



Instant-On Technology for In-Car Telematics and Infotainment Systems

Sheridan Ethier, software engineer
Randy Martin, automotive engineering manager
QNX Software Systems
sheridan@qnx.com, randy@qnx.com

A Matter of Timing

The sophistication of in-car telematics and “infotainment” systems is growing rapidly. Already, automakers and Tier One auto suppliers are introducing devices that offer everything from dynamic navigation and realtime traffic reports to DVD playback, digital radio, voice-controlled operation, and automated 9-1-1 dialing.

To handle this complexity, systems designers are turning to full-featured, protected-mode RTOSs, deployed on 32-bit processors that provide onchip support for various automotive technologies, including the J1850, CAN, and MOST communication buses. Despite their sophistication, these systems must satisfy the same timing requirements as older hardware-software solutions. For instance, from the time that it is powered on, a telematics control unit must be able to receive CAN messages within 60 to 100 milliseconds. Problem is, the complex software running on such a device can easily take hundreds of milliseconds, or longer, to boot up.

Critical Milestones

An in-car telematics or infotainment unit typically boots from a cold condition (completely powered off) or from a CPU reboot condition (returning from a state where SDRAM has been turned off). Critical milestones during the boot process may include:

- receiving CAN messages within 30 to 100 milliseconds after power on
- responding to the above messages within 100 milliseconds of receiving them
- reading Class 2 messages from a vehicle bus and responding to wake events
- initializing a MOST transceiver and responding to MOST requests

Many of these milestones must occur within tens of milliseconds from system power on.

To address these timing requirements, many designs rely on an auxiliary communications processor or external power module — a relatively simple solution, but an expensive one. That hardware is no longer needed, however, thanks to breakthrough technology developed by engineers at QNX Software Systems. Dubbed “mini-driver technology,” the approach consists of small, highly efficient device drivers that start executing *before* the OS kernel is initialized. Defined in the system’s boot loader, a mini-driver can respond to a power-up message in a quick, timely fashion and ensure no other messages are lost while the OS boots up. Once the OS has booted, the mini-driver may continue running or it may pass control to a full-featured driver, which can then access any messages the mini-driver has buffered.

Example Application

For an example of a mini-driver application, consider an automotive telematics device connected to a CAN bus. When the car is started, the CAN bus master device will send a “Power On” message to all connected devices within 65 milliseconds; see Figure 1. The software in the telematics device must

get ready to receive, and possibly reply to, this CAN message within a somewhat shorter timeframe: 55 milliseconds or less. That’s because the CPU takes approximately 10 milliseconds to execute its first instruction after being powered on.

In addition, the telematics device must reply with a “Device Up” message within 100 milliseconds of receiving the “Power On” message; this tells the bus master that the device is operational. If the device fails to respond within this time period, the bus master may consider it nonoperational and remove it from the bus. After the initial power-on handshaking sequence, the telematics device must also buffer, and possibly reply to, any further messages that it receives over the CAN bus during the remainder of the boot sequence. As Figure 1 illustrates, these messages may arrive at a rate of 1 every 10 milliseconds.

In this scenario, it is unlikely that the operating system (OS) kernel can be fully initialized before the first “Power On” CAN message, because of bottlenecks in the booting process. These include copying the OS image into RAM, switching the CPU from physical mode to virtual mode, and waiting for clocks to stabilize. In fact, the boot time for a full-featured, protected-mode RTOS — from power-on-reset to launching the first user application — is typically measured in hundreds of milliseconds. Thus, the mini-driver must begin working during the system initialization, before the OS image is copied to RAM.

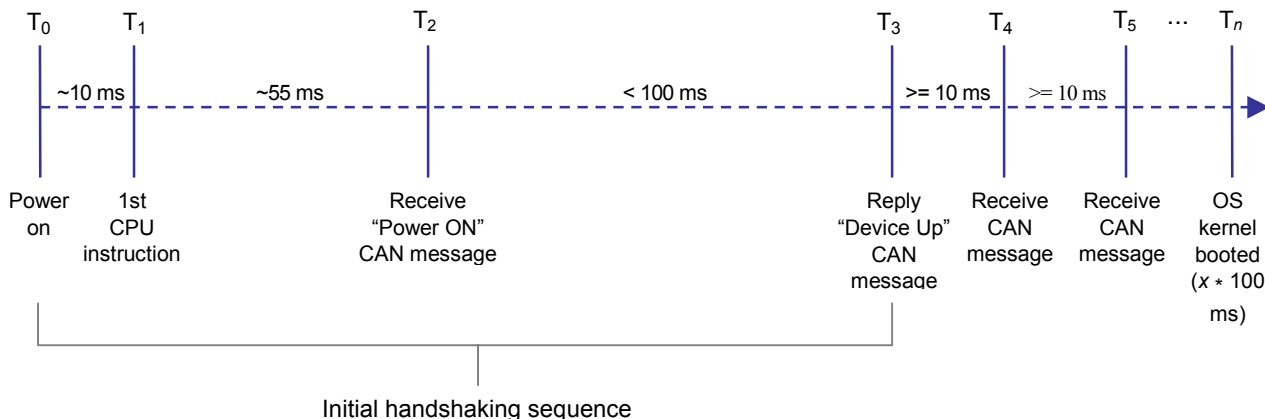


Figure 1 — Sample initialization timing sequence for a telematics system on the CAN bus.

Architecture Overview

QNX[®] mini-drivers are based on a simple concept: to make a peripheral device usable throughout the booting process. Once the peripheral is initialized, the various phases of the boot process must meet the timing constraints placed on the system by reading and possibly replying to messages received over the peripheral bus.

