CHAPTER

Computing Platforms

4

CHAPTER POINTS

- CPU buses, I/O devices, and interfacing.
- The CPU system as a framework for understanding design methodology.
- System-level performance and power consumption.
- Security features of platforms.
- Development environments and debugging.
- Design examples: Alarm clock, jet engine controller.

4.1 Introduction

In this chapter, we concentrate on **computing platforms** created using microprocessors, I/O devices, and memory components. The microprocessor is an important element of the embedded computing system, but it cannot do its job without memories and I/O devices. We need to understand how to interconnect microprocessors and devices using the CPU bus. The application also relies on software that is closely tied to the platform hardware. Luckily, there are many similarities between the platforms required for different applications, so we can extract some generally useful principles by examining a few basic concepts.

The next section surveys the landscape of computing platforms, including both hardware and software. Section 4.3 discusses CPU buses. Section 4.4 describes memory components and systems, while Section 4.5 introduces some useful I/O devices. Section 4.6 considers how to design with computing platforms. Section 4.7 discusses how to design a system using an embedded platform. Section 4.8 develops methods to analyze performance at the platform level, and Section 4.9 considers the power management of the platform. Section 4.10 discusses security support in the platform. We close with two design examples: an alarm clock design example in Section 4.11 and a wearable pedometer in Section 4.12.

4.2 Basic computing platforms

Although some embedded systems require sophisticated platforms, many can be built around the variations of a generic computer system, ranging from 4-bit microprocessors through complex systems-on-chips. The platform provides an environment in which we can develop our embedded application. It encompasses both hardware and software components—one without the other is generally not effective.

4.2.1 Platform hardware components

We are familiar with the CPU and memory as an idealized computer system. A practical computer needs additional components. As shown in Fig. 4.1, a typical computing platform includes several major hardware components:

- The CPU provides basic computational facilities.
- RAM is used for program and data storage.
- Flash (also known as ROM) holds the code and data that do not change.
- A direct memory access (DMA) controller provides DMA capabilities.
- Timers are used by the operating system for a variety of purposes.
- A high-speed bus connected to the CPU bus through a bridge allows fast devices to communicate efficiently with the rest of the system.
- A low-speed bus provides an inexpensive way to connect simpler devices and may be necessary for backward compatibility.

Buses

The bus provides a common connection between all the components in the computer: the CPU, memories, and I/O devices. We discuss buses in more detail in Section 4.3; the bus transmits addresses, data, and control information so that one device on the bus can read or write to/from another device.

While simple systems will have only one bus, more complex platforms may have several buses or interconnection networks. Buses are often classified by their overall performance: low-speed, high-speed, and so on. Multiple buses serve two purposes. First, devices on different buses will interact far less than those on the same bus. Dividing devices among buses can help reduce the overall load and increase the utilization of the buses. Second, low-speed buses usually provide simpler and cheaper

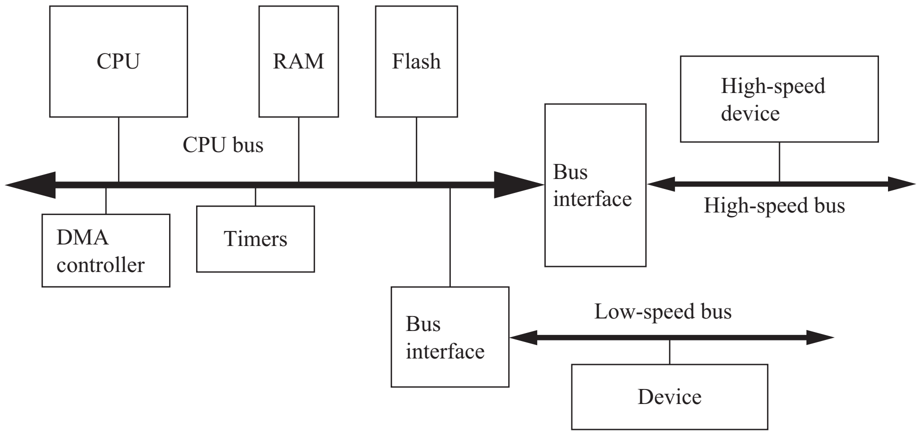


FIGURE 4.1

Hardware architecture of a typical computing platform.

interfaces than high-speed buses. A low-speed device may not benefit from the effort required to connect it to a high-speed bus.

Wide ranges of buses are used in computer systems. The USB, for example, is a bus that uses a small bundle of serial connections. For a serial bus, the USB provides high performance. However, complex buses, such as Peripheral Component Interconnect Express (PCIE), may use many parallel connections and other techniques to provide higher absolute performance.

Access patterns

Data transfers may occur between many pairs of components: CPU to/from memory, CPU to/from I/O device, memory to memory, or I/O to I/O device. Because the bus connects all these components (possibly through a bridge), it can mediate all types of transfers. However, basic data transfer requires executing instructions on the CPU. We can use a direct memory access (DMA) unit to offload some of the basic transfers. We discuss DMA in more detail in Section 4.3.

Single-chip platforms

We can also put all the components for a basic computing platform on a single chip. A single-chip platform makes the development of certain types of embedded systems much easier, providing rich software development of a PC with the low cost of a single-chip hardware platform. The ability to integrate a CPU and devices on a single chip has allowed manufacturers to provide single-chip systems that do not conform to board-level standards.

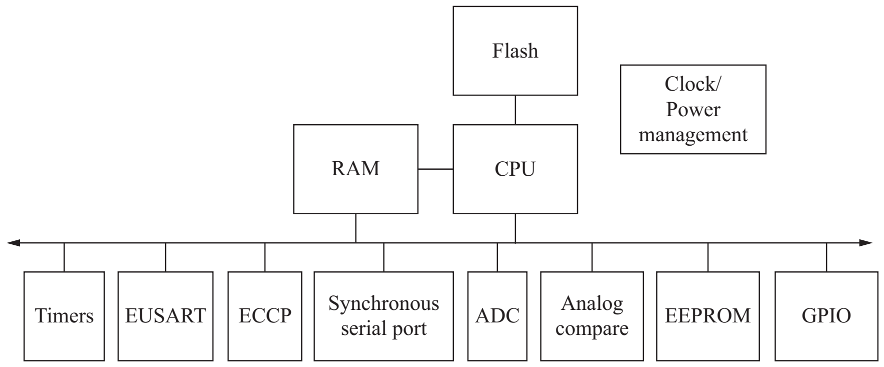
Microcontrollers

A microcontroller is a single chip that includes a CPU, memory, and I/O devices. The term was originally used for platforms based on small 4- and 8-bit processors, but it can also refer to single-chip systems using large processors.

The next two examples look at two different single-chip systems. Application Example 4.1 looks at PIC16F882, whereas Application Example 4.2 describes Cypress PSoC 6.

Application Example 4.1: System Organization of PIC16F882

Here is a block diagram of the PIC16F882 (as well as 883 and 886) microcontroller [Mic09]:



PIC is a Harvard architecture; the flash memory used for instructions is accessible only to the CPU. The flash memory can be programmed using separate mechanisms. The microcontroller includes several devices: timers; a universal synchronous/asynchronous receiver/transmitter (EUSART); capture-and-compare modules; a primary synchronous serial port; an analog-to-digital converter (ADC); analog comparators and references; an electrically erasable programmable ROM; and general-purpose I/O (GPIO).

Application Example 4.2: System Organization of Cypress PSoC 6

Cypress PSoC 6 CY8C62x8 and CY8C62xA [Cyp20] provide dual CPUs: a 150-MHz Arm Cortex-M4F with a single-cycle multiplier and floating-point and memory protection units; and a 100-MHz Cortex-M0+ with a single-cycle multiplier and memory protection unit. The memory system includes flash, static RAM (SRAM), and one-time programmable memory. The power management system provides six power modes. I/O includes several types of serial communication, an audio system, timing and pulse-width modulation, and programmable analog functions.

4.2.2 Platform software components

Hardware and software are inseparable; each needs the other to perform its function. Much of the software in an embedded system comes from outside sources. Hardware vendors generally provide a basic set of software platform components to encourage the use of their hardware. These components range across many layers of abstraction.

Layer diagrams are often used to describe the relationships between different software components in a system. Fig. 4.2 shows a layer diagram of an embedded system. The hardware abstraction layer (HAL) provides a basic level of abstraction from hardware. Device drivers often use the HAL to simplify their structures. Similarly, the

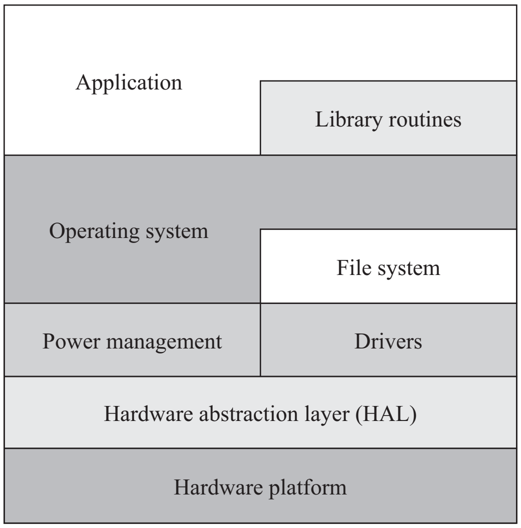


FIGURE 4.2

Software layer diagram for an embedded system.

power management module must have low-level access to hardware. The operating system and file system provide the basic abstractions required to build complex applications. Because many embedded systems are algorithm-intensive, we often make use of library routines to perform complex kernel functions. These routines may be developed internally and reused, or in many cases, they come from the manufacturer and are heavily optimized for the hardware platform. The application makes use of all these layers, either directly or indirectly.

