start running sooner.

Copying 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 to the flash chip. To measure the duration of the copy operation, you can use the code in Listing 2.

```
#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;
}
```

Listing 2 — Code to measure flash copy time.

To get reasonably accurate results, you should run the code in Listing 2 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. As Table 2 shows, fast CPUs can lose their advantage to their slower competitors if the system has a slow bus architecture or too many wait states.

Architecture	Clock speed	Measured flash copy speed (per Kbyte)
SH-4	200MHz	59 µs
ARM9	200MHz	93 µs
PowerPC	400MHz	514 µs

Table 2 — Sample flash copy times.

Why not execute directly out of flash and skip the copy step altogether? Because reading from flash is very slow and should be avoided. Forcing the CPU to continually read code from flash would save some time initially, but the abysmally slow performance that would result far outweighs any benefit. To make this strategy even worth considering, the CPU instruction cache would have to be larger than the OS image.

Reduce Startup Size

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

Remove unnecessary debug printing

Callouts in either IPL or Startup handle any debug printing that happens early in the system boot (prior to the serial driver being loaded). These callout routines normally write directly to the registers of the first UART. But before stage ③ in the boot process, 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 it 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 115200 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 size of IFS

As mentioned earlier, Startup copies the image file system (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 Listing 2). If it takes 100 µs per Kbyte to copy, then a 40K executable will take 4 milliseconds to copy. 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 QNX Neutrino, typically have two representations in the file system: a regular file name (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; that way, 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 Momentics installation creates duplicates of linked files instead of symbolic links.

  If you use both versioned and non-versioned 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 EFS** — If any file doesn't need to start early in the boot process, move it into a flash external file system (EFS). The smallest image file system (IFS) consists of the kernel, libc, a UART driver, a flash driver, and not much else. After the flash driver loads, it can automount the EFS partitions, and you can start running the remainder of your drivers or applications files out of the EFS.

  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 EFS, 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; see Figure 2. It can also remove functions from shared objects if those functions aren't referenced anywhere in the IFS; see Figure 3. The system optimizer will create 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.

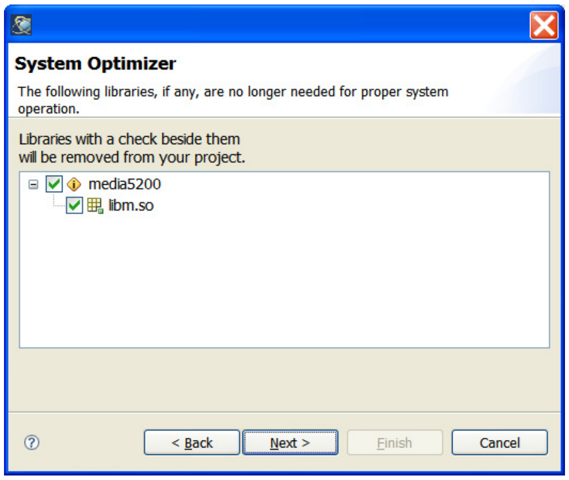
  - The reduced versions of the shared objects will contain only the functions required to run the files within your IFS. If you subsequently add a binary outside of the IFS, that binary will fail to load if it relies on any of the removed functions.

- 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. See Table 3 for an example of how the actions chosen in figures 2 and 3 reduced the size of a typical IFS. This IFS contained the full set of files needed for the BSP, but no application code. Of course, your mileage will vary, depending on how fully the code in your IFS binaries utilizes functions within shared libraries. The more functions you call, the fewer objects can be removed, and the smaller the resultant savings.



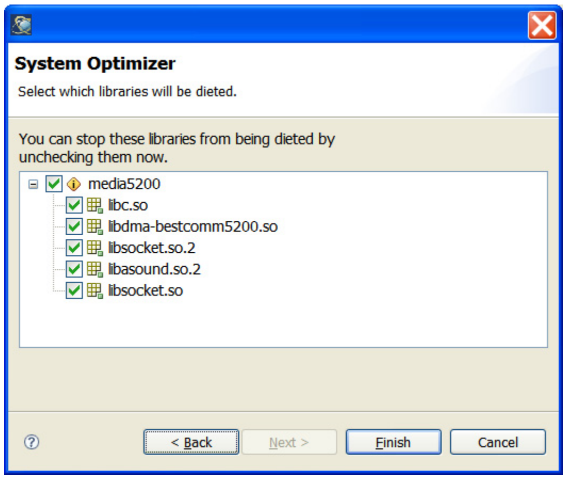
System Optimizer

The following libraries, if any, are no longer needed for proper system operation.

