CHAPTER 8

EMBEDDED SYSTEM COMPONENTS

In this chapter

- Introduction
- Hardware Components
- Firmware Components
- RTOS System Software
- Software Application Components

8.1 Introduction

System design can be approached in a bottom-up or a top-down fashion. The *bottom-up approach* consists of determining the fundamental components that go into a system. The *top-down approach* is a hierarchical breakdown of the system into subsystems and then into components. The top-down approach can be viewed as a concrete breakdown of the system into smaller parts as suggested here, but often an abstract top-down approach is useful where the system is broken down by service and function. This functional top-down approach is described in Chapter 15, "System Life Cycle." In this chapter, we first examine common components of a real-time embedded system in the concrete sense, going from the overall system down to components. Familiarity of the components can assist the designer in making more optimal system design decisions. The design of a real-time embedded system can be viewed as a hierarchy of subsystems, as depicted in Figure 8.1, showing a real-time stereo-vision tracking system.

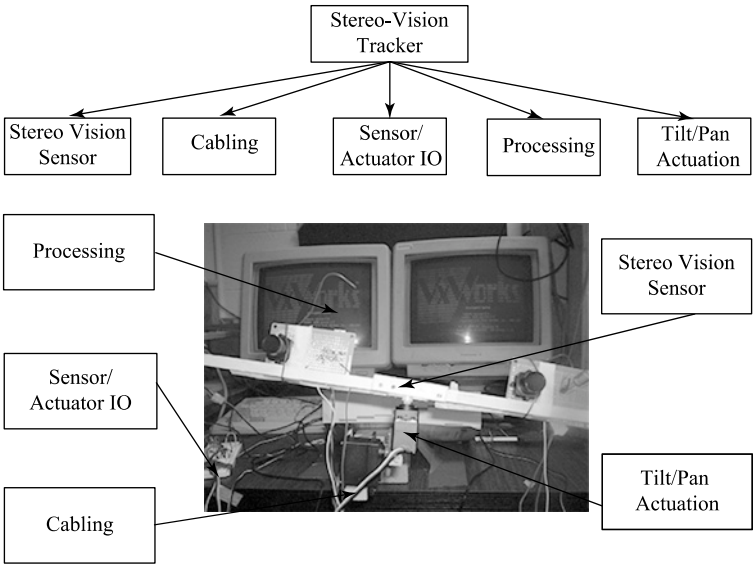

FIGURE 8.1 Subsystems in a Stereo-Vision Tracking System

The stereo-vision tracking system has a simple goal—keep a bright object in the field of view of both cameras even if the object moves and estimate the distance from the camera assembly to the object. A more functional service view of the same stereo-vision tracking system would look much different. This is depicted by the hierarchy for the same system shown in Figure 8.2.

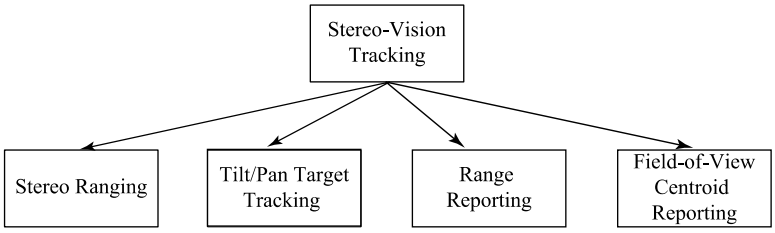

FIGURE 8.2 Services in a Stereo-Vision Tracking System

The stereo-vision tracking system is an example design that is examined in more detail in Chapter 18, "Computer Vision Applications."

8.2 Hardware Components

The hardware components of a real-time embedded system will include a wide range of components that are mechanical and electrical. For example, in the stereo-vision tracking system we have:

- Structural and mechanical—camera assembly
- Electromechanical actuators—tilt and pan servos
- Electromechanical sensors (transducers)—none, but servo position sensors could be added
- Optical sensors—NTSC cameras
- Cabling—power, NTSC signal, RS232, CAT-5 twisted pairs
- Digital state machines, microcontrollers, and microprocessors—x86 microprocessor, PIC microcontroller
- Analog front-end (sensor) and back-end (actuator) circuits—NTSC, TTL pulse-width modulation
- Networks or bus interfaces—RS232 serial, Ethernet, PCI
- Thermal management—CPU fans

Typically additional test equipment hardware may also require including monitors, development computing environment, oscilloscope, digital multimeter, and a logic analyzer. This, however, is not part of the system, although required to fully implement and verify its proper implementation and operation.

In the following sections, basic hardware components, such as those used in the stereo-vision tracking system, are described.

8.2.1 Sensors

Sensors are devices that respond to physical stimulus (light, heat, pressure, stress/strain, acceleration, magnetism) by transforming the associated energy into electrical energy or by modifying the electrical properties in a circuit. For example, a camera is a sensor that converts photon energy into electrical charge that represents the photon flux for each picture element in an array. A thermistor is a resistor circuit where the resistance of the thermistor changes with temperature, and therefore so does the circuit current at

a given voltage and voltage drop across the load. A sensor assembly may also interface this analog front end to a digital encoding interface. Analog-to-digital converters (ADCs) are used to sample and hold charge, thereby converting the analog circuit current/voltage into a digital value. For example, an 8-bit ADC will encode a sensor circuit operational voltage range into 256 levels. Without encoding, sensors are useful in analog control systems, but for use in digital control systems, encoding is critical. Real-time embedded systems therefore require digital encoding of all sensor inputs, with the exception of subsystems, which are all analog.

The sensor AFE (Analog Front End) involves physics that are very particular to the environmental phenomena being sensed and the method for generating measurable changes in an analog circuit based upon physical stimulus. Many sensors are electromechanical devices where mechanical stimulus, such as stress/strain, the force per unit area and resulting deformation, or motion, causes a change in analog circuit voltage/current. Resistance in many materials is a function of stress, strain, and/or temperature; thus these mechanical properties can be measured using the right material as a resistor in a circuit in the AFE. Motion can be sensed also with a variation of resistance through potentiometers, where resistance is modified by mechanically varying the resistive path in a circuit. A simple example is the use of a multi-turn potentiometer to modify resistance in a circuit with rotation. The sensor couples a physical phenomenon to an electrical one. This coupling may be more erudite, as is the case with an optical encoder that uses periodic interruption of an optical-coupler (LED and photodiode) in a circuit through mechanical mechanisms, such as a filter wheel. In this case the optical-coupler interruptions are counted to estimate rotation.