4.3 The CPU bus

The **bus** is the mechanism by which the CPU communicates with memory and devices. A bus is, at a minimum, a collection of wires, but it also defines a protocol by which the CPU, memory, and devices communicate. One of the major roles of the bus is to provide an interface to memory. Of course, I/O devices also connect to the bus. Based on an understanding of the bus, we can study the characteristics of the memory components in this section, focusing on DMA. We also look at how buses are used in computer systems.

The term bundle is used to describe a collection of signals without an associated protocol. We use the terms “controller” and “responder” to describe roles in protocols; the Open Compute Project documents guidelines for inclusive and open terminology [Car20].

4.3.1 Bus organization and protocol

A bus is a common connection between components in a system. As shown in Fig. 4.3, the CPU, memory, and I/O devices are all connected to the bus. The signals comprising the bus provide the necessary communication: the data, addresses, a clock, and control signals.

Bus controller

In a typical bus system, the CPU serves as the **bus controller** and initiates all transfers. If any device could request a transfer, then other devices might be starved

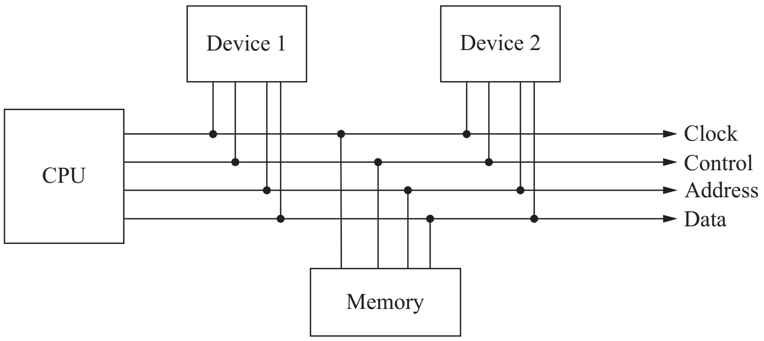


FIGURE 4.3

Organization of a bus.

of bus bandwidth. Via a bus controller, the CPU reads and writes data and instructions from memory. It also initiates all reads or writes on I/O devices. DMA allows other devices to temporarily become the bus controller and transfer data without the CPU's involvement.

Four-cycle handshake

The basic building block of most bus protocols is the **four-cycle handshake**, as illustrated in Fig. 4.4. The handshake ensures that, when two devices want to communicate, one is ready to transmit, and the other is ready to receive. The handshake uses a pair of wires dedicated to the handshake: **enq** (meaning enquiry) and **ack** (meaning acknowledge). Extra wires are used for the data transmitted during the handshake. Each step in the handshake is identified by a transition on *enq* or *ack*:

1. *Device 1* raises its *enq* output to signal an enquiry, which tells *device 2* that it should get ready to listen for data.
2. When *device 2* is ready to receive, it raises its *ack* output to signal an acknowledgment.
3. Once the data transfer is complete, *device 1* lowers its *enq* output.
4. After seeing that *enq* has been released, *device 2* lowers its *ack* output.

At the end of the handshake, both handshaking signals are low, just as they were at the start of the handshake. The system thus returns to its original state in readiness for another handshake-enabled data transfer.

Bus signals

Microprocessor buses build on the handshake for communication between the CPU and other system components. The term *bus* is used in two ways. The most basic use is as a set of related wires, such as address wires. However, the term may also mean a protocol for communicating between components. To avoid confusion, we will use the term **bundle** to refer to a set of related signals. The fundamental bus operations are reading and writing. The major components on a typical bus include:

- *Clock* provides synchronization to the bus components.
- *R/W'* is true when the bus is reading and false when the bus is writing.
- *Address* is an *a*-bit bundle of signals that transmits the address for access.
- *Data* is an *n*-bit bundle of signals that can carry data to or from the CPU.
- *Data-ready* signals when the values on the data bundle are valid.

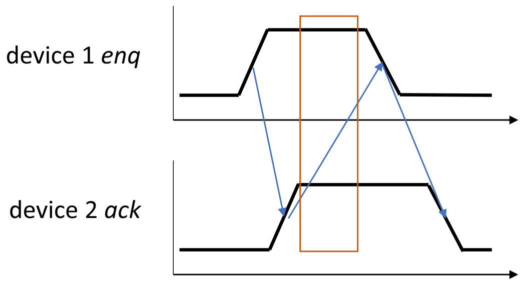


FIGURE 4.4

The four-cycle handshake.

All transfers on this basic bus are controlled by the CPU. The CPU can read or write a device or memory but devices and memory cannot initiate transfers. This is reflected by the fact that *R/W'* and address are unidirectional signals because only the CPU can determine the address and direction of a transfer.

Bus transactions

We refer to a read or a write on a bus as a **transaction**. The operation of a bus transaction is governed by the bus protocol. Most modern buses use a clock to synchronize the operations of devices on the bus. The bus clock frequency does not have to match that of the CPU, and in many cases, the bus runs significantly slower than the CPU does.

Bus reads and writes

Fig. 4.5 shows a sequence diagram for a read followed by a write. The CPU first reads a location from the memory, and then, writes it to dev1. The bus mediates each transfer. The bus operates under a protocol that determines when components on the bus can use certain signals and what those signals mean. The details of bus protocols are not important here, but it is important to keep in mind that bus operations take time, since the clock frequency of the bus is often much lower than that of the CPU. We will see how to analyze platform-level performance in Section 4.7.

Sequence diagrams don't give us enough detail to fully understand the hardware. To provide the required details, the behavior of a bus is most often specified as a

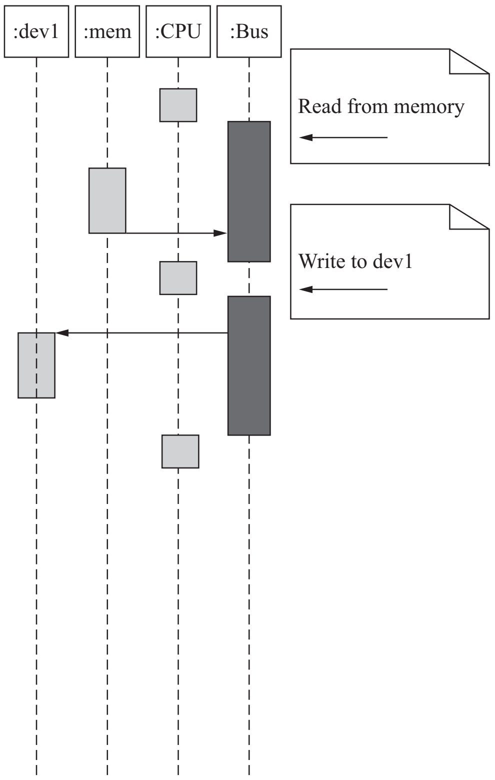


FIGURE 4.5

A typical sequence diagram for bus operations.

timing diagram. A timing diagram shows how the signals on a bus vary over time. Because signals such as address and data can take on many values, some standard notation is used to describe signals, as shown in Fig. 4.6. *A's* value is known at all times; thus, it is shown as a standard waveform that changes between zero and one. *B* and *C* alternate between **changing** and **stable** states. A stable signal has, as the name implies, a stable value that can be measured by an oscilloscope, but the exact value of that signal does not matter for the purposes of the timing diagram. For example, an address bus may be shown to be stable when the address is present, but the bus's timing requirements are independent of the exact address on the bus. A signal can go between a known 0/1 state and a stable/changing state. A changing signal does not have a stable value. Changing signals should not be used for computation. Timing diagrams sometimes show **timing constraints** to describe the precise relationships in time between events on the signals. We draw timing constraints in two different ways depending on whether we are concerned with the amount of time between events or only the order of events. The timing constraint from *A* to *B*, for example, shows that *A* must go high before *B* becomes stable. The constraint from *A* to *B* also has a time value of 10 ns, indicating that *A* goes high for at least 10 ns before *B* becomes stable.

Fig. 4.7 shows a timing diagram for the example bus. The diagram shows a read followed by a write. Timing constraints are shown only for the read operation, but similar constraints apply to the write operation. The bus is normally in read mode because that does not change the state of any of the devices or memories. The CPU can then ignore the bus data lines until it wants to use the results of a read. Notice also that the direction of data transfer on bidirectional lines is not specified in the timing diagram. During a read, the external device or memory sends a value on the data lines, whereas during a write, the CPU controls the data lines.

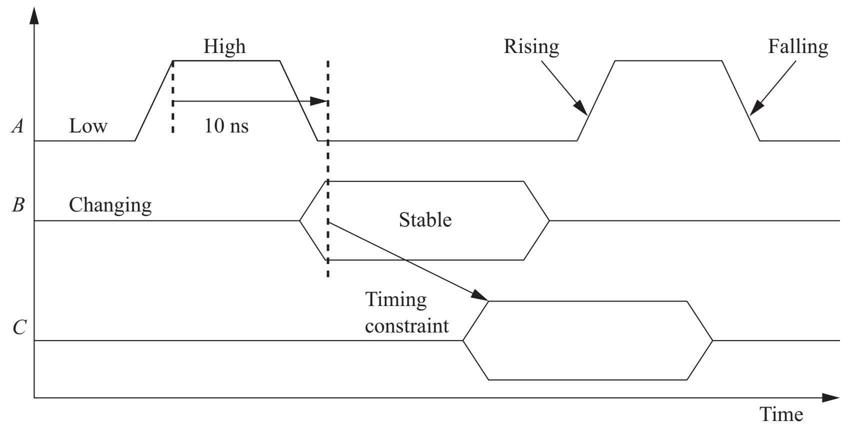


FIGURE 4.6

Timing diagram notation.

