Click to prove you're human

Embedded systems examples

Embedded systems are ubiquitous. These dedicated small computers are present in communications systems, vehicles, manufacturing machinery, detection systems, and many machines that make our lives easier. The open nature of Android Linux and its availability for many different hardware architectures makes it an excellent candidate for embedded platforms. The following are the most common concepts you should know while working with embedded devices. Bootloader A bootloader is a small piece of software that executes soon after you power up a computer. On a desktop PC, the bootloader resides on the master boot record (MBR) of the hard drive, and is executed after the PC BIOS performs various system initializations. The bootloader then passes system information to the kernel (for instance, the hard drive partition to mount as root) and then executes the kernel. In an embedded system, the role of the bootloader is more complicated, since an embedded system does not have a BIOS to perform the initial system configuration. The low-level initialization of the microprocessor, memory controllers, and other board-specific hardware varies from board to board and CPU to CPU. These initializations must be performed before a kernel image can execute. At a minimum, a bootloader for an embedded system performs the following functions: Initializes the hardware, especially the memory controller. Provides boot parameters for the operating system image. Starts the operating system image. Additionally, most bootloaders also provide convenient features that simplify development and update of the firmware, such as: Reading and writing arbitrary memory locations. Uploading new binary images to the board's RAM via a serial line or Ethernet. Copying binary images from RAM to Flash memory. Kernel The kernel is the fundamental part of an operating system. It is responsible for managing the resources and the communication between hardware and software components. The kernel offers hardware abstraction to the applications and provides secure access to the system memory. It also includes an interrupt handler that handles all requests or completed I/O operations. Kernel modules Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without requiring a system reboot. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without these modules, Linux developers would have to build monolithic kernels and add new functionality directly into the kernel image. The result would be a large, cumbersome kernel. Another disadvantage of working without a kernel module is that you would have to rebuild and reboot the kernel every time you add new functionality. In embedded systems, where functionality can be activated depending on the needs, kernel modules become a very effective way of adding features without enlarging the kernel image size. Root file system Operating systems normally rely on a set of files and directories. The root file system is the top of the hierarchical file tree. It contains the files and directories critical for system operation, including the device directory and programs for booting the system. The root file system also contains mount points where other file systems can be mounted to connect to the root file system hierarchy. Applications Software applications are programs that employ the capabilities and resources of a computer to do a particular task. Applications make use of hardware devices by communicating with device drivers, which are part of the kernel. Cross-compilation If you generate code for an embedded target on a development system with a different microprocessor architecture, you need a cross-development environment. A cross-development compiler is one that executes in the development system (for example, an x86 PC), but generates code that executes in a different processor (for example, if the target is ARM).