To load a program into memory, you must first load a program into memory. The bootup process, often a complex multistep sequence involving numerous substeps, solves this problem. Any boot-up process, including booting up Windows, Linux, or an embedded RTOS (real-time operating system), begins with the application of power to the system and the subsequent removal of system reset. During POR (power-on-reset) assertion, you may have to reconfigure hardware peripherals if operational values differ from those of default settings. Embedded microcontrollers, for example, often offer various hardware-reset-configuration schemes.

Over the last several decades, booting up has evolved from a simple DOS-based step to more complicated multiple-operating-system choices or even peripheral-based boot-up techniques. A USB (Universal Serial Bus) interface, for example, allows you to boot up a disk image from an external storage device; this approach is increasingly popular in industrial- and embedded-system applications because it provides abundant flexibility. In the case of software corruption, for example, in which a system requires the reloading of new firmware, the USB technique allows a service engineer to simply copy new software onto a flash drive and boot from it. The service department therefore saves the thousands of dollars in expenses that it would otherwise incur in transporting the equipment to the manufacturer for repair.

To enable system-boot flexibility both from USB, PCIe (Peripheral Component Interconnect Express), and SDHC (secure-digital-high-capacity) interfaces and from conventional memory devices requires in-depth hardware and software capabilities. Opensource firmware—specifically, the UBoot (Universal Boot Loader) utility, which finds wide use in embedded-system platforms—may also be of value. The Linux-based boot loader can automatically boot up the operating system; alternatively, it allows a user to manually run explicit commands to start the operating system, and it supports booting from a variety of interfaces (see sidebar “The U-Boot”).

Windows XP system boot
A simple x86 boot sequence is fairly self-explanatory (Figure 1). Windows XP follows comparable steps, albeit with more sophistication. The boot sequence begins with the application of system power; the processor remains in reset. When all voltages and current levels are acceptable, the power supply sends a power-good signal to the processor. The next step, POR negation, takes place when the availability of the good-power signal negates the processor reset to allow the CPU to begin operation. The CPU points to the ROM boot address and begins executing the BIOS (basic-input/output-system) code. The ROM BIOS then performs POST (power-on self-test), a basic test of core hardware to verify basic performance. The boot-up process then reports any errors that occur at this point using beep codes because the video subsystem has not yet initialized.

The next step is video-card initialization, during which the BIOS searches for a video-card adapter by scanning memory addresses C000:0000 through C780:0000 to find a video ROM. The video test initializes and tests any discovered video adapter, potentially along with its video memory, and displays configuration information. If a “cold” start is taking place, the ROM BIOS executes a full POST. If it is a “warm” start, the boot-up process skips the memory-test portion of the POST.

The CMOS now reads from the BIOS. During this step, the BIOS locates and reads configuration information stored in the CMOS. A small coin battery cell on the motherboard typically maintains the CMOS, a small—typically, 64-byte area of memory—on the motherboard. The CMOS memory stores information such as date, time, and peripheral-boot order. If the first bootable disk is a fixed disk, the BIOS examines its first sector for an MBR (master boot record). The MBR comprises a partition table, which describes the layout of the fixed disk, and a partition-loader code, which includes instructions for continuing the boot process. The boot, or partition, loader then examines the partition table for an active partition. The partition loader searches the first sector of that partition for a boot record. The boot process then checks the active partition’s boot record for a valid boot signature. If it finds one, it executes the boot-sector code as a program.

The NTLDR (New Technology Loader) for Windows, a hidden system file that resides in the root directory of the system partition, controls the loading of Windows XP. During NTLDR’s initial phase, it moves the processor from real mode to protected mode, enabling 32-bit memory accesses and turning on memory paging. It then loads the appropriate minidrivers to allow NTLDR to load files from a partition formatted with any of the file systems that Windows XP supports, including FAT (fileallocation table)-16, FAT-32, and NTFS (New Technology File System).

If the boot-initialization file resides in the root directory, NTLDR reads its contents into memory. If the file contains entries for more than one operating system, NTLDR suspends the boot sequence, displays a menu of choices, and waits for the user to make a selection. NTLDR then continues the boot-up process by locating and loading the DOS-based New Technology executable file to perform hardware detection. After selecting a hardware configuration, NTLDR begins loading the Windows XP New Technology kernel file. During this process, the screen clears, and a series of white rectangles subsequently progresses across it. NTLDR now loads device drivers that are marked as
boot devices. Before performing this load, NTLDR relinquishes control of the computer.

At this point, the system displays a graphics screen with a status bar indicating the load status. During later initialization phases, the system cannot accept device interrupts. The I/O manager also begins to load the remainder of the system drivers, picking up where NTLDR left off. The last task for this initialization phase is to launch the session-manager subsystem, which is responsible for creating the user-mode environment. The session-manager subsystem then loads the Windows device driver, which implements the graphics subsystem. Windows XP boot is not complete until a user has successfully logged onto the system. The Windows log-in file begins the log-in process, which the kernel loads as a service, and displays the log-on dialogue box.