The sensor always includes the AFE, but may also include the analog-to-digital encoding as well. In the case of an NTSC (National Television Standards Committee) camera, the camera outputs an analog signal that encodes photo-intensity in an image field of view in an analog raster output. The NTSC analog signal can be further encoded from the NTSC signal into a digital image, which is an array of alpha-RGB (Red, Green, Blue) pixels that indicate luminance and chrominance of subareas of the camera's field of view—picture element or pixel alpha-RGB values encoded using an ADC. This is the approach taken in the stereo-vision example. An alternative might employ a CCD (Charge Coupled Device) camera, which provides a more direct encoding of photo-intensity (photon flux) in terms of electrical charge. The range of methods used by sensors to encode the wide

range of physical phenomena and associated energy into electrical energy is too broad to comprehensively discuss in this text. The key concept, however, is that all sensors do convert physical stimulus into electrical outputs that affect an analog circuit, which in turn can be encoded into a digital input using an ADC. The ADC implementation has significant impact on the encoding capability, including the following:

- Sampling frequency
- Sample accuracy
- Input range

The ADC takes an analog input (voltage or current) and converts it into a digital word, most often from 8-bit to 16-bit. The ADC requires a reference voltage, V_{ref}, and normally encodes all inputs into a range of values from zero to: $\frac{V_{ref}}{(2^n)}$, so that for a 5V reference, a typical 16-bit ADC can encode the voltage in an AFE into increments representing a change of 0.0763 millivolts, with an input range of 0 to 5V for the AFE signal. The resolution can clearly be increased by reducing the reference voltage V_{ref}; however, this is at the cost of constraining the input range. This trade-off drives the selection of how many bits the ADC provides in the encoding—to accommodate large input ranges and high resolution, the ADC must have more bits for the encoding. The final question is, how fast can the AFE be sampled? This depends upon the type of ADC:

- Flash—using comparators, one per voltage step, and resistors
- Successive approximation—comparators and counting logic

The flash ADC conversion speed is the sum of the comparator delays and logic delay—typically flash ADCs are the fastest variety. The successive approximation ADC uses comparators to determine first whether the input is greater than half the reference, then whether it's greater than one quarter, and so on until the LSB (Least Significant Bit) comparison is made and the signal level has been approximated successively to the bit accuracy of the ADC—this takes as many clock cycles as the ADC has bits. One more issue with any ADC is how the input signal is sampled—if it changes significantly during the ADC process, then the results will not be accurate, so ADCs must sample and hold the input. The sample and hold time will

add latency and thus reduce the maximum inter-sample frequency. Furthermore, the ADC may automatically sample and provide an interrupt or FIFO input to a state machine or microprocessor, or the ADC may require commands to sample the input and then convert it. For high-rate encoding, such as video, a dedicated hardware state machine typically provides the ADC control and stores encoded data in a FIFO for transfer to a microprocessor via DMA. Methods for transferring encoded data are discussed in more detail in the "Firmware Components" section.

8.2.2 Actuators

Fundamentally, an *actuator* is a transducer that converts electrical energy into some other form, such as sound, motion, heat, or electromagnetism. The simplest form of actuation is switching. The relay provides a mechanism that can be actuated to open or close a switch on command from a digital IO interface. This on/off control does not provide continuous output or simple variation of output amplitude over time. A *servomechanism*, or *servo*, is an actuator that converts electrical energy into mechanical rotation, using a motor and a control interface. Heating elements that are simple resistors can be modulated to provide heat for a system that requires minimum operating temperatures. Likewise, for systems that require active cooling, actuator subsystems can provide cooling using fans, louvers, or some other form of conductive, convective, or radiative cooling. Digital values are decoded into analog signals through an analog back end (ABE) for actuation so that a digitally encoded value drives the voltage in the ABE circuit. This is most often done using either PWM (Pulse-Width Modulation) or a DAC (Digital-to-Analog Converter) so that the amplitude in the ABE can be driven by a stream of digital encoded outputs. With PWM, a periodic digital pulse (e.g., TTL logic level) is driven out with a duty cycle that is proportional to the desired amplitude of the signal at a given point in time. The DAC provides the proportional output automatically based upon the last commanded digital output rather than decoding using a digital duty cycle. Much like ADC sensor interfaces, DAC actuator interfaces should be characterized by the following:

- Type of actuation—on/off or DAC/PWM modulated
- Speed of actuation
- Accuracy of modulation

Most often for accurate high-rate actuators, a DAC is required rather than relays or PWM. For audio output, PCM (Pulse Code Modulation) is used for input sampling and driving an output DAC for duration and at variable output levels.

Actuators can be very unstable and suffer overshoot or failure to settle without careful design and potential feedback from sensors. For example, a scanning mirror can be used to move an optical field of view very accurately by deflecting a pickoff mirror on an optical path through a small angle. An electromechanical mechanism known as a voice-coil flexure can be driven by a DAC so that electromagnetic coils are used to deflect a mirror on a rubber flexure to the left or right; furthermore, the mirror can be restored to a previous position by allowing the flexure to spring back, dampened by the electromagnetic coils. Some of the high-rate feedback control for such an actuator might be implemented as a traditional analog control circuit rather than relying upon the digital real-time embedded system to provide such control.

8.2.3 IO Interfaces

The IO in general to and from a real-time embedded system can be classified first as either analog or digital. In the case of analog IO, as seen in the previous section, an ADC is required to encode analog inputs and a DAC, PWM, or relay interface is required to decode digital outputs when analog IO is interfaced to a real-time embedded system. Many embedded systems may actually be subsystems in a much larger system and therefore may not actually have direct analog IO—instead, many real-time embedded systems have digital IO only or in addition to analog IO. Either way, at some point, all IO becomes digital once encoded or prior to decode. So, prior to the ABE or after the AFE, the embedded system simply sees digital IO. The form of the digital IO, however, can vary significantly and can be characterized as:

- Word-at-a-time IO
- Block IO

Furthermore, the method of interfacing word or block IO can be:

- Memory Mapped IO
- Port IO

In the case of word IO, a simple set of registers defines the interface to the AFE/ABE and provides status, data, and control for encode/decode. A single word is written to an ABE for output to a data register and the output started by setting control bits and output status monitored using the status register. Likewise, for an AFE interface, status can be polled to determine when new encoded data is available and samples commanded through control and monitored via status. The word IO interfaces require significant interaction with the real-time embedded system and are not very efficient, but do provide simple low-rate IO interfaces. These interfaces may require programmed IO where a CPU is involved in each input and output for all phases of the read/write, status monitoring, and control. This is often not desirable for higher-rate interfaces, where even powerful CPUs would spend way too many cycles on programmed IO and not enough on processing to provide services. So, most high-rate interfaces have a block IO interface where a state machine or DMA engine provides command and control of the word encoding/decoding and less frequently interrupts the CPU when significantly large blocks of data have been encoded/decoded—typically 1024, 2048, or 4096 byte blocks.