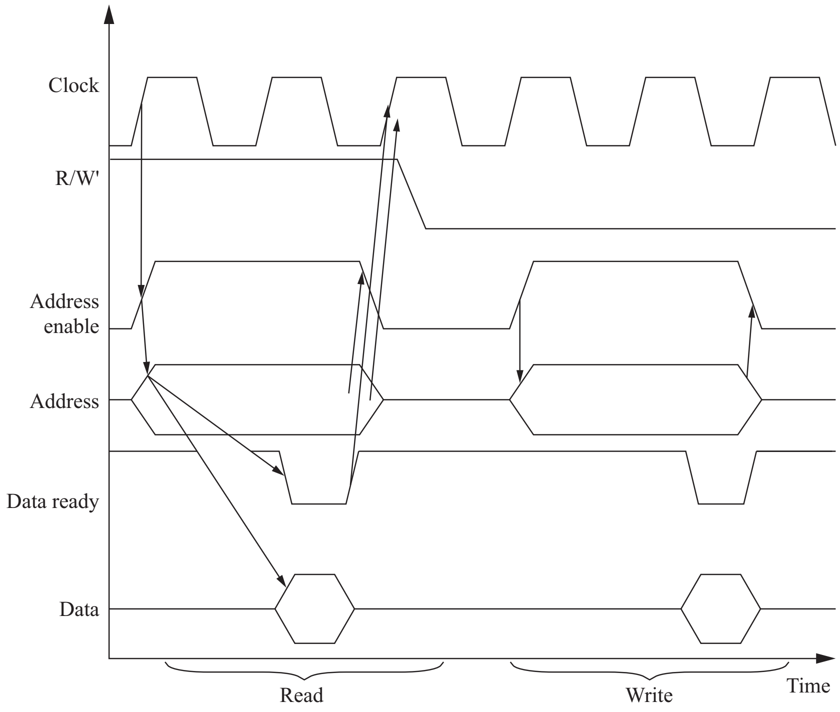


FIGURE 4.7

Timing diagram for read and write on the example bus.

With practice, we can see the sequence of operations for a read on the timing diagram:

- A read or write is initiated by setting the address enable high after the clock starts to rise. We set *R/W'* = 1 to indicate a read, and the address lines are set to the desired address.
- One clock cycle later, the memory or device is expected to assert the data value at that address on the data lines. Simultaneously, the external device specifies that the data are valid by pulling down the *data-ready* line. This line is **active low**, meaning that a logically true value is indicated by a low voltage to provide increased immunity from electrical noise.
- The CPU is free to remove the address at the end of the clock cycle and must do so before the beginning of the next cycle. The external device has a similar requirement for removing the data value from the data lines.

The write operation has a similar timing structure. The read/write sequence illustrates that timing constraints are required for the transition of the *R/W'* signal between read and write states. The signal must, of course, remain stable within a read or write.

As a result, there is a restricted time window in which the CPU can change between the read and write modes.

The handshake that tells the CPU and devices when data are to be transferred is formed by *data-ready* for the acknowledge side but is implicit for the enquiry side. Because the bus is normally in read mode, *enq* does not need to be asserted, but the acknowledgment must be provided by *data-ready.*

The *data-ready* signal allows the bus to be connected to devices that are slower than the bus. As shown in Fig. 4.8, the external device does not need to immediately assert *data-ready.* The cycles between the minimum time at which data can be asserted and when they are actually asserted are known as **wait states.** Wait states are commonly used to connect slow, inexpensive memories to buses.

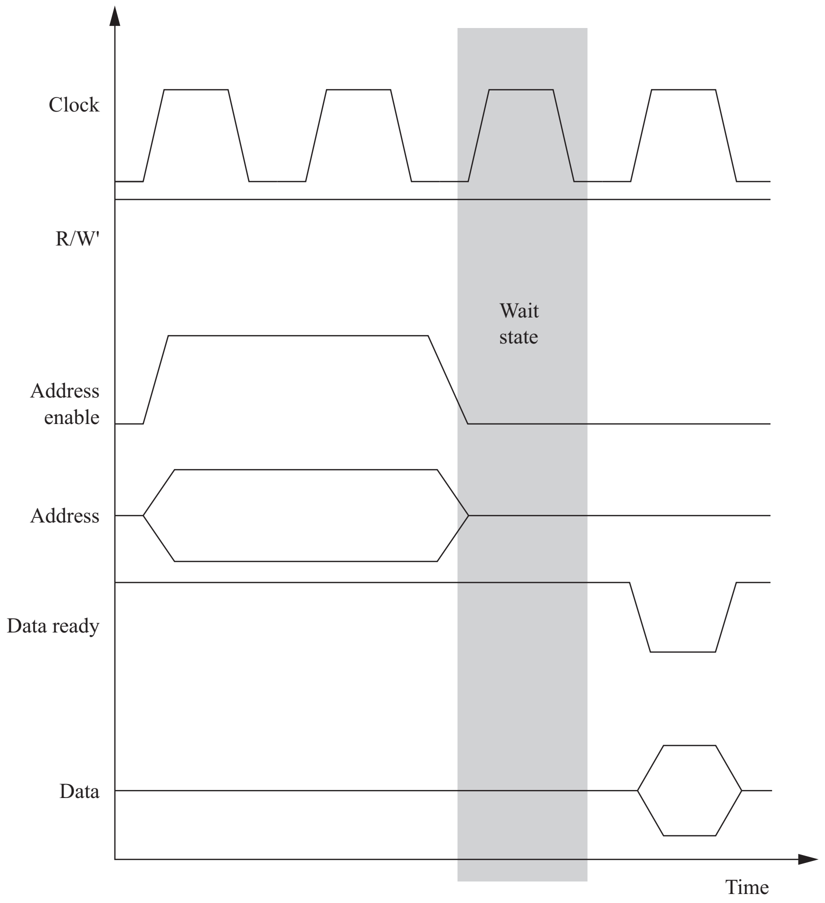


FIGURE 4.8

A wait state on a read operation.

We can also use bus handshaking signals to perform **burst transfers**, as illustrated in Fig. 4.9. In this burst read transaction, the CPU sends one address but receives a sequence of data values. We add an extra line to the bus, called *burst'* here, which signals when a transaction is actually a burst. Releasing the *burst* signal tells the device that enough data have been transmitted. To stop receiving data after the end of *data 4*, the CPU releases the *burst'* signal at the end of *data 3* because the device requires some time to recognize the end of the burst. These values come from successive memory locations starting at the given address.

Some buses provide **disconnected transfers.** In these buses, the requests and responses are separate. The first operation requests a transfer. The bus can then be used for other operations. The transfer is completed later, when the data are ready.

The state machine view of the bus transaction is also helpful and a useful complement to the timing diagram. Fig. 4.10 shows the CPU and device state machines for the read operation. As with a timing diagram, we do not show all possible values of

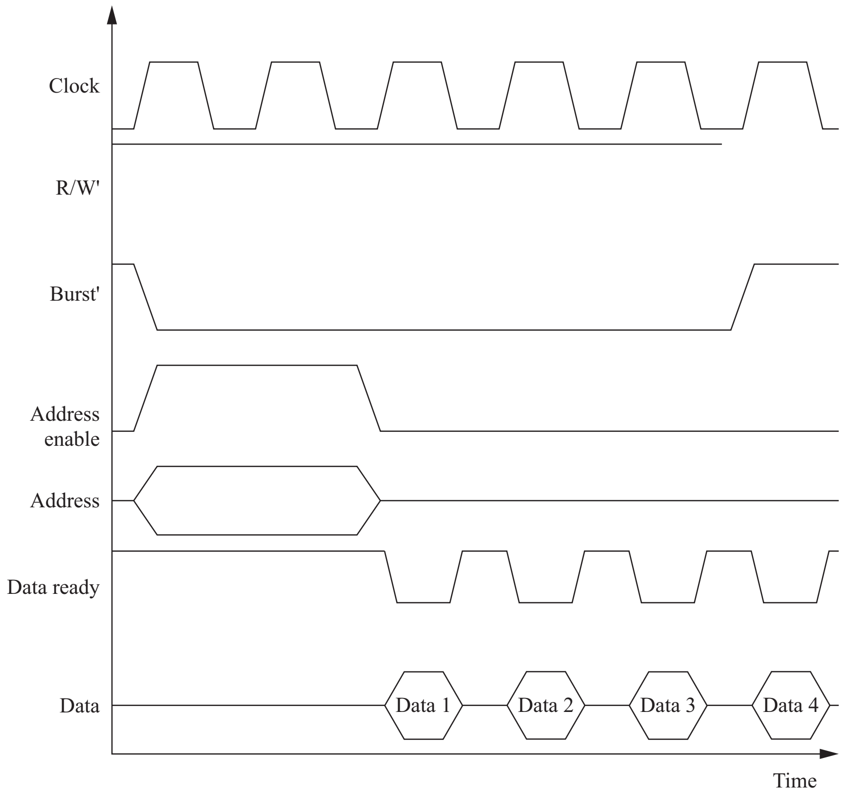


FIGURE 4.9

A burst read transaction.