Libraries with a check beside them will be removed from your project.

- ✓ media5200
  - ✓ libm.so

Figure 2 — Using the system optimizer to remove unused libraries.



System Optimizer

Select which libraries will be dieted.

You can stop these libraries from being dieted by unchecking them now.

- ✓ media5200
  - ✓ libc.so
  - ✓ libdma-bestcomm5200.so
  - ✓ libsocket.so.2
  - ✓ libasound.so.2
  - ✓ libsocket.so

Figure 3 — Using the system optimizer to remove functions from libraries.

Library	Original size	Reduced size	Bytes removed	Reduction
libc.so	716K	499K	217K	30%
libsocket.so	173K	144K	29K	17%
libdma-bestcomm.so	39K	28K	11K	28%
libm.so	174K	0 (removed)	174K	100%
Total	**1102K**	**671K**	**431K**	**39%**

Table 3 — Example system optimizer reductions.

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. If your system's flash timing is on the slow side with regard to Table 2, 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.

Default build script

The build files 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.

Waitfor placement

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`**s are grouped to make sense, rather than to ensure maximum performance. For example, look at Listing 3.

```
.
.
.
# Starting PCI driver
pci-mgt5200
waitfor /dev/pci 4

# Starting Network driver
io-net -dmpc5200 verbose -ptcpip
waitfor /dev/io-net/en0 10
ifconfig en0 192.168.0.2

# Starting Flash driver...
devf-mgt5200 -s0xfe000000,32M
waitfor /fs0p1
/fs0p1/dumper &

# Starting USB driver...
io-usb -dohci-mgt5200 ioport=0xf0001000,irq=70
waitfor /dev/io-usb/io-usb 4
.
.
.
```

Listing 3 — Build script sequence with original waitfor placement.

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.

For example of modifying the script in this way, see Listing 4.

```
.
.
.
# Starting PCI driver
pci-mgt5200

# Starting Network driver
io-net -dmpc5200 verbose -ptcpip

# Starting Flash driver...
devf-mgt5200 -s0xfe000000,32M

# Starting USB driver...
io-usb -dohci-mgt5200 ioport=0xf0001000,irq=70

# Move these after other driver launches
# to make sure hardware has had enough delay
waitfor /fs0p1
/fs0p1/dumper &

waitfor /dev/io-net/en0 10
ifconfig en0 192.168.0.2

# Nothing utilizes these devices within this block,
# so these waitfors have been moved to the bottom.
# The intervening code has taken the place of any
# needed delays.
waitfor /dev/pci 4
waitfor /dev/io-usb/io-usb 4
.
.
.
```
Listing 4 — Build script sequence with reordered waitfor placement.

Figures 4 and 5 illustrate the benefits of optimized **`waitfor`** placement.

This technique has a potential drawback: 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 milli-seconds, then that driver is a reasonable target for this technique.

Default waitfor placement

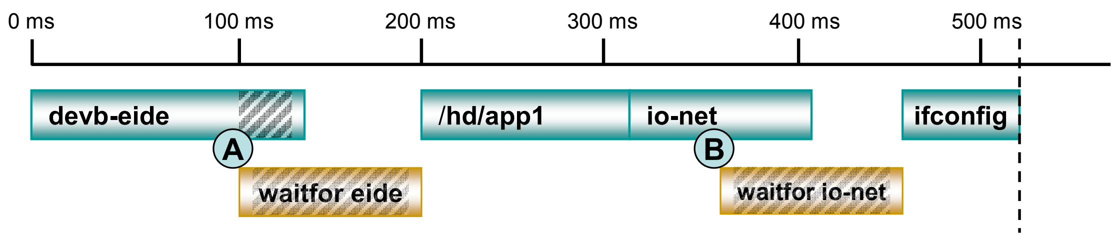


Figure 4 — Default `waitfor` placement. At point A, the EIDE driver starts waiting for a hardware interrupt. Since the named resource isn't available when the **`waitfor`** starts, the **`waitfor`** sleeps for 100 milliseconds. In the meantime, the hardware finishes and the EIDE driver completes. But because the **`waitfor`** is still completing its 100 millisecond sleep, there is a delay in launching **`app1`** off of the hard disk. At point B, the **`io-net`** networking manager puts itself into the background as a server. As a result, the next line in the script, a **`waitfor`**, begins to execute. Again, the resource isn't immediately available, so the **`waitfor`** sleeps. **`Io-net`** finishes without having to wait for hardware, but because the **`waitfor`** was timed poorly, the system sits idle and wastes time before starting **`ifconfig`**.