Processor cores traditionally have provided IO through dedicated pins from the CPU to other devices called IO ports. An alternative is to save on off-chip interface pins by memory mapping IO onto existing address and data lines in/out of the CPU so that IO causes devices to be read or written in the same address space as memory devices. Many processors, in fact, provide both port IO and memory mapped IO, such as the Intel x86. When devices are memory mapped, care must be taken to ensure that the MMU (Memory Management Unit) is aware that particular address ranges are being used for device IO so that output data is not cached, so that writes are fully drained to the device rather than buffered when needed, and so that the address range is allowed to be accessed without exception. Memory locations can be cached, and often it is not necessary for the CPU to wait for writes to be updated in actual memory devices once writes have been queued—for device IO, normally all writes should be fully drained and not cached so that actuation is reliable. If, for example, an output to a DAC was cached for later write-back and this output was driving isochronal speaker output, this would cause an actuation dropout.

Figure 8.3 shows the components of the stereo-vision sensor subsystem for the stereo-vision tracker. This sensor device consists of two NTSC (National Television Standards Committee) cameras and two PCI frame

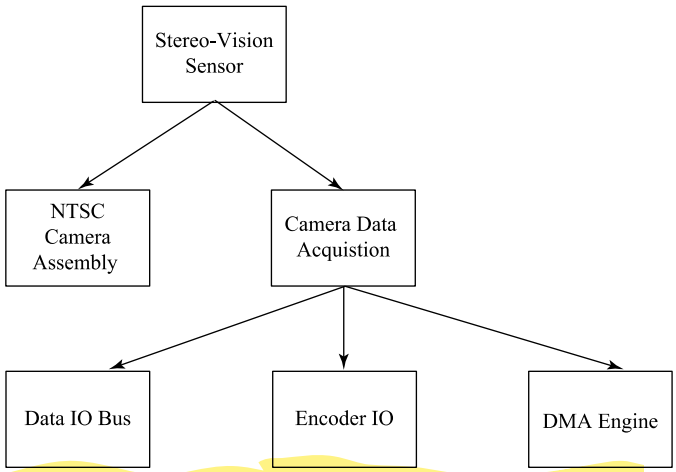

FIGURE 8.3 NTSC Vision Subsystem in Stereo-Vision Tracking System

grabbers, which acquire and encode the NTSC camera output. The data acquisition is composed of an NTSC signal encoding interface, a PCI bus data IO DMA channel, and a programmable DMA engine. The encoding is performed at 30 frames per second for a selection of video-encoding formats, including the maximum resolution of 640x480 32-bit pixels, where each pixel is composed of an 8-bit intensity, alpha, and three 8-bit fields encoding RGB (Red, Green, Blue). The AFE for an NTSC encoder uses a PLL (Phase Lock Loop) to synchronize with the NTSC signal in order to sample and digitize the signal to form a YCrCb (Y=luminance, CrCb=red and blue chrominance). The color NTSC signal format was an enhancement to the original grayscale television signal format. The basic NTSC signal format includes 525 horizontal traces to illuminate a phosphor screen, with 262.5 even scan lines and 262.5 odd with blanking time for signal retracing. The intensity of the tracing beam is modulated during the even/odd interlaced line tracing such that each pixel is illuminated for 125 nanoseconds for 427 pixels/line and 10 microseconds of blanking between lines, yielding a scan line time of 63.6 microseconds. The NTSC camera produces a signal conforming to this NTSC standard for direct input into a standard television monitor. The interlacing of horizontal scan lines—that is, tracing odd lines followed by even lines tracing each frame—reduces flicker at the NTSC frame rate of approximately 30 fps (29.97 actual). The AFE PLL synchronizes with the scan line by detecting the NTSC sync and blanking levels and then programs the ADCs to sample the signal for each pixel to

encode YCrCb. The YCrCb data is latched into an internal FIFO memory, and a synchronized DMA engine drains the FIFO with PCI bus transfers from the encoder to host system memory. The YCrCb format for NTSC was chosen so that grayscale televisions can display color NTSC signals by simply using the luminance portion of the signal alone. Most encoders support automatic conversion from YCrCb into alpha-RGB using linear scaling formulae based upon characteristics of human vision:

$$R = Y + Cr$$
$$G = Y - 0.51 \times Cr - 0.186 \times Cb$$
$$B = Y + Cb$$

The relationship between the YCrCb and RGB signals are:

$$Y = 0.3R + 0.59G + 0.11B$$
$$Cr = R - Y = R - (0.3R + 0.59G + 0.11B)$$
$$Cb = B - Y = B - (0.3R + 0.59G + 0.11B)$$

The DMA engine is a simple processor with an RISC instruction set that provides control over the encoding as well as the PCI DMA transfer and generation of host interrupts. So, for example, microcode can be written to encode the 525 NTSC input even and odd lines with 427 pixels, each into a range of formats (e.g., 320x240 alpha-RGB or 80x60 grayscale) based upon the ADC sampling rates with instructions to transfer the encoded data and to generate an interrupt at the completion of each frame encoded and transferred over PCI. The 320x240 alpha-RGB frames (307,200 bytes/frame) are transferred by the DMA engine using multiple PCI bus bursts, typically 512 bytes to 4K each. This is typical of a high-rate block transfer IO interface for a high data rate sensor. In the case of this example, two encoders are bursting video data on the PCI bus simultaneously to two different DMA buffers in two different host memory address ranges.

The stereo-vision tracker also incorporates a low-rate actuation IO interface to enable the system to tilt and pan the stereo-vision sensor (cameras and baseline mount) to follow a bright target that may be moving in order to keep the target in both camera fields of view. Figure 8.4 describes the components making up this low-rate actuation subsystem.

The tilt/pan actuation subsystem uses two servos to provide the tilt and pan rotational degrees of freedom. The servos are commanded with a TTL PWM signal (Servo PWM IO) generated by a microcontroller (Servo Controller). The specific signal generated, and thus position of the servo, can

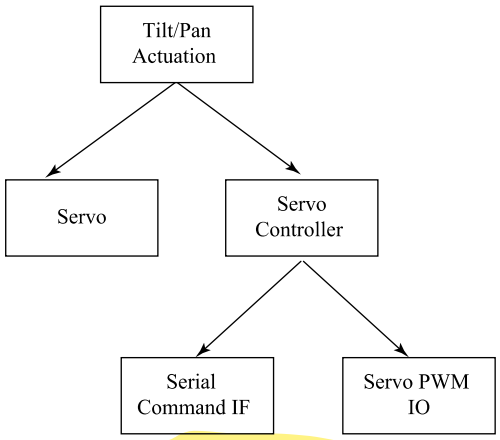

FIGURE 8.4 Tilt/Pan Servo Subsystem in Stereo-Vision Tracking System

be commanded by a microprocessor through a multi-drop serial interface (Serial Command IF). The serial command interface is simple and allows the microprocessor to write out command data only 1 byte at a time. This is a typical low-rate interface.