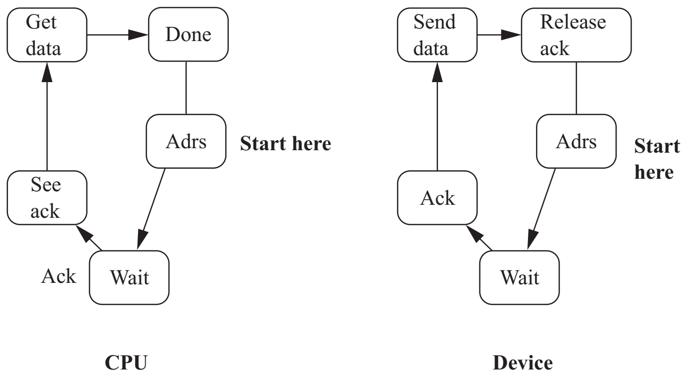


FIGURE 4.10

State diagrams for the bus read transaction.

address and data lines; instead, we concentrate on the transitions of control signals. When the CPU decides to perform a read transaction, it moves to a new state, sending bus signals that cause the device to behave appropriately. The device's state transition graph captures its side of the protocol.

Some buses have data bundles that are smaller than the natural word size of the CPU. Using fewer data lines reduces the cost of the chip. Such buses are easiest to design when the CPU is natively addressable. A more complicated protocol hides the smaller data sizes from the instruction execution unit in the CPU. Byte addresses are sequentially sent over the bus, receiving one byte at a time; the bytes are assembled inside the CPU's bus logic before being presented to the CPU proper.

4.3.2 Direct memory access

Standard bus transactions require the CPU to be in the middle of every read and write transaction. However, there are certain types of data transfers in which the CPU does not need to be involved. For example, a high-speed I/O device may want to transfer a block of data into the memory. Although it is possible to write a program that alternately reads the device and writes to memory, it would be faster to eliminate the CPU's involvement and allow the device and memory communicate directly. This capability requires that some unit other than the CPU be able to control operations on the bus.

Direct memory access (DMA) is a bus operation that allows reads and writes not controlled by the CPU. A DMA transfer is controlled by a **DMA controller**, which requests control of the bus from the CPU. After gaining control, the DMA controller performs read and write operations directly between the devices and the memory.

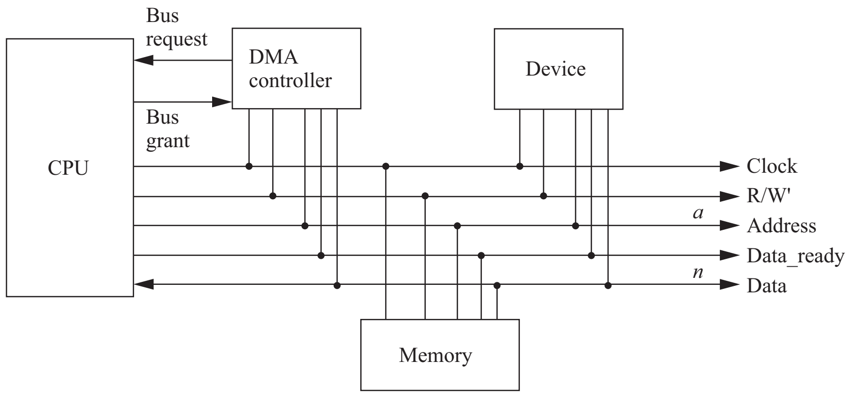


FIGURE 4.11

A bus with a direct memory access controller.

Fig. 4.11 shows the configuration of a bus with a DMA controller. The DMA requires the CPU to provide two additional bus signals:

- The **bus request** is an input to the CPU through which the DMA controllers ask for ownership of the bus.
- The bus **grant** signals that the bus has been granted to the DMA controller.

The DMA controller can act as a bus controller. It uses the bus request and bus grant signal to gain control of the bus using a classic four-cycle handshake. A bus request is asserted by the DMA controller when it wants to control the bus, and the bus grant is asserted by the CPU when the bus is ready. The CPU will finish all pending bus transactions before granting control of the bus to the DMA controller. When it does grant control, it stops driving the other bus signals: R/W', address, and so on. Upon becoming a bus controller, the DMA controller has control of all bus signals (except, of course, for bus requests and bus grants).

Once the DMA controller is a bus controller, it can perform reads and writes using the same bus protocol as with any CPU-driven bus transaction. Memory and devices do not know whether a read or write is performed by the CPU or by a DMA controller. After the transaction is finished, the DMA controller returns the bus to the CPU by de-asserting the bus request, causing the CPU to de-assert the bus grant.

The CPU controls the DMA operation through registers in the DMA controller. A typical DMA controller includes the following three registers:

- A starting address register specifies where the transfer is to begin.
- A length register specifies the number of words to be transferred.
- A status register allows the DMA controller to be operated by the CPU.

The CPU initiates a DMA transfer by setting the starting address and length registers appropriately, and then, writing the status register to set its start transfer bit.

After the DMA operation is complete, the DMA controller interrupts the CPU and tells it that the transfer is done.

Concurrency during DMA

Because the CPU cannot use the bus during a DMA transfer, what does it do? As illustrated in Fig. 4.12, if the CPU has enough instructions and data in the cache and registers, it may be able to continue doing useful work for quite some time and may not notice the DMA transfer. However, when the CPU needs the bus, it stalls until the DMA controller returns bus controllership to the CPU.

To prevent the CPU from idling too long, most DMA controllers implement modes that occupy the bus for only a few cycles at a time. For example, the transfer may be made 4, 8, or 16 words at a time. As illustrated in Fig. 4.13, after each block, the DMA controller returns control of the bus to the CPU and goes to sleep for a preset period, after which it requests the bus again for the next block transfer.

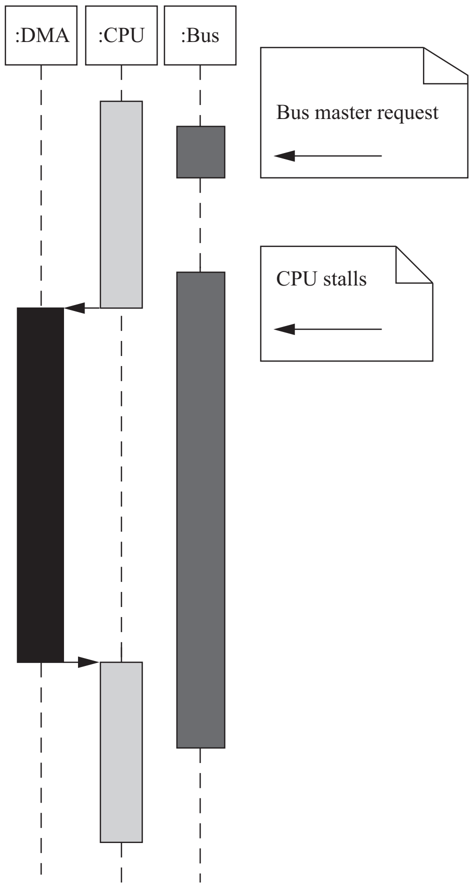


FIGURE 4.12

UML sequence of system activity around a direct memory access transfer.

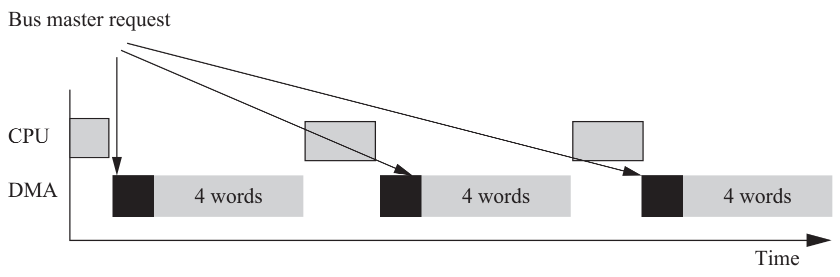


FIGURE 4.13

Cyclic scheduling of a direct memory access request.

4.3.3 System bus configurations

A microprocessor system often has more than one bus. As shown in Fig. 4.14, high-speed devices may be connected to a high-performance bus, whereas lower-speed devices may be connected to a different bus. A small block of logic, known as a **bridge**, allows the buses to connect to each other. There are three reasons to do this:

- Higher-speed buses may provide wider data connections.
- A high-speed bus usually requires more expensive circuits and connectors. The cost of low-speed devices can be reduced by using a lower-speed, lower-cost bus.
- The bridge may allow buses to operate independently, thereby providing some parallelism in I/O operations.

Bus bridges

Let's consider the operation of a bus bridge between what we call a fast bus and a slow bus, as illustrated in Fig. 4.15. The bridge is a responder on the fast bus and the controller of the slow bus. The bridge takes commands from the fast bus, upon which it is a responder, and issues those commands on the slow bus. It also returns the results

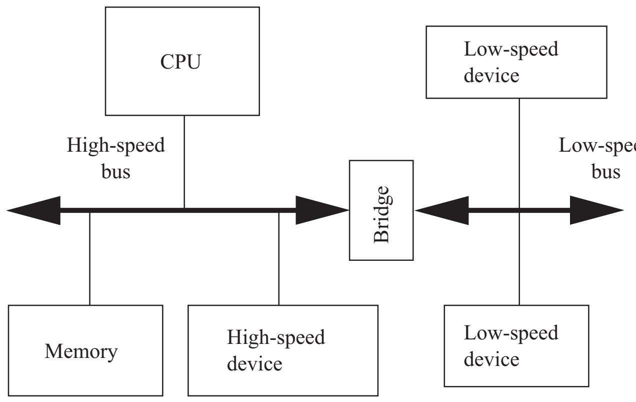


FIGURE 4.14

A multiple bus system.