The mini-driver consists of two fundamental components:

- **Handler function** — Performs read-write access to the peripheral hardware and implements any other logic related to the mini-driver, such as reading, buffering, and responding to messages.
- **Message data storage** — A block of memory that allows the mini-driver handler function to store any messages it receives. These messages can subsequently be retrieved by user applications or by a full-featured device driver.

Once a mini-driver is created, its handler function is called throughout the boot process to ensure that messages aren't missed. To trigger the handler function, the system may use the following mechanisms:

- **Polling** — The system polls the mini-driver handler function at various times during the boot sequence to determine if the peripheral has received any data. To enable this polling, the boot code may make calls to the handler function at user-defined intervals.
- **Timer interrupt-based polling** — The mini-driver handler function is attached as a pseudo-interrupt handler based on a system timer, and the handler is called at a fixed interval. The mini-driver is still in polling mode, but, in this case, the system doesn't have to explicitly call the handler function throughout the boot code.
- **Device interrupt** — The device interrupt mechanism attaches the mini-driver handler function directly to the device's interrupt source. This avoids polling altogether and calls the interrupt handler only when the hardware event occurs.

A device interrupt is, in many cases, the most desirable triggering mechanism, but may not be possible in all systems or throughout the entire boot process. Consequently, a system may switch triggering mechanisms when moving from one phase of the boot process to another.

Seamless transition

Once the system has booted, a full-featured device driver can take control from the mini-driver, without losing data or incurring blackout times. The full driver simply attaches to the device interrupt, and the mini-driver exits gracefully. Moreover, the full driver can access any messages the mini-driver has buffered in its data storage, ensuring a seamless transition. (If the system doesn't need a full driver, the mini-driver can simply continue to run.)

Control Sequence at a Glance

The operation of a mini-driver can be summarized in a few simple steps:

1. The mini-driver gains control of the system at specified intervals during the initial boot sequence. This is a polled approach, as hardware interrupts aren't yet turned on.

The mini-driver can have a small protocol stack associated with it. It will have just enough logic to:

- initialize hardware
 - buffer incoming messages in its data storage
 - respond to critical events as needed
2. At some point during the boot phase, hardware interrupts are turned on. The system can then invoke the mini-driver on an interrupt basis. (Because a system timer is itself an interrupt, a polling approach can still be used, if required.)
 3. After the OS has been loaded, the mini-driver can pass control to a full-featured driver.
 4. The full-featured driver will have the complete protocol stack associated with it. Once started, the full-featured driver can read and process all the data collected by the mini-driver during the boot sequence.

With these booting techniques, a general-purpose CPU can behave much like a dedicated processor. For instance, laboratory tests of a system based on a 396MHz Freescale MPC5200 processor demonstrate that a mini-driver can easily handle a 1 millisecond message interval. In other words, if data is sent to the device every millisecond at power on, the mini-driver can successfully receive and process all the data while the system boots up. Similar results can also be achieved on processors based on architectures such as SH-4 and ARM.

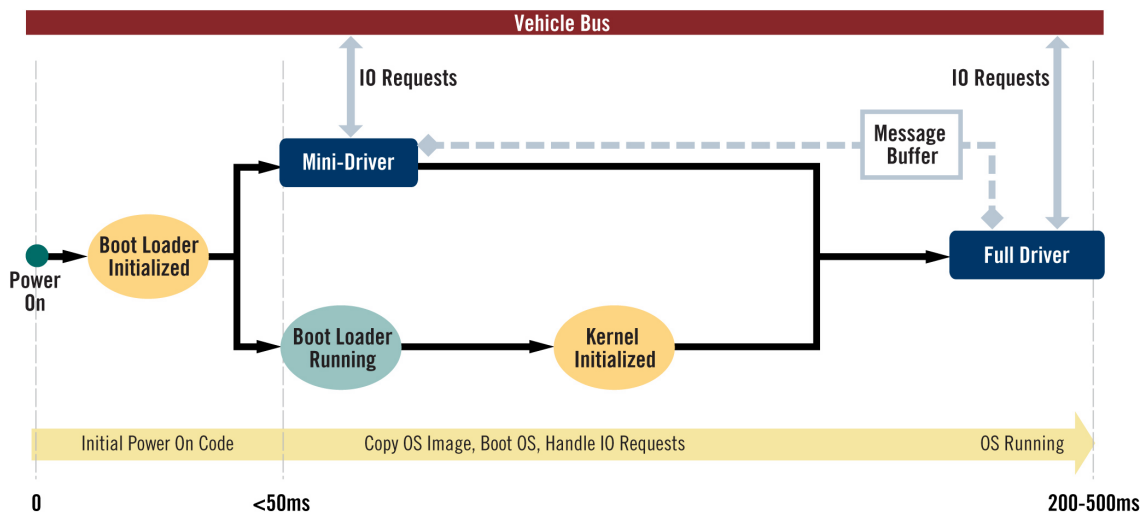


Figure 2 — Using mini-drivers, a software-rich telematics system based on the QNX Neutrino® RTOS can respond to a CAN power-on message in under 40 milliseconds, without the need for an auxiliary communications processor.



About QNX Software Systems

With millions of installations worldwide, QNX Software Systems Ltd. is the global leader in realtime, microkernel operating system technology. Companies like Cisco, DaimlerChrysler, Lockheed Martin, Panasonic, Siemens, and General Electric rely on QNX technology to build ultra-reliable systems for the networking, automotive, medical, military, and industrial automation markets. Founded in 1980, QNX Software Systems maintains offices throughout North America, Europe, and Asia.

www.qnx.com