8.2.4 Processor Complex or SoC

Almost all modern real-time embedded systems include a general-purpose CPU to process firmware/software to provide updateable and flexible services by processing and linking sensor inputs to actuator outputs. If the services that a real-time embedded system must provide are so well known that they can be fully committed to a hardware state machine, then perhaps a processor complex (or set of interconnected CPUs or CPU cores) is not needed. Most often services are expected to change over time or are not well enough specified initially or way too complex to consider hardware-only implementations. The processor complex may be composed of the following:

- A single CPU with port IO and bus interface MMIO
- Multiple CPUs on an internal bus with port/MMIO
- Multiple CPUs with an interconnection network and port/MMIO
- An SoC (System on a Chip) with multiple CPU cores interconnected on-chip with memory, IO, flash, and any number of peripherals making it a single-chip solution

In the case of our working example, the stereo-vision system, a main x86 CPU provides an image processing platform to compute the centroid of the target object as seen by the left and right cameras and encoded using the PCI-bus NTSC encoder subsystem. The servo control is achieved using the Servo Controller, a Microchip PIC that commands multiple servos to tilt/pan the camera assembly using TTL logic-level PWM based upon a serial byte stream command to the controller. Figure 8.5 shows the subsystems (Servo Control and Image Processing) that compose the overall stereo-vision system processing to provide the tracking and ranging services. The Servo Control subsystem uses a digital control law based upon calculated centroid inputs to tilt/pan the stereo-vision sensors in real time to keep the target in the field of view and produces a series of servo commands as output. The Image Processing subsystem uses alpha-RGB video frames at a maximum rate of 30 fps to compute the centroid of the target as seen by each camera and the range to the target based upon a triangulation calculation.

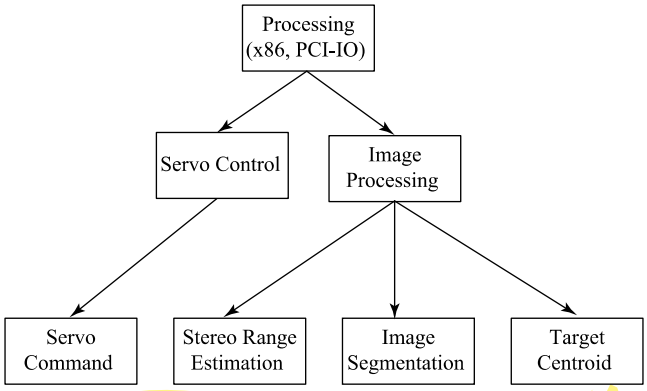

FIGURE 8.5 Processing Subsystem in Stereo-Vision Tracking System

8.2.5 Processor and IO Interconnection

For multi-CPU real-time embedded systems, an interconnection network is required to enable IO and processing to be distributed. The interconnection network can be

- Simple bus or back-plane (e.g., PCI or VME)
- On-chip local bus with bus interface unit to back-plane IO bus
- A crossbar on-chip interconnection between CPUs
- An off-board network—for example, firewire, USB, Ethernet

Figure 8.6 shows taxonomy for interconnection strategies that can be used to integrate CPUs and IO interfaces in an embedded system.

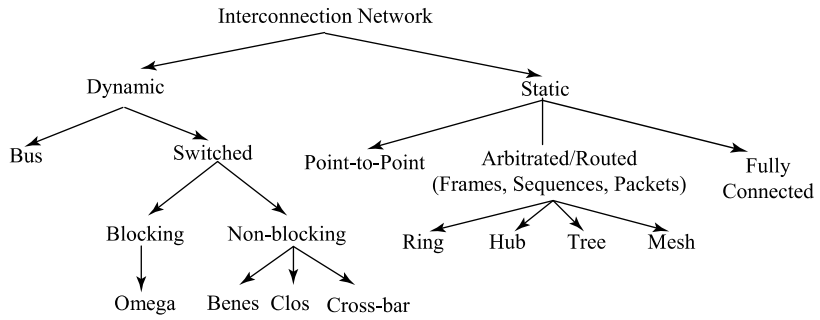

FIGURE 8.6 Taxonomy of Processor-IO Interconnection Strategies

Most embedded systems are integrated with a scalable bus architecture, point-to-point serial links, or networks.

8.2.6 Bus Interconnection

Many different bus architectures have been used and are being used for embedded systems. To better understand integration of processing and IO with a bus interconnection, this chapter will examine the VME (Versa Module Extension) bus and the PCI (Peripheral Component Interconnect) bus. The VME bus has historically been a popular and simple bus architecture used with embedded systems, and, by contrast, PCI is an emergent bus architecture that offers traditional parallel bus integration as well as high-speed serial interconnection with PCI Express.

The PCI bus, introduced as a replacement for the ISA (Industry Standard Architecture) bus prevalent in the desktop PC domain, has evolved and become a popular IO to processor complex integration method in embedded systems. A goal of the first PCI standard, 2.1, was to provide a bus where IO adapters could be interfaced to processor complexes with plug-and-play integration. Plug and play provides a standard for PCI controllers (masters) to probe the bus and find devices after they have been added without any modification to the master interface. The PCI bus was designed to integrate with the legacy ISA bus through an interface called the South Bus. The main PCI controller interface was called the North Bus. At the time that PCI was first introduced, many embedded systems were integrated using the VME bus (Versa Local Bus Module Expansion). Building a real-time embedded system based upon bus integration allows system designers to decompose

the system into subsystems with interface and processor boards that can be designed, built, and tested as units and later integrated on a standard interconnection. Both VME and PCI provide this modularity compared to custom back-plane or onboard integration. The bus integration also provides a fast signal interface compared to packet or frame transmission networks (this has started to change recently, as we'll see later). Table 8.1 briefly summarizes both VME-32 and PCI 2.x, comparing features.