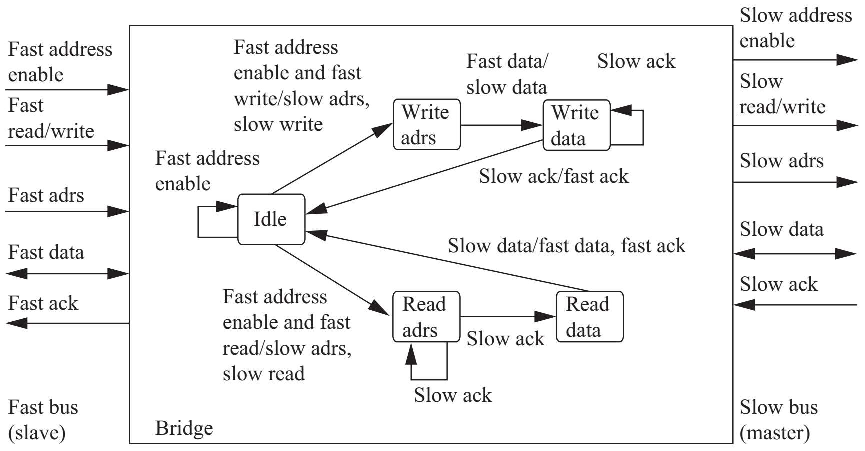


FIGURE 4.15

Unified modeling language state diagram of bus bridge operation.

from the slow bus to the fast bus. For example, it returns the results of a read on the slow bus to the fast bus.

The upper sequence of states handles a write from the fast bus to the slow bus. These states must read the data from the fast bus and set up the handshake for the slow bus. Operations on the fast and slow sides of the bus bridge should overlap as much as possible to reduce the latency of bus-to-bus transfers. Similarly, the bottom sequence of states reads from the slow bus and writes the data to the fast bus.

The bridge also serves as a protocol translator between the two bridges. If the bridges are very close in protocol operation and speed, a simple-state machine may be enough. If there are larger differences in the protocol and timing between the two buses, the bridge may need to use registers to hold some data values temporarily.

Arm bus

Because the Arm CPU is manufactured by many different vendors, the bus provided off-chip can vary from chip to chip. ARM has created a separate bus specification for single-chip systems. The Advanced Microcontroller Bus Architecture (AMBA) bus [ARM99A] supports CPUs, memories, and peripherals integrated into a system-on-silicon. As shown in Fig. 4.16, the AMBA specification includes two buses. The AMBA High-Performance Bus (AHB) is optimized for high-speed transfers and is directly connected to the CPU. It supports several high-performance features: pipelining, burst transfers, split transactions, and multiple bus controllers.

A bridge can be used to connect the AHB to an AMBA Peripheral Bus (APB). This bus is designed to be simple and easy to implement; it also consumes relatively little power. The APB assumes that all peripherals act as responders, simplifying the logic required for both the peripherals and the bus controller. It also does not perform pipelined operations, which simplifies bus logic.

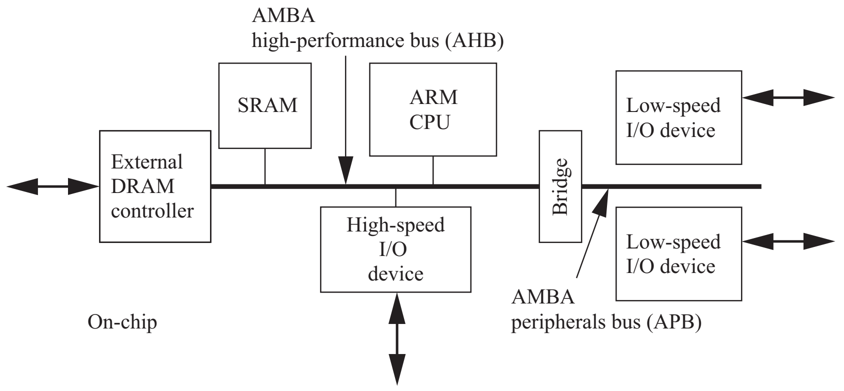


FIGURE 4.16

Elements of the Arm AMBA bus system.

4.4 Memory devices and systems

RAMs can be both read and written. They are called random access because, unlike magnetic disks, addresses can be read in any order. Most bulk memory in modern systems is **dynamic RAM (DRAM)**. DRAM is very dense; however, it does require that its values be **refreshed** periodically because the values inside the memory cells decay over time.

Basic DRAM organization

Although the basic organization of memories is simple, several variations exist that provide trade-offs [Cup01]. As shown in Fig. 4.17, a simple memory is organized as a two-dimensional array. Assume, for the moment, that the memory is accessed one bit at a time. The address for that bit is split into two sections: rows and columns. Together, they form the complete address in the array. If we want to access more than one bit at a time, we can use fewer bits in the column part of the address to select several columns simultaneously. The division of an address into rows and columns is important because it is reflected at the pins of the memory chip and is thus visible to the rest of the system. In a traditional DRAM, the row is sent first, followed by the column. Two control signals tell the DRAM when those address bits are valid: row address select or RAS and column address select or CAS.

Refreshing

The DRAM must be refreshed periodically to retain its values. Rather than refreshing the entire memory at once, DRAMs refresh a part of the memory at a time. When a section of the memory is being refreshed, it can't be accessed until the refresh is complete. The DRAM cycles through the memory, refreshing it section by section.

Bursts and page mode

Memories may offer some special modes that reduce the time required for access. Bursts and page mode accesses are both more efficient forms of access, but they differ in how they work. Burst transfers perform several accesses in sequence using a single

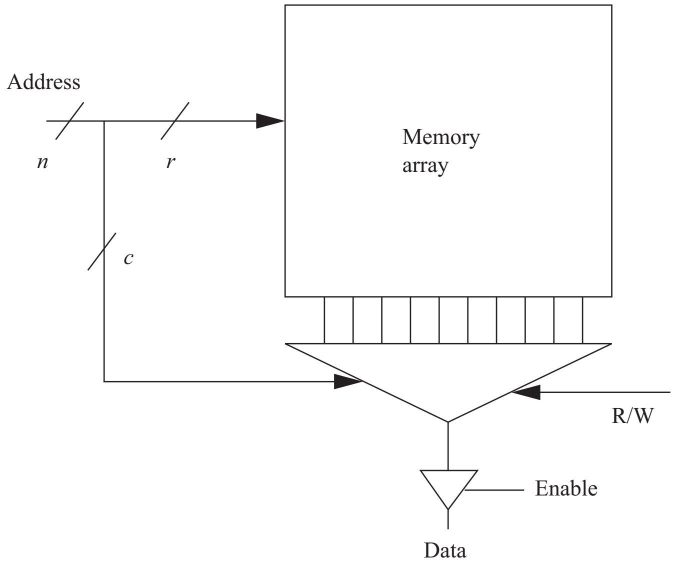


FIGURE 4.17

Organization of a basic memory.

address and possibly a single CAS signal. Page mode, in contrast, requires a separate address for each data access.

Types of DRAM

The most common type of DRAM is synchronous DRAM (SDRAM). A separate family of standards provides synchronous DRAM for graphics applications.

Synchronous dynamic RAM

SDRAMs use RAS and CAS signals to break the address into two parts, which select the proper row and column in the RAM array. Signal transitions are relative to the SDRAM clock, which allows the internal SDRAM operations to be pipelined. As shown in Fig. 4.18, transitions in the control signals are related to a clock [Mic00]. SDRAMs include registers that control the mode in which the SDRAM operates. SDRAMs support burst modes that allow several sequential addresses to be accessed by sending only one address. SDRAMs generally also support an interleaved mode that exchanges pairs of bytes.

Memory packaging

Memory for PCs is generally purchased as **single in-line memory modules (SIMMs)** or **double in-line memory modules (DIMMs)**. A SIMM or DIMM is a small circuit board that fits into a standard memory socket. A DIMM has two sets of leads compared with a SIMM's one. Memory chips are soldered to the circuit board to supply the desired memory.

ROMs are preprogrammed with fixed data and are useful in embedded systems because a great deal of the code and perhaps some data do not change over time. **Flash memory** is the dominant form of ROM. Flash memory can be erased and rewritten using standard system voltages, allowing it to be reprogrammed inside a typical system. This allows applications, such as automatic distribution of upgrades, wherein the

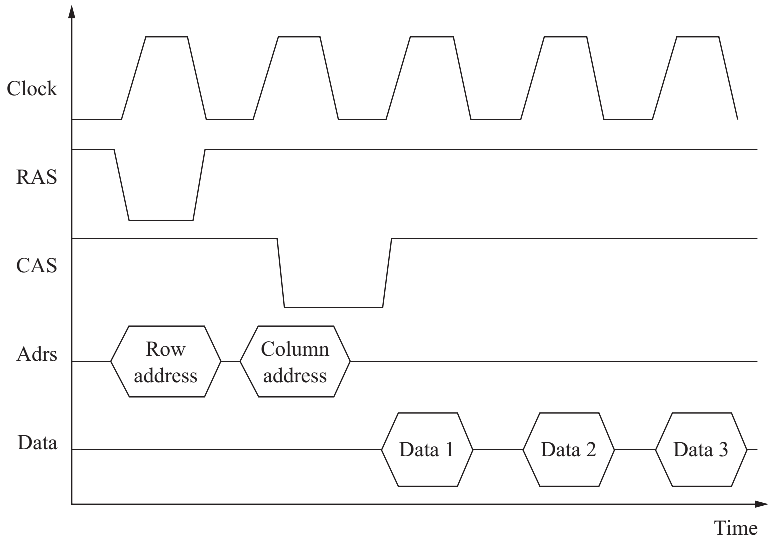


FIGURE 4.18

An synchronous dynamic RAM read operation.

flash memory can be reprogrammed while downloading the new memory contents from a telephone line. Early flash memories had to be erased in their entirety; modern devices allow memory to be erased in blocks. Most flash memories today allow certain blocks to be protected. A common application is to keep the boot-up code in a protected block while allowing updates to other memory blocks on the device. As a result, this form of flash is commonly known as **boot-block flash** [Int03]. **One-time programmable** (OTP) memory is based on a different storage cell (an anti-fuse) that can be programmed once. OTP memories are typically small in capacity.