Optimized waitfor placement

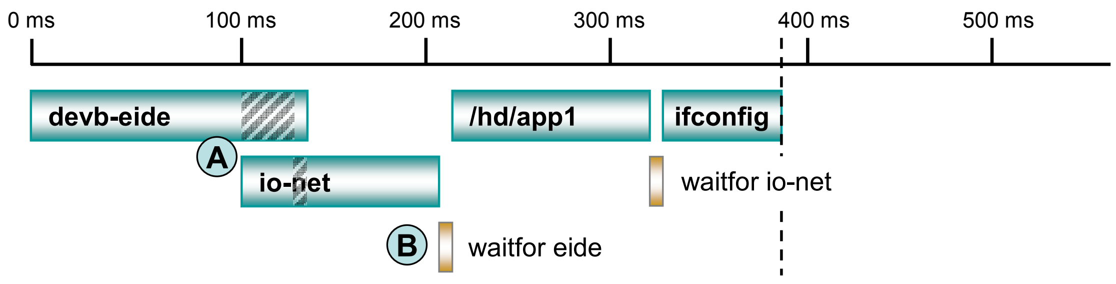


Figure 5 — Optimized `waitfor` placement. At point A, the script starts **`io-net`** to take advantage of the EIDE driver's idle time and doesn't use **`waitfor`** until it's actually needed. At point B, the system is ready to launch **`app1`**, and the **`waitfor`** completes immediately since there has been enough time for the driver to finish. **`/hd/app1`** starts slightly later than in the default waitfor scenario, but the overall boot time is significantly faster.

Microkernel versus monolithic kernel

If you're used to working with a monolithic kernel like Linux or Windows CE, you might be inclined to start all your drivers before you start any applications. With a microkernel OS like the QNX Neutrino RTOS, you have more flexibility and can reorder some of your startup to take advantage of any idle time. That includes starting applications before starting drivers, wherever it makes sense.

For instance, your system may need to play a “jingle” within a second after power on. To achieve this, you could easily reorder the startup so that a jingle playback utility runs immediately after the audio driver loads. You could also move the audio driver so that it loads before most other drivers. (The flash and serial drivers will probably start first, however.)

With this approach, audio-file playback can start while the rest of the system continues to load. In other words, you don’t do the playback in the *main*() of your normal application’s startup. The audio playback can be a small “throwaway” utility that terminates as soon as it is finished. This approach lets you load the rest of the drivers and applications while the jingle is playing.

Modular versus monolithic application design

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.

Library load time

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.

Language choice

Chances are, you wouldn’t use Java to meet early boot requirements. The Java Virtual Machine (JVM) and class libraries must load before applications can do anything, an operation that can add seconds to the boot time.

Likewise, if you use C++ and have a serious need for early boot speed, you will want to think through your design. Careful use of C++ isn't a problem. But C++ applications tend to be larger than their C counterparts, especially when they use frameworks like the Standard Template Library (STL). They can also use a lot of dynamic memory allocation, which takes time. In addition, C++ uses large shared object libraries like libcpp.so, which can contribute to longer load times.

When using C++, try to design components that need quick booting to be very small, with as few templates, frameworks, and global constructors as possible.

Establishing a baseline

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.

QNX SOFTWARE SYSTEMS

About QNX Software Systems

QNX Software Systems, a Harman International company, is the leading global provider of innovative embedded technologies, including middleware, development tools, and operating systems. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® development suite, and QNX Aviage® middleware family together provide the industry's most reliable and scalable framework for building high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for network routers, medical instruments, vehicle telematics units, security and defense systems, industrial robotics, and other mission- or life-critical applications. The company is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com