TABLE 8.1 Comparison of PCI and VME Buses

Feature	VME bus	PCI 2.x bus
Bus transfers	Asynchronous 20 MHz	Synch clock 33/66 MHz
Target addressing	32, 24, or 16 bit address (A32, A24, A16)	Multiplexed 32/64 bit address/data bus
Data transfer	32, 24, or 16 bit separate data bus	Multiplexed with 32 bit address bus
Data transfer types	Word or block transfer limited to a specific block size (e.g., 512 bytes)	Burst transfer always with min and max length
Device interrupt mechanism	Daisy-chained priority interrupts	4 shared interrupt lines: A-D routed to programmable interrupt controller
Interrupt vectoring	Interrupt data cycle following interrupt level	Map A-D onto processor vector—e.g., onto IRQ 0…15 on x86 with direct IRQ to vector mapping
Bus access arbitration for multiple initiators (masters)	No arbitration, firmware or custom controller must ensure mutex access	Built-in hidden arbitration
Device addressing	Custom-designed MMIO	Plug 'n' play configuration space allows firmware to set an MMIO or IO base address at run time
Expansion and form factors	Custom bus integration on 6U boards with 3U/6U D-shell form factor	No custom expansion, but many standard form factors: compact PCI, PC/104+, PMC, and standard PC 2.x
Faster options	VME-64+	PCI-X 1.0a, 2.0, and PCI Express

The features of PCI that have made it successful and a popular integration bus are burst transfer (most IO has become high-rate and is most efficient with block transfer), built-in arbitration, and plug-and-play configuration. The two most commonly used form factors for real-time embedded systems are Compact PCI (D-shell back-plane connector) and PC/104+ (a stackable small board form factor). The PCI bus has also been very popular as a chipset interconnection on single-board embedded systems. One reason for the popularity of PCI as a chip interconnection on board is the definition of PCI bridges, which allow for bus-to-bus PCI integration. The PCI 2.x standard provides 32-bit 33-MHz bus cycles and up to 64-bit at 66 MHz for updated 2.x, yielding bandwidth of 128 million bytes per second to 512 million bytes per second. The effective bandwidth given arbitration, addressing, and device response latency overhead will be significantly reduced by bus transaction protocol overhead, but still on the order of 100 to 500 million bytes per second. For the stereo-vision example, which transfers two streams of 32-bit alpha-RGB 320 × 240 frames 30 times per second, it requires 18,432,000 bytes per second, or approximately 20% of the available PCI 2.1 effective bandwidth (assuming effective bandwidth is 100 million bytes per second). A single, full-resolution encoding (525 × 427) video stream would require 26,901,000 bytes per second, or about 27% of PCI 2.1 effective bandwidth.

The chipset used for video encoding in the stereo-vision example is the Bt878, now also updated as the Cirrus Stream Machine, and works by fetching DMA RISC engine code from host memory, so some additional PCI transfers are initiated by the chip to fetch code as well as transfer of frame data to the host memory. The stereo-vision systems implemented at the University of Colorado using PCI 2.1 have had no problem making use of PCI 2.1 for this application with a 320 × 240 30 fps alpha-RGB encoding. Many embedded applications have more than enough bandwidth available from PCI 2.x. One final important feature of PCI is that initiators can configure targets for a maximum and minimum burst length. The minimum serves as a method to reduce overhead so that targets can't transfer small blocks that would incur high overhead for each bus-arbitration and address cycle compared to fewer larger block transfers. The maximum prevents a target from overusing the bus and provides some fairness in bus arbitration for multi-target systems.

Since the first edition, many embedded camera systems now use USB 2.0 for standard definition or lower-resolution high-definition (e.g., 720p)

for raw uncompressed video. At about 1080p at 30 Hz and above, if frames are not compressed, UBS 2.0 has insufficient bandwidth to transport the frames. So, many USB 2.0 web cameras and embedded cameras have built-in MPEG-2 or MPEG-4 encoders. For machine and computer vision applications, raw uncompressed frames are ideal because the image processing algorithms generally can't work on compressed data directly. In fact, most cameras today have built-in MPEG-4 encoders, including embedded SPI (Serial Peripheral Interface), MIPI (Media Independent Peripheral Interface), and even gigabit Ethernet cameras. Developers and researchers in machine and computer vision therefore sometimes still use NTSC analog cameras or use Camera Link, direct parallel LVDS (Low-Voltage Differential Signal) bus, or specialized gigabit Vision Ethernet links. Most mobile phones use SPI or MIPI and provide lower-resolution digital video or high-definition snapshots. This too is changing rapidly as of the time of publication of this second edition, with the emergence of USB 3.0 cameras, gigabit Vision Ethernet, and continued use of NTSC and Camera Link. The author most often uses NTSC-to-USB-2.0 frame grabbers, Camera Link to USB 3.0 bridges, or standard definition with USB 2.0 to avoid the overhead of decoding digital video frames in computer and machine vision applications. Finally, to come full circle to PCI, PCI Express, which is largely compatible with PCI in general from a software and driver viewpoint, also continues to be a great transport for digital video [Siewert14].

Since the publication of the first edition of this book, the trend toward high-speed differential serial IO has continued to evolve with the emergence of USB-3.0, PCI Express 3.0, and the emergence of Thunderbolt and announcements of new PCI Express 4.0 and USB-3.1 standards. PCI Express 4.0 is capable of 16 billion transactions per second with 128b/130b encoding (130 bits to encoded 128 data bits) and an effective data rate of 15.754 billion bits per second at the link layer. The new USB-3.1 will be capable of data transfer rates close to 10 billion bits per second and uses a low-overhead 128b/132b encoding. Most of the early differential serial buses used 8b/10b encoding to control running disparity (the number of repeated 1's or 0's—logic high or low) to improve signal integrity. Overall, the use of differential serial continues to be the pervasive technology for high data rate IO for both embedded and general-purpose computing. The trend can be traced back to the original development of gigabit Ethernet, USB, and PCI Express, which have for the most part completely replaced traditional serial and parallel bus protocols in the new millennium and are

likely to continue to be the pervasive interconnection technologies. The following chapters, unchanged from the first edition, trace the emergence of differential serial links, buses, and transport protocols and the transition to high-speed differential serial links from analog serial and parallel buses. The development of encoding methods to improve signal integrity (8b/10b) and the use of link layer and higher-layer protocols to improve error handling for all methods of interconnection can be traced back to the development of Infiniband in 1999, which has become a high-performance computing interconnection standard, but has influenced many other related standards, such as USB, Ethernet, and PCI Express. An understanding of the history that led to the emergence of PCI Express, USB, and gigabit Ethernet will assist the reader with strategies to scale embedded solutions, methods to interface high data rate devices, such as cameras, and provide a long-term vision for IO.

The commercial computing market, specifically high-speed networking, graphics, and databases, pushed PCI and other parallel bus technologies at the turn of the millennium in the year 2000 to evolve into a very high-bandwidth interconnection. At that time, the AGP (Accelerated Graphics Port) standard was developed as a specific single-target expansion for PCI to accommodate high-bandwidth RAMDAC (RAM Digital-to-Analog Converters) used to drive monitors with high-fidelity graphics. Following PCI 2.x, the PCI-X 1.0a and PCI-X 2.0 standards were developed for networking and database host bus adapters and provide 64-bit 133 MHz and up to 64-bit 266/533 MHz bandwidth. The theoretical limit of the PCI bus signaling is 533 MHz. At this speed, the problem of skew between the address/data lines is significant and requires careful layout of the bus traces and advanced signaling techniques that make PCI-X 2.0 expensive and difficult to implement, especially for buses that accommodate more than one target and initiator.