4.4.1 Memory system organization

Modern memory is more than a one-dimensional array of bits. Memory chips have surprisingly complex organizations that allow us to make useful optimizations. For example, memories are usually divided into several smaller memory arrays.

Memory controllers

Modern computer systems use a memory controller as the interface between the CPU and the memory components. As shown in Fig. 4.19, the memory controller shields the CPU from knowledge of the detailed timing of different memory components. If the memory also consists of several different components, the controller will manage all accesses to all memories. Memory accesses must be scheduled. The memory controller receives a sequence of requests from the processor. However, it may not be possible to execute them as quickly as they are received if the memory component is already processing access. When faced with more accesses than the resources

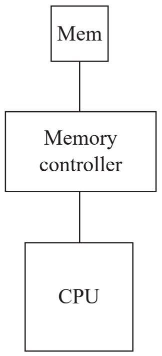


FIGURE 4.19

The memory controller in a computer system.

available to complete them, the memory controller will determine the order in which they will be handled and schedule the accesses accordingly.

Channels and banks

Channels and banks are two ways to add parallelism to the memory system. A channel is a connection to a group of memory components. If the CPU and memory controller can support multiple channels that operate concurrently, we can perform multiple independent accesses using the different channels. We may also divide the complete memory system into banks. Banks can perform accesses in parallel because each bank has its own memory arrays and addressing logic. By properly arranging memory into banks, we can overlap some of the access times for these locations and reduce the total time required for the complete set of accesses.

Fig. 4.20 shows a memory system organized into channels and banks. Each channel has its own memory components and connection to the processor. Channels operate completely separately. The memory in each channel can be subdivided into banks. The banks in a channel can be accessed separately. Channels are, in general, more

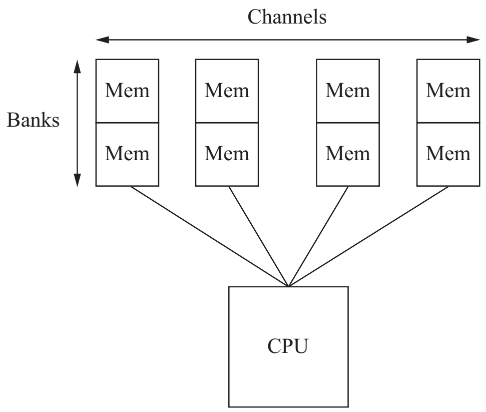


FIGURE 4.20

Channels and banks in a memory system.

expensive than banks. A two-channel memory system, for example, requires twice as many pins and wires connecting the CPU and memory as a one-channel system does. Memory components are often separated internally into banks, providing that access to the outside is less expensive.

4.5 I/O devices

While a wide range of I/O devices can be built for computer systems, a few basic types provide important functionalities for embedded computing systems.

Counter

A **counter** is a register with logic to count. It can be used to count a wide range of inputs and events. A counter may provide both up and down counts. Most counters will provide a reset input to reset the count to a known value, typically zero.

Timer

A **timer** is a counter whose count is driven by a periodic signal. Timers are used to time many different types of system activities. In Chapter 6, we will see how timers help with the operation of a real-time operating system.

Real-time clock

A **real-time clock** (**RTC**) is a counter whose output corresponds to the clock time. The RTC typically keeps track of time relative to a fixed time standard. Software can then be used to convert the RTC value to human-usable time.

General-purpose I/O

General-purpose I/O (GPIO) pins are just what the name implies; they can be used for a wide range of input or output functions. A GPIO pin can be configured as either input or output; it may also be disabled or put into a high-impedance mode. A more sophisticated GPIO interface may allow the pin to be configured to different logic levels. Software can write a value to the GPIO pin when it is in output mode, and the value is held until it is changed. When the pin is in input mode, the software can read the pin's value.

Data conversion

Data conversion refers to the translation between analog and digital values. Two important characteristics of data conversion are as follows:

- the **rate** at which conversion can be performed;
- the **precision** in bits of the conversion; and
- the **accuracy** of the conversion.

Analog-to-digital converter

An analog-to-digital converter (**ADC or A/D converter)** translates an analog value into a digital value. Many different techniques for ADC have been developed that provide a wide range of speeds, precisions, and costs. Generally, a high-speed, high-precision analog-to-digital conversion is expensive.

The next two examples compare ADCs intended for use in different applications.

Example 4.1: Texas Instruments ADC12xJ1600-Q1

Texas Instruments ADC12xJ1600-Q1 [Tex20A] can be used for automotive radar systems. It provides a maximum analog/digital sample rate of 1.6 gigasamples per second (GSPS) at 12 bits of resolution. Its signal-to-noise ratio at 100 MHz is 57.4 dB. The full-scale input voltage is 800 mV. A JESD204C serial data interface is used to retrieve data.

Example 4.2: Texas Instruments ADS126x

Texas Instruments ADS126x [Tex21] is useful in applications that require high accuracy: weight scales, thermocouples, strain-gauge sensors, and so on. It provides two levels of analog-to-digital conversion: a 32-bit precision converter and a 24-bit auxiliary converter. Both precision and auxiliary converters use sigma-delta architecture. The converter operates from 2.5 to 38.4K samples per second. It provides high accuracy: low offset and gain drift, low noise, and high linearity.

Digital-to-analog converter

A **digital-to-analog converte**r (**DAC** or **D/A converter**) converts a digital value to an analog value that may be expressed as a voltage or current. Digital-to-analog conversion is conceptually more straightforward than the analog-to-digital case, but high-accuracy DACs are generally more expensive.

The next two examples compare digital-to-analog converters intended for different applications.

Example 4.3: Texas Instruments AFE7422

Texas Instruments AFE7422 [Tex19] is designed for radio-frequency operations, such as phased array radar; it combines both DAC and ADC. It provides two 14-bit, 9 GSPS DACs and two 14-bit, 3 GSPS ADCs. It operates over a frequency range of 10 MHz to 6 GHZ. Both the transmit and receive signals paths provide configurable digital processing.

Example 4.4: Texas Instruments DACx1001

Texas Instruments DACx1001 [Tex20B] provides high accuracy and low noise for instrumentation, such as semiconductor testing and medical radiology. Three versions are available in 20-, 18-, and 16-bit resolution. A four-wire serial interface provides digital access.

Buses

Directly connecting an I/O device to a high-speed CPU bus is not always a cost-effective solution. Lower-speed, lower-cost buses can be used to connect to simple, low-cost devices. The next example describes a simple bus used for audio data.

Example 4.5: I^2S Bus

The I^2S bus [Phi96] is a standard interface for digital audio. An audio system may perform several processing steps on an audio signal using several different chips. I^2S provides a standard, low-cost interface.

I^2S is designed to carry stereo data with the two channels alternating. The I^2S bus uses three signals:

- SCK is the system clock. A bus master provides the clock signal.
- WS is word select. The word select line determines which channel of audio is on the serial data line: low for left audio and high for right audio.

- Serial data provide the data encoded as two's complement with the most significant bit first. The standard does not define the number of bits in a sample; the end of a sample is signaled by a change in WS.

4.6 Designing with computing platforms

In this section, we concentrate on creating a working embedded system based on a computing platform. We first look at some example platforms and what they include. We then consider how to choose a platform for an application and how to make effective use of that platform.

4.6.1 Example platforms

The design complexity of the hardware platform can vary greatly, from a totally off-the-shelf solution to a highly customized design. A platform may comprise one chip or even dozens.

Open-source platforms

Fig. 4.21 shows a BeagleBoard [Bea11], an open-source, low-cost platform for embedded system projects. The processor is an Arm Cortex-A8, which also comes with several built-in I/O devices. The board includes many connectors and supports a variety of I/O: flash memory, audio, video, and so forth. The support environment provides basic information about the board design, such as schematics, a variety of software development environments, and many example projects built with the BeagleBoard.

Developer and evaluation boards

Chip vendors often provide their own evaluation boards or modules for their chips. The evaluation board may be a complete solution, or it may provide what you need, with only slight modifications. The hardware design (e.g., netlist and board layout) is typically available from the vendor; companies provide this information to make it easy for customers to use their microprocessors. If the evaluation board does not completely meet your needs, you can modify the design using the netlist and board layout without starting from scratch. Vendors generally do not charge royalties for hardware board design.

Fig. 4.22 shows a Jetson Nano developer kit [Fra19]. The processor includes a quad-core Arm core, a 128-core NVIDIA Maxwell GPU, and video encoder and decoder. I/O includes four USB 3.0 ports, a MIPI camera port, an HDMI port, and Gigabit Ethernet. The system boots off a microSD card. A Linux-based development system can be run on the board.

Fig. 4.23 shows an Arm evaluation module. Like the BeagleBoard, this evaluation module includes a basic platform chip and a variety of I/O devices. However, the main purpose of BeagleBoard is as an end-use, low-cost board, and the evaluation module is primarily intended to support software development and serve as a starting point for a more refined product design. As a result, this evaluation module includes some features that would not appear in the final product, such as the connections to the processor's pins that surround the processor chip itself.