**Embedded-system reset**

Older microcontrollers had a single fixed configuration state for the entire register suite after reset deassertion. This situation translated into fixed values for parameters, such as clock-speed configuration, start-address location, pad slew, drive strength, external-memory-port size, and peripheral enable/disable state. It enforced restrictions on the way users employed chips after reset deassertion. For example, if the system disabled an on-chip oscillator providing a clock to off-chip peripherals at reset, those peripherals would be able to work only after software re-enabled the oscillator.

In simpler systems, such behavior might be acceptable, but it may not meet requirements if that same chip is finding use in more complex applications that require different configurations during reset. To provide flexibility to a configuration of registers after reset, you can implement various microcontroller-based design schemes. Some microcontrollers also support multiconfiguration schemes, in which the system selects any configuration by reading the state of pins during reset. Here we discuss four reset configurations: default, fuse programming, external pins, and serial interface.

**Figure 2** Loading default values during boot-up is the most common reset configuration.

Loading default values during bootup is the most common reset configuration, and it requires no special onboard setup (**Figure 2**). Conversely, it provides no flexibility or options to configure any register. All the registers initialize to fixed values; thus, the chip exits reset in only one fixed state. This approach is the fastest for initializing the system before the boot process, but it is the least powerful mode in terms of capabilities for controlling the system state. It might work for some applications but is less preferable if you use the same microcontroller in varying applications with varying boot requirements.

A system POR asserts internal chip reset, with both signals being active low; when deasserted, it restarts the clock and loads the reset configuration in the system registers. Depending on the system design, other tasks might gate the boot-up process. For example, the system might wait for a PLL (phase-locked loop) to achieve clock locking before the internal reset deasserts and the system begins executing the boot code.
Fuse programming involves a reset configuration that is the result of programming through a chip’s special test mode, fuses, or on-chip nonvolatile flash registers (Figure 3). This mode implements special bits and registers through either on-chip fuses or an array of nonvolatile flash-memory registers for configuring reset-control-word information. These registers require write-once capability; therefore, you can program them only once in a chip’s lifetime. This mode usually requires a hardware-setup or software sequence to program these special registers or fuses. Once fuses are programmed and reset is asserted, reset-control-word information is read from the fuses and copied to the desired system registers. The system then internally deasserts the reset and begins executing code.

This scheme provides the flexibility to configure different system-register options but requires the implementation of special fuse registers in the design. Because the fuses are one-time programmable and secure, they can effectively enable and disable functions within the chip, thereby creating “phantom” parts with lower prices. This strategy is common with semiconductor vendors, which sell the same sliver of silicon with different costs and features—achieved by blowing the fuses to enable or disable functions.

You can also reset the configuration through external pins. This scheme uses a group of microcontroller pins to control the reset configuration. These pins are externally pulled high or low during reset to define a configuration option. Once the system reset deasserts, the microcontroller internally latches these values and decodes them to configure the system registers. This scheme provides limited flexibility in selecting control-word configurations. The available number of configurations is directly proportional to the number of pins for this purpose.

Engineers often implement the design by means of an external buffer or a line driver, such as the 74LVC125, which drives either a logic one or a logic zero to the pins for reset configuration (Figure 4). The reset signal usually connects to the enable pin of a tristate buffer—that is, an external line driver—so any change from one to zero or vice versa to the input of the tristate buffer appears as an input to the pins that eventually handle the reset configuration. This approach provides additional flexibility and control. For example, if one of the buffers controls whether PLL is enabled or disabled during boot-up, a buffer output can allow users to enable the PLL when the buffer connects to the drain-to-drain voltage and conversely disable it when the buffer connects to the source-to-source voltage. The number of pins available for this purpose usually constrains this approach. Note, too, that most external line drivers, such as that of the 74LVC125, integrate groups of four or eight buffers. To limit the implementation’s cost, define the buffer numbers in multiples of four.
You can also load the reset configuration through an external serial interface. For a highly integrated complex microprocessor, it is impractical to either dedicate or share pins for the numerous available power-up options. This scheme conversely involves loading the chip-reset-configuration data from an external serial memory (Figure 5). In a typical implementation flow, when system reset asserts, the chip establishes communication with the serial memory, subsequently transferring reset-configuration information to the microcontroller. Upon serial-data reception, the microcontroller configures system registers based on the received data and deasserts reset. This method provides the maximum flexibility in configuring options in system registers because serial memories can store a large number of data bytes.

In advanced serial-configuration schemes, the serial memory can even hold software code. In such cases, the system reads both reset-configuration data and boot code from external serial memory during the microprocessor reset sequence, thereby requiring few I/O pins. By reading data stored in, for example, external SPI (serial-peripheral-interface) memory, the system would also need to configure the SPI memory’s clock frequency along with the powerup options for the microprocessor and optionally load code into the microprocessor’s memory. The system would have to accomplish all these tasks before the negation of device reset, thereby ensuring that the chip is properly configured when exiting the reset state. Serial memories’ low cost, simple implementation, high flexibility, and optional software-boot code often make this option the preferred one for booting or loading the reset configuration.