The development of gigabit Ethernet in the early years of the new millennium (2000) at 1 G and 10 G rates (where G = gigabit/sec) helped drive the demand for high-rate PCI bus development for host interfaces to this new high-speed network interconnection. The PCI-X 1.0a standard, at 64-bit 133 MHz (just over 1 GB/sec), became popular for gigabit Ethernet network interfaces at this time; however, this high-end parallel bus was still not sufficient to support 10 G Ethernet. To support 10 G Ethernet, PCI Express was developed, which has a drastically different signaling and physical layer than PCI or PCI-X. PCI Express provides high-speed 2.5 G serial byte

lanes that can be ganged up. With the introduction of PCI-X, the standard introduced an important new concept—split transactions.

In PCI 2.x, when a device has high response latency, the bus delays until the target device responds. With the delay policy, having even one slow target on the bus decreases performance for all targets and initiators on the bus. Split transactions eliminate this delay by providing buffer queues for writes to the bus so that they can be posted by an initiator and drained to a slow target over the bus when it's ready. The initiator is not delayed in this case as long as the write buffer queue is not exhausted before the data is drained to the target. Likewise, on reads, split transaction allows the initiator to post a read request to the bus, which initiates the target read, and if the target is slow, allows the target to negotiate for completion in a later transaction—the bus is freed in the meantime for other transactions. Both PCI-X and PCI Express are split-transaction bus standards—this greatly improves the effective bandwidth because it does not allow the bus to be held for arbitrarily long delay periods. However, there is, of course, still arbitration and addressing overhead.

8.2.7 High-Speed Serial Interconnection

As traditional back-plane buses have become problematic as far as laying out signal traces and dealing with high-speed signaling and skew (rates above 100 MHz), several new high-speed serial interconnection standards were introduced, including:

- Universal Serial Bus
- Firewire
- PCI Express
- Gigabit Ethernet

All four serial/network interconnections can be used for real-time embedded systems and provide an attractive alternative to bus integration. A full discussion of all the high-rate serial protocols is beyond the scope of this text.

The wide adoption of PCI for real-time embedded systems in the past and key features of PCI Express make it an interconnection for scaling existing systems that became popular in the new millennium, and it continues to grow and has become pervasive. PCI Express can be routed on a board, on a back-plane, and even out of a box on a cable for short distances.

Furthermore, it's composed of serial byte lanes operating at 2.5 G for each lane, with the capability to gang up lanes in x1, x2, x4, x8, and x16 configurations. The stated design goal of PCI-E (PCI Express) is to maximize the bandwidth per pin on the interconnection. Each PCI-E byte lane is full duplex, allowing concurrent transmit and receive at 2.5 G. Given these characteristics, we can compare the PCI 2.x, PCI-X, and PCI-E standards as far as bandwidth per pin:

1. PCI-E: [(2.5 Gb/s/direction X 8b/direction) X (1B/8b)]/40 pins = 100 MB/s/pin
2. PCI 2.x: [(32b X 33 MhZ) X (1B/8b)]/84 pins = 1.58 MB/s/pin
3. PCI-X 2.0 266: [(64b X 266 MhZ) X (1B/8b)]/150 pins = 7.09 MB/s/pin

The trend for PCI-E continues to improve from this initial 2.5 billion transactions per second and high effective bandwidth per pin with rates in PCI-E 4.0 expected to top out at 16 billion transactions per second per byte lane; this latest version of PCI-E, PCI-E 4.0, was announced in 2011 and the final specification has been targeted for release in 2017. The link layer encoding and the higher-level transport layers used in PCI-E add overhead, but the effective data transfer rates are still very high and the pin efficiency is still much higher than any previous parallel bus IO. The ability to gang up serial byte lanes from x1 to x4, x8, x16, and even x32 PCI-E will also keep PCI-E relevant for some time to come.

PCI-E clearly has the advantage from the perspective of interconnection layout and cabling over earlier parallel bus technologies, such as PCI 2.x and PCI-X. The complication is that serial byte lane transmission requires significant digital signal processing on each byte lane. Like fiber channel and gigabit Ethernet, PCI-E uses an 8b/10b encoding scheme with a link layer and network layered architecture to achieve 2.5 G transfer rates. Given the demands of gigabit transport, the cost of this digital signal processing and network stack implementation has actually become more feasible than the cost of traditional bus high-speed layout. Furthermore, the ability to use high-speed serial interconnection on-chip, onboard, and off-board makes these standards more attractive than traditional bus architectures. Finally, PCI-E has been designed to be compatible with PCI 2.x and PCI-X from the firmware viewpoint, despite a radically different data transport method. The PCI-E standard supports the same plug-and-play configuration, burst transfers, interrupts, and all basic features of PCI 2.x.

The PCI-E interconnection provides byte lane interconnection with a network layered architecture, as shown in Figure 8.7.

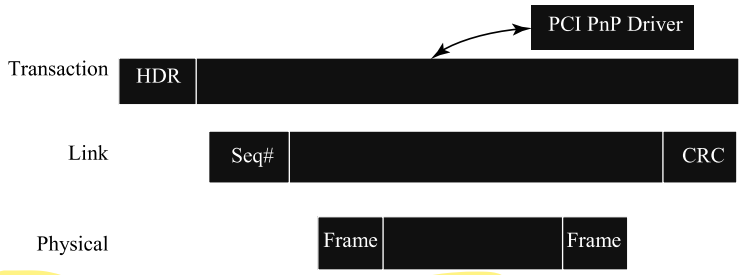

FIGURE 8.7 PCI Express Byte Lane Network Architecture

The PCI-E standard provides not only significantly more bandwidth compared to PCI 2.x and PCI-X but also some features to support real-time continuous media with isochronal channels. The isochronal channels provide bandwidth and latency performance guarantees for transport. The advent of PCI-E, USB, Firewire, and gigabit Ethernet has provided an alternative interconnection architecture for real-time embedded systems. It is likely that these new high-speed serial interconnections will be designed into many future real-time embedded systems.