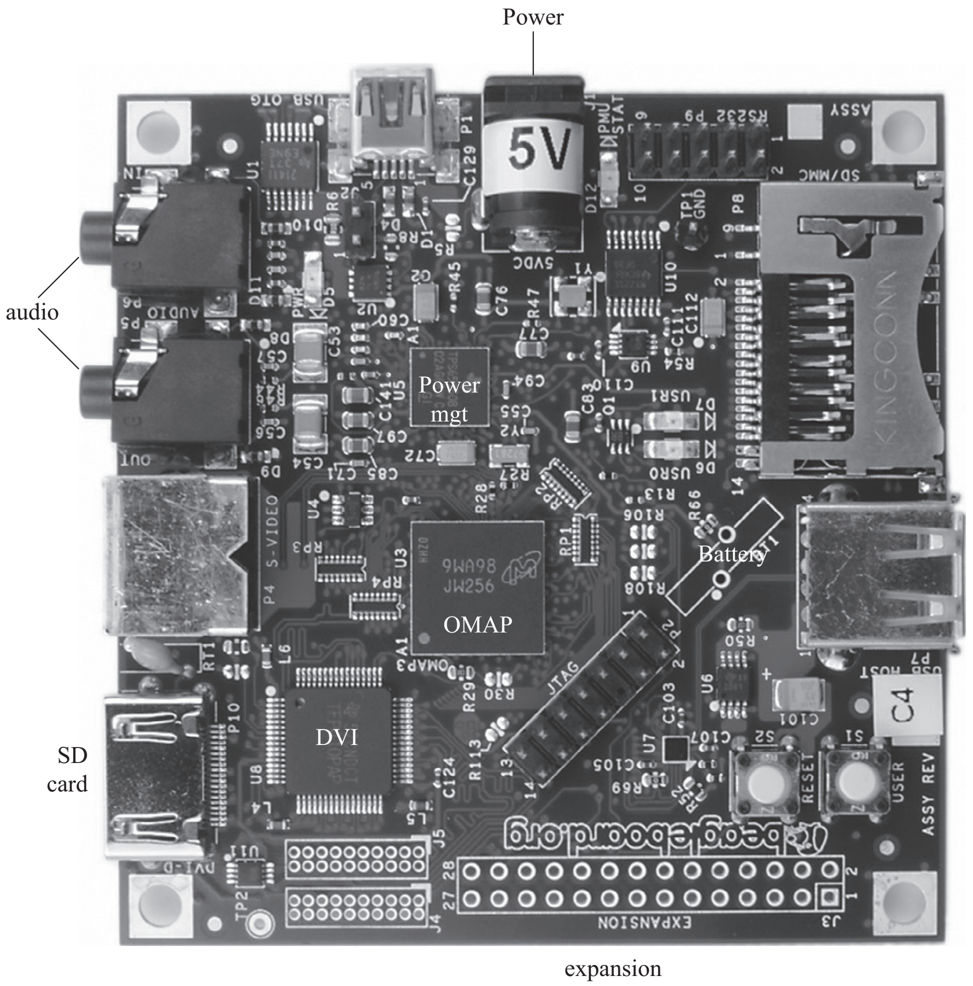


FIGURE 4.21

A BeagleBoard.

4.6.2 Choosing a platform

We will probably not design the platform for our embedded system from scratch. We may assemble hardware and software components from several sources, and we may also acquire a complete hardware/software platform package. Several factors will contribute to your decision to use a particular platform.

Hardware

The hardware architecture of the platform is the more obvious manifestation of the architecture, because you can touch it and feel it. The various components may all play a role in the suitability of the platform.

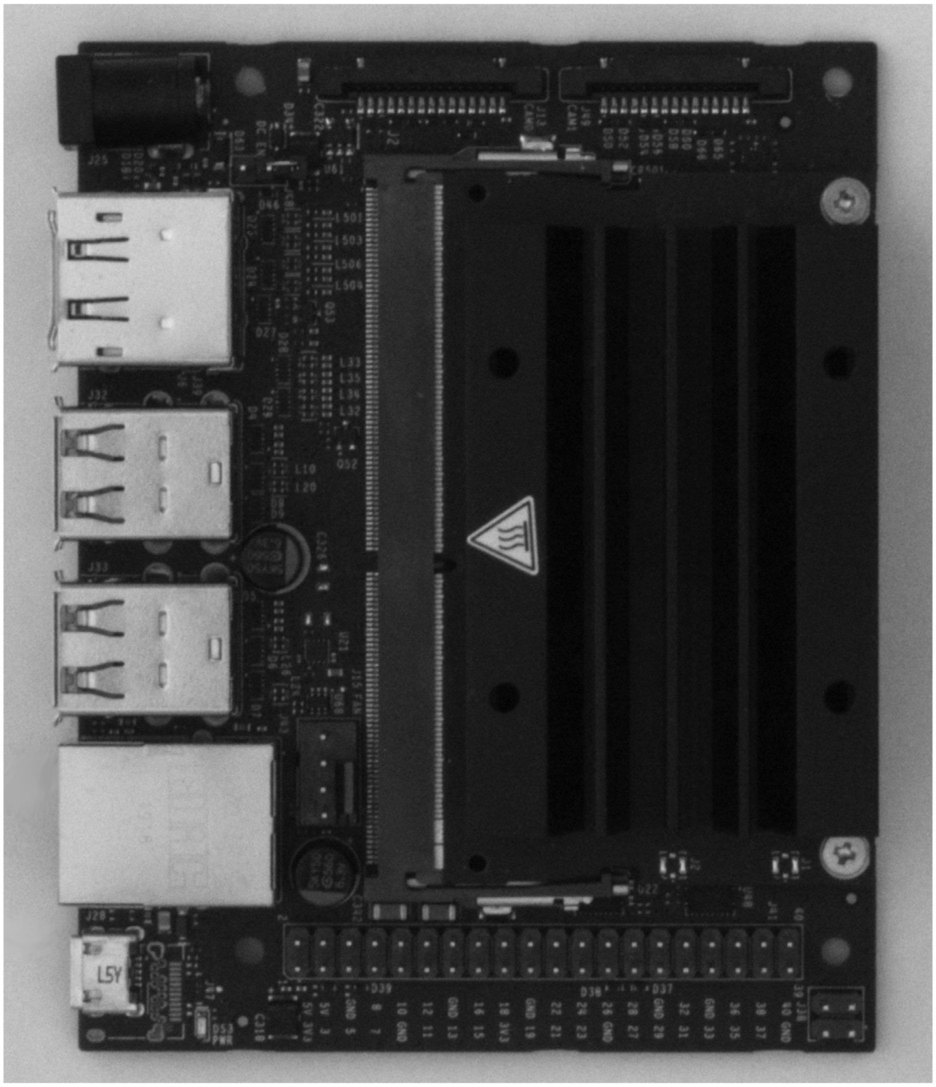


FIGURE 4.22

Jetson Nano Developer Kit.

- *CPU:* An embedded computing system clearly contains a microprocessor. But which one? There are many different architectures, and even within an architecture, we can select from among models that vary in clock speed, bus data width, integrated peripherals, and so on. The choice of CPU is one of the most important decisions, but it cannot be made without considering the software that will be executed on the machine.

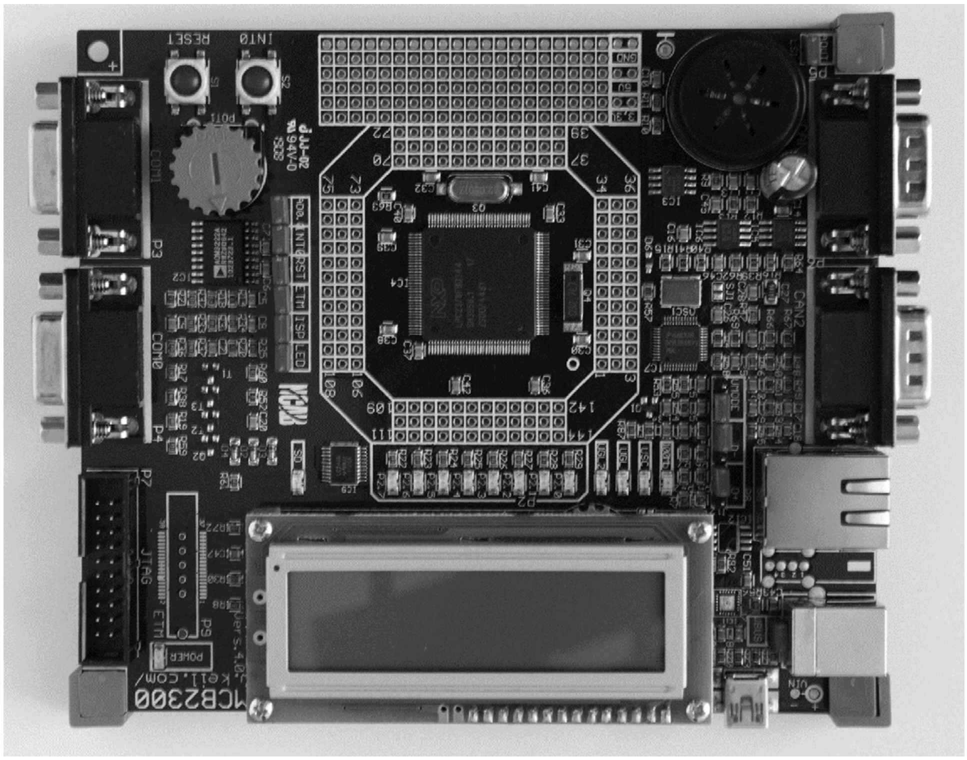


FIGURE 4.23

An Arm evaluation module.