System-boot components

Boot components are primary or secondary devices, depending on their ability to support boot immediately after reset deassertion. The primary option provides a direct-boot capability and facilitates the processor’s first instruction fetch from the memory location in which software-initialization code resides. You should enable the primary interfaces immediately after reset and can sometimes configure them through the reset option. You can place boot loaders or small initialization-code sequences in these locations. You initialize and configure secondary interfaces in these primary boot-code sequences and boot loaders. They mainly hold the large operating-system kernel’s code, which loads after the boot loader performs basic initialization. You can execute the operating-system kernel directly from these interfaces or copy it in processor-accessible memory (Figure 6). For example, an on-chip ROM component can act as the primary boot device, including initial initialization code, which subsequently switches execution to a
secondary option, such as external DDR SDRAM, which in turn holds the complete operating-system kernel.

Common hardware-boot components allow a system to boot from a variety of interfaces (Figure 7). Booting from internal flash memory is among the most common and simplest methods of configuring an embedded microcontroller that includes the necessary on-chip resources. This method reduces dependencies on external interfaces because the boot loader resides in on-chip memory. After system-reset deassertion, the processor points to the flash memory’s starting address and loads the necessary initialization code and operating systems. Operating systems with small footprints are compatible with this approach because a practical limit exists on the amount of on-chip flash memory that can be available. This approach is also one of most secure ways to boot a processor because modifying code residing in on-chip memory requires fewer changes than do off-chip boot options.

As with Windows XP, some microcontrollers integrate ROM as the primary boot option. The boot ROM includes a basic boot loader so that the microcontroller can subsequently perform a more sophisticated boot sequence on its own, loading programs from various sources, such as Ethernet, NAND-flash memory, an SD (secure-digital) card, an MMC (multimedia card), or a USB interface. Boot-ROM usage enables more flexible boot sequences than does hard-wired logic, and it allows users the choice to boot up from various peripherals. Users often employ the boot-ROM feature for system recovery when someone inadvertently erases the usual boot software in nonvolatile memory other than ROM. You cannot reprogram boot ROM, so applications that require a secure boot may include security checks so that the boot-up stops if one or multiple security checks fails.

An external bus interface allows the system to boot directly from external NOR flash or other parallel memories. It is one of the fastest ways to boot the system because the interface to external memory can be 32 bits or more with a reasonable frequency of operation. For a full-fledged operating system such as Linux, or Windows, it can take several milliseconds to seconds to boot the system due to the size of the operating system, which can be annoying to the user. Keeping the boot loader/operating system in external parallel memory reduces the boot-up time drastically for systems in which boot time is critical.

NAND-flash memories are gaining popularity in numerous applications due to their higher read throughput, although this throughput is lower than that of NOR-flash memory. NAND flash also offers faster erases and lower cost per byte than does NOR flash. The primary use for NAND-flash memory—for example, in a USB flash drive—is to store large quantities of data and code. However, in recent years, an increasing number of embedded-system applications also support NAND-flash memory as a primary boot option. In these cases, the microcontroller must include a NAND-flash memory controller to handle read and write accesses. The NAND-flash-memory interface also requires more than 20 pins, so this option may not be cost-effective and otherwise practical unless you use a package with more pins.

Booting from internal volatile memory is always a secondary boot technique because a primary boot
device must load the RAM before it can execute code. Commonly, a primary boot loads the operating system and drivers, copying them to RAM. At this point, the RAM-based code takes control of the system. Executing code from RAM is faster and consumes less power than do other memory technologies, including either external or internal flash memory. However, because the internal RAM is volatile, some system designs enable the RAM to switch to battery power in the event of the failure of the primary supply, thereby eliminating any further need to copy the code from external or internal memory when the primary power returns and thus reducing subsequent initialization and boot time.

Booting from a DRAM is also a secondary boot technique. DRAM use is common in high-end applications that must handle abundant multimedia content and that require high throughput. You can view DRAM as a bigger and faster extended RAM buffer for managing complicated applications. In one example, ROM includes the boot loader, and NAND-flash memory contains the operating system and the application code (Figure 8). The boot process begins with system initialization through the ROM-boot loader, which also includes the reset vector. The main operating system code copies from NAND-flash memory to DDR SDRAM, subsequently switching execution control to the external volatile memory. This scheme is efficient because executing from DDR SDRAM is faster than reading directly from NAND-flash memory. You can use similar schemes to copy code from other interfaces, such as Ethernet.

You may want to boot from various interfaces, such as SDHC, SPI, I²C (inter-integrated circuit), USB, SATA (serial advanced-technology attachment), PCIe, and Ethernet. These interfaces all represent secondary boots. A primary boot interface, such as ROM, initializes a secondary boot interface, such as USB, before code execution switches to the secondary boot device. Storing boot code in external nonvolatile serial memories, such as SPI flash memory or I²C EEPROM, can be useful for microcontrollers that have low pin counts and can afford to have longer boot-ups. Such schemes first copy the boot code from the external memory to the onchip RAM and code execution switches to the RAM after reset, so the scheme can immediately fetch boot code after reset deassertion.