8.2.8 Low-Speed Serial Interconnection

Many real-time embedded systems include not only high-rate IO for services such as video or network transport but also low-rate command/response or monitoring interfaces. For example, in our stereo-vision system, the servos are commanded through a low-rate multi-drop RS232 interface. The Microchip PIC (Programmable Integrated Circuit) has a TTL logic-level digital serial interface that can be interfaced to the higher-voltage RS232 serial interface. The servos in the stereo-vision application can tilt/pan through wide angles quickly, but have an accuracy of only several degrees, so the command rate required for tracking a quickly moving object at a distance of 10 feet or more is on the order of 1 to hundreds of milliseconds. The command protocol used on the PIC is a simple byte stream command format with opcode bytes and operand bytes. To command a given servo to a new position simply requires sending a PIC address (because the serial interconnection is multi-drop), a servo address, and opcode byte, followed by a servo position operand. A command therefore requires 4 bytes, and at a 1 millisecond rate, this is only 4,000 bytes/second. Clearly PCI,

USB, Firewire, or any of the previously presented high-rate interconnection architectures are not warranted for this type of interface.

The RS232, common serial, point-to-point data transmission has been used in real-time embedded systems since the advent of the industry and remains a common low-rate and debug interface. The RS232 link normally tops out around 115,200 bits/second (about 12 KB/sec) and is not capable of long-distance transmission due to line noise at the 12-volt signaling levels it uses. Other options have evolved that provide similar low- to medium-rate transmission with longer distance, multi-drop, and higher bit rates. These options are widely used in real-time embedded systems:

- RS422—a differential +/- 5v serial link capable of 1 megabit/sec and distances up to 1 km
- Multi-drop RS232, RS422—adding a protocol to address targets on a common link with capability to forward
- I2C—a medium-speed digital interconnection typically used onboard to interconnect chips such as EEPROM to a processor
- SPI (Serial Peripheral Interface)—a digital serial protocol capable of medium rates

A full discussion of all the low- to medium-rate serial protocols is beyond the scope of this text and continues to evolve rapidly over time, but the introduction provided here is a good starting point.

8.2.9 Interconnection Systems

Having discussed the components that can be used to interconnect devices with processors in a real-time embedded system, let's briefly discuss how these components might be arranged in an interconnection architecture. Real-time embedded system architectures and design will be discussed more fully in Chapter 12, but an overview will help summarize the possibilities. Two architectures are most common. The first is the hierarchical network interconnecting a main processor complex with a number of microcontrollers. This architecture has been most popular in robotics and also for aerospace applications, where many sensors and actuators are distributed in a large system, yet processing and services provide system-level functions. For example, a robotic arm that has five degrees of freedom (base, shoulder, elbow, wrist, and claw) with torque control so that

it can handle massive objects might include a microcontroller to provide control for each joint motor. The five microcontrollers, one at each joint, can interface locally to the actuation motor with a DAC and provide closed-loop feedback control based upon local position and stress/strain sensors to provide smooth rotation, even when the arm is handling objects with significant mass. Providing the local control reduces the amount of cabling back to the main processor. The DAC and the sensor interface cabling are routed to the microcontroller, which is physically integrated local to each joint (or control point). The microcontroller has more than sufficient capability to provide the torque control and closed-loop monitoring. Now, the five microcontrollers can be interfaced via a low- to medium-rate serial interconnection back to the main processor complex for commands and to provide status—the main processor complex runs complex services, such as path planning, possibly camera-based object recognition, a user interface, and system health and status monitoring. In fact, the robotics community likens this hierarchical approach to the human body, which includes local reflex control as well as centralized processing in the brain—for example, Rodney Brooks's subsumption architecture [Brooks86].

The alternative to the hierarchical interconnection is a centralized processor complex integrated on a high-rate interconnection, such as PCI, with many IO device interfaces also integrated on PCI. This architecture has an advantage in that all processing can be done in a single processor complex; the distributed processing of the hierarchical architecture requires different development, debug, and test methods compared to the processor complex. Often the processor complex is a microprocessor with an RTOS, and the microcontrollers are simple Main+ISR applications. The downside to the centralized processing is that most often all IO cabling must come from the common central processing enclosure and be routed to sensors and actuators distributed throughout the system. Deciding which type of architecture makes most sense is often driven by the system requirements for actuators and sensors—the number, how distributed they are, how much latency in sensor/actuator activation is allowable, and, of course, cost and complexity.

8.2.10 Memory Subsystems

Real-time embedded systems require nonvolatile data storage to boot the system and to start services. After a power-on reset, the processors in the processor complex each vector to a hardware-defined starting address to execute code. This starting address, typically a high address, such

as 0xFFE0_0000, is designed to map a nonvolatile storage device, such as EEPROM or Flash memory, so that boot code can be stored permanently at this address and executed following a reset to initialize the system. The boot code initializes all basic interfaces and normally loads a basic RTOS so that application services can be loaded and run. The code (often called text segment), data (initialized, uninitialized, and read-only), heap, and stack segments must be created in a working memory by firmware. Figure 8.8 shows a typical memory map for an embedded system.

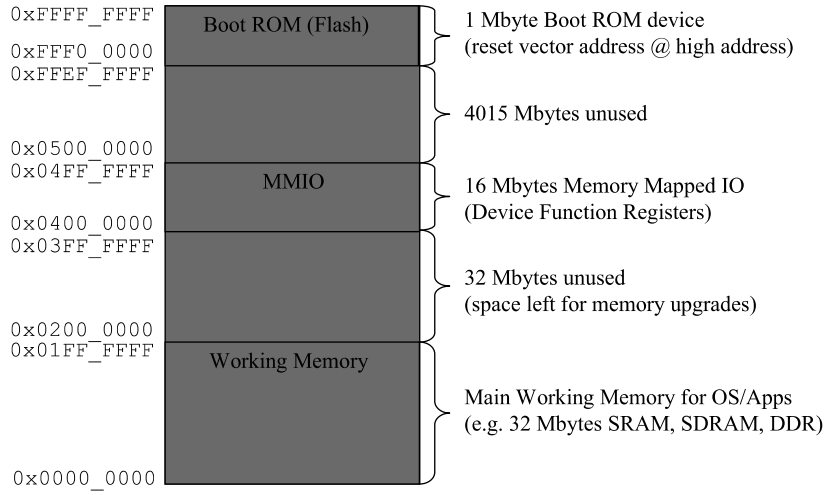

FIGURE 8.8 Common Memory Map for an Embedded System

The memory map is really a logical view of memory from the viewpoint of address space through which firmware and software can access devices. From a hardware viewpoint, memory is better described as a hierarchy of storage devices, including:

- Registers (CPU and memory mapped for device control)
- Cache
- Working Memory
- Extended Memory

Figure 8.9 shows a typical memory hierarchy for an embedded system.