- *Bus:* The choice of a bus is closely tied to that of a CPU because the bus is an integral part of the microprocessor. However, in applications that make intensive use of the bus owing to I/O or other data traffic, the bus may be more of a limiting factor than the CPU. Attention must be paid to the required data bandwidths to ensure that the bus can handle the traffic.
- *Memory:* Once again, the question is not whether the system will have memory, but the characteristics of that memory. The most obvious characteristic is the total size, which depends on both the required data volume and the size of the program instructions. The ratio of ROM to RAM and the selection of DRAM versus SRAM can have a significant influence on the cost of the system. The speed of the memory plays a large part in determining system performance.
- *Input and output devices:* If we use a platform built out of many low-level components on a printed circuit board, we may have a great deal of freedom in the I/O devices connected to the system. Platforms based on highly integrated chips only come with certain combinations of I/O devices. The combination of I/O devices

may be a prime factor in platform selection. We may need to choose a platform that includes some I/O devices we do not need to get the devices that we do need.

Software

When we think about the software components of the platform, we generally think about both the run-time components and the support components. Run-time components become part of the final system: the operating system, code libraries, and so on. Support components include the code development environment, debugging tools, and so on.

Run-time components are a critical part of the platform. An operating system is required to control the CPU and its multiple processes. A file system is used in many embedded systems to organize internal data and as an interface with other systems. Many complex libraries—digital filtering and fast Fourier transform—provide highly optimized versions of complex functions.

Support components are critical to making use of complex hardware platforms. Without proper code development and operating systems, the hardware itself is useless. Tools may come directly from the hardware vendor, from third-party vendors, or from developer communities.

4.6.3 Intellectual property

Intellectual property (IP) is something that we can own, but not touch: software, netlists, and so on. Just as we need to acquire hardware components to build our system, we also need to acquire IP to make that hardware useful. Here are some examples of the wide range of IP that we use in embedded system design:

- run-time software libraries;
- software development environments; and
- schematics, netlists, and other hardware design information.

IP can come from many different sources. We may buy IP components from vendors. For example, we may buy a software library to perform certain complex functions and incorporate that code into our system. We may also obtain it from developer communities online.

Example 4.6 looks at the IP available for the BeagleBoard.

Example 4.6: BeagleBoard IP

The BeagleBoard Web site (http://www.beagleboard.org) contains both hardware and software IP. The hardware IP includes:

- schematics for the printed circuit board;
- artwork files (known as Gerber files) for the printed circuit board; and
- a bill of materials that lists the required components.

Software IP includes:

- a compiler for the processor
- a version of Linux for the processor

4.6.4 Development environments

Although we may use an evaluation board, much of the software development for an embedded system is done on a PC or workstation known as the **host**, as illustrated in Fig. 4.24. The hardware upon which the code will finally run is known as the **target.** The host and target are frequently connected by a USB link, but a higher-speed link, such as Ethernet, can be used.

The target must include a small amount of software to talk to the host system. That software will take up some memory, interrupt vectors, and so on, but it should generally leave the smallest possible footprint in the target to avoid interfering with the application software. The host should be able to do the following:

- load programs into the target;
- start and stop program execution on the target; and
- examine memory and CPU registers.

A **cross-compiler** runs on one type of machine but generates code for another. After compilation, the executable code is typically downloaded to the embedded system via USB. We also often make use of host-target debuggers, in which the basic hooks for debugging are provided by the target, and a more sophisticated user interface is created by the host.

We often create a **testbench program** that can be built to help debug the embedded code. The testbench generates inputs to stimulate a piece of code and compares the outputs against expected values, providing valuable early debugging help. The embedded code may need to be slightly modified to work with the testbench, but careful coding, such as using the #ifdef directive in C, can ensure that the changes can be undone easily and without introducing bugs.

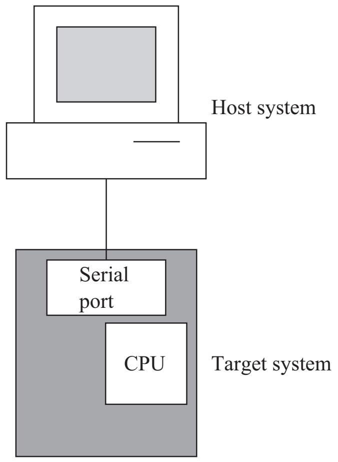


FIGURE 4.24

Connecting a host and target system.

4.6.5 Watchdog timers

The **watchdog** is a useful technique for monitoring the system during operation. Watchdogs, when used properly, can help improve the system's safety and security.

The most basic form of a watchdog is a watchdog timer [Koo10]. This technique uses a timer to monitor the correct operation of the software. As shown in Fig. 4.25, the timer's rollover output (set to high when the count reaches zero) is connected to the system reset. The software is modified so that it reinitializes the counter, setting it back to its maximum count. The software must be modified so that every execution path reinitalizes the timer frequently, such that the counter never rolls over and resets the system. The timer's period determines how frequently the software must be designed to reinitialize the timer. A reset during operation indicates some type of software error or fault.

More generally, a watchdog processor monitors the operation of the main processor [Mah88]. For example, a watchdog processor can be programmed to monitor the control flow of a program executed on the main processor [Lu82].

Example 4.7 discusses the use of a watchdog timer on the Ingenuity Mars helicopter.

Example 4.7: Watchdog Timer Expiration on Ingenuity Mars Helicopter

The expiration of a watchdog timer on the Ingenuity Mars helicopter caused a delay in its first flight [NAS21A]. The watchdog expired during a high-speed spin test of the rotors on Sol 49 (April 9, 2021). The flight computer was in the midst of a transition from preflight to flight mode. This issue was addressed with a software update to modify the flight controller boot process [NAS21B].

4.6.6 Debugging techniques

A good deal of software debugging can be done by compiling and executing the code on a PC or workstation. However, at some point, running code on the embedded hardware platform is necessary. Embedded systems are usually less friendly programming environments than PCs. Nonetheless, the resourceful designer has several options available for debugging the system.

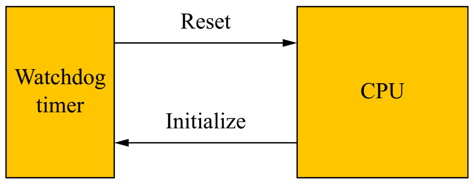


FIGURE 4.25

A watchdog timer in a system.

The USB port found on most evaluation boards is one of the most important debugging tools. In fact, it is often a good idea to design a USB port into an embedded system even if it will not be used in the final product; USB can be used not only for development debugging, but also for diagnosing problems in the field or field upgrades of software.

Another important debugging tool is the **breakpoint.** The simplest form of a breakpoint is for the user to specify an address at which the program's execution is to break. When the PC reaches that address, the control is returned to the monitor program. From the monitor program, the user can examine and/or modify the CPU registers, after which execution can be continued. Implementing breakpoints does not require using exceptions or external devices.

Programming Example 4.1 shows how to use instructions to create breakpoints.

Programming Example 4.1: Breakpoints

A breakpoint is a location in the memory at which a program stops executing and returns to the debugging tool or monitor program. Implementing breakpoints is very simple; you simply replace the instruction at the breakpoint location with a subroutine call to the monitor. In the following code, to establish a breakpoint at location 0x40c in some Arm code, we've replaced the branch (B) instruction normally held at that location with a subroutine call (BL) to the breakpoint handling routine:

When the breakpoint handler is called, it saves all registers, and can then display the CPU state to the user and take commands.

To continue execution, the original instruction must be replaced in the program. If the breakpoint can be erased, the original instruction can simply be replaced, and control is returned to that instruction. This will normally require fixing the subroutine return address, which will point to the instruction after the breakpoint. If the breakpoint is to remain, then the original instruction can be replaced, and a new temporary breakpoint is placed at the next instruction (taking jumps into account, of course). When the temporary breakpoint is reached, the monitor puts back the original breakpoint, removes the temporary one, and resumes execution.

The Unix *dbx* debugger shows the program being debugged in source code form, but that capability is too complex to fit into most embedded systems. Very simple monitors will require you to specify the breakpoint as an absolute address, which will require you to know how the program was linked. A more sophisticated monitor will read the symbol table and allow you to use labels in the assembly code to specify locations.

LEDs as debugging devices

Never underestimate the importance of light-emitting diodes (LEDs) in debugging. As with serial ports, it is often a good idea to design in a few to indicate the system state, even if they will not normally be seen in use. LEDs can be used to show error conditions, when the code enters certain routines, or to show idle-time activity. LEDs can be entertaining as well; a simple flashing LED can provide a great sense of accomplishment when it first starts to work.

In-circuit emulation

When software tools are insufficient to debug the system, hardware aids can be deployed to give a clearer view of what is happening when the system is running.

The **microprocessor in-circuit emulator (ICE)** is a specialized hardware tool that can help to debug software in a working embedded system. At the heart of an ICE is a special version of the microprocessor that allows its internal registers to be read out when they are stopped. The ICE surrounds this specialized microprocessor with additional logic that allows the user to specify breakpoints and examine and modify the CPU state. The CPU provides as much debugging functionality as a debugger within a monitor program, but it does not take up any memory. The main drawback of in-circuit emulation is that the machine is specific to a particular microprocessor, even down to the pinout. If you use several microprocessors, maintaining a fleet of ICEs to match can be exorbitant.

Logic analyzers

The **logic analyzer** [Ald73] is the other major piece of instrumentation in the embedded system designer's arsenal. Think of a logic analyzer as an array of inexpensive oscilloscopes; the analyzer can sample many different signals simultaneously (tens to hundreds), but can display only 0, 1, or changing values for each. All these logic analysis channels can be connected to the system to record the activity on many signals simultaneously. The logic analyzer records the values of the signals into an internal memory, and then, displays the results on a display once the memory is full or the run is aborted. The logic analyzer can capture thousands or even millions of samples of data on all of these channels, providing a much