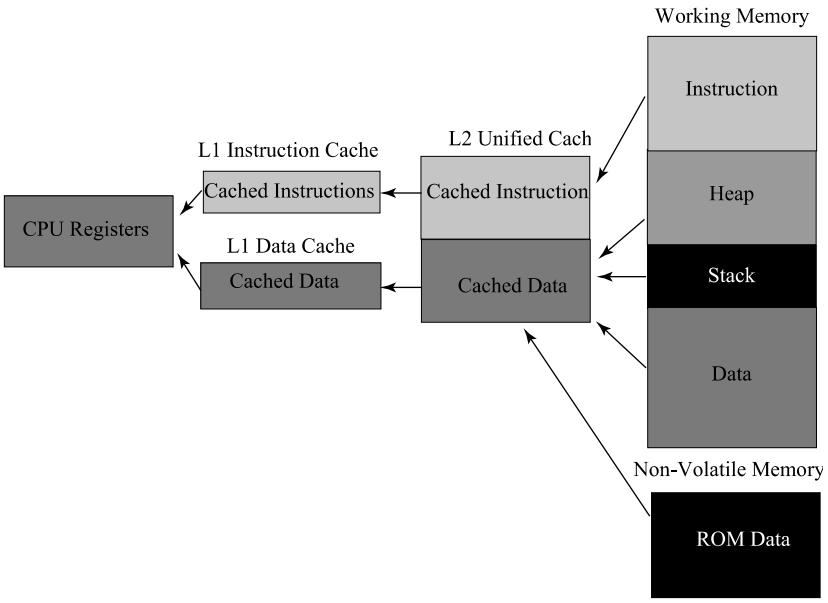

FIGURE 8.9 Common Physical Memory Hierarchy for an Embedded System

8.3 Firmware Components

Some components can be realized only in hardware, but many can be implemented with software or firmware (noting that software interfacing directly to hardware is typically called firmware). Furthermore, if the real-time embedded system has any software-based services or even just management, firmware is needed to interface hardware resources to software applications.

8.3.1 Boot Code

The universal definition of *firmware* is code or software that runs out of a nonvolatile device to make hardware resources available for the rest of the application software. Firmware providing this function is normally referred to in general as *board support package* (BSP) firmware because traditionally this firmware has initialized and made available all onboard resources for a processor complex to software applications. Before the resources have been fully initialized, the firmware boots the board by executing code out of a nonvolatile device so that one or more basic interfaces are made operable and the system can now download additional application software. For example, the BSP boot firmware might initialize an Ethernet interface and provide TFTP download of application code for execution.

8.3.2 Device Drivers

Device interface drivers are most often considered firmware because they directly interface to hardware resources and make those resources available to higher-level software applications. The architecture of a device driver interface is depicted in Figure 8.10 and includes both an HW device interface and an SW application interface.

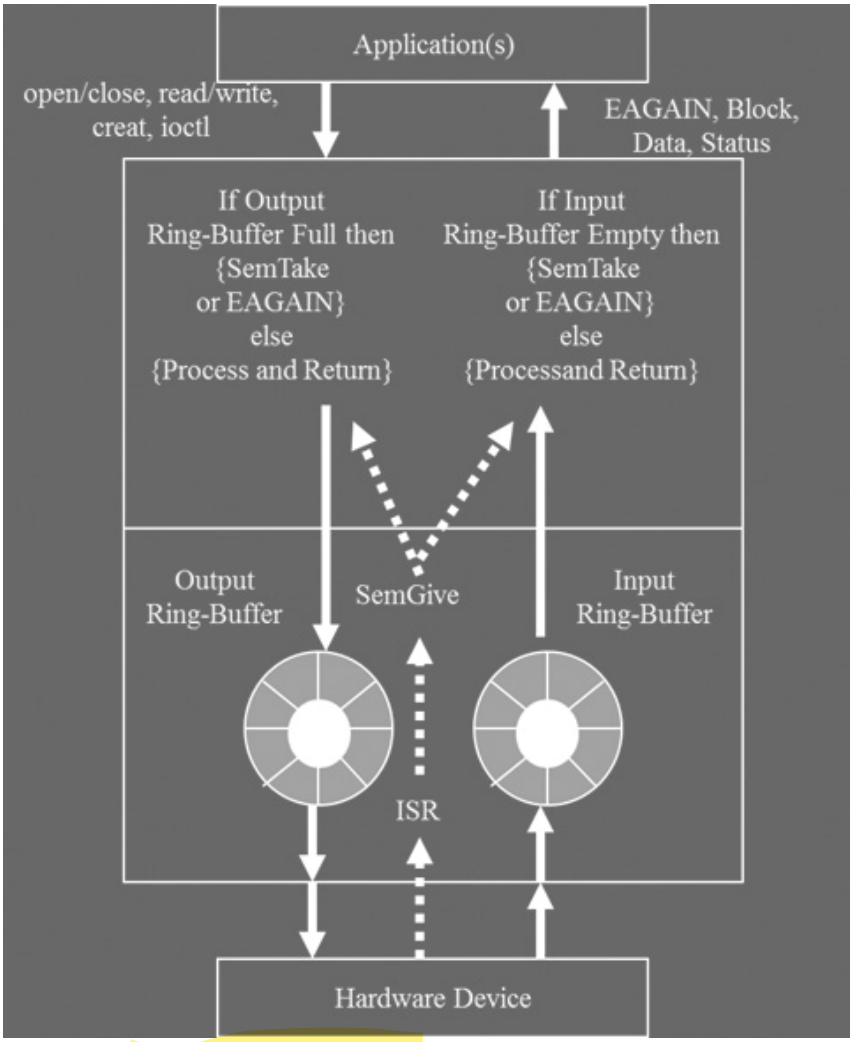

FIGURE 8.10 Device Driver Firmware Interface

8.3.3 Operating System Services

Not all real-time embedded systems require an operating system, as discussed in Chapter 3. Most RTOS implementations do, however, provide a layer of software that acts a single interface for all applications to gain access to system resources. Furthermore, most real-time systems incorporate an RTOS, which provides a framework for resource management and for scheduling processor resources with an RM policy. The RTOS also provides commonly needed services and libraries used by application services. The most fundamental services and mechanisms provided include the following:

- Priority preemptive scheduler for threads
- Thread control block management
- Inter-thread synchronization and communication (e.g., semaphores and message queues)
- Basic IO for system debug and bring-up (e.g., serial, Ethernet, LED)
- Interrupt service routine installation on interrupt vectors
- Transition from boot to operational state
- Timers for delays and blocked thread timeouts
- Drivers for basic hardware devices (serial, Ethernet, timers, nonvolatile memory)

Extended services beyond these may be provided to assist development and debug of a system:

- Cross debug agent
- Interactive shell to view control blocks and system context
- Ability to dynamically load and execute code object files
- Interface to resource analysis tools (e.g., WindView, now known as System Viewer)

8.4 RTOS System Software

An RTOS does not need to provide the same wealth of system services as a full multiuser operating system, such as Linux. The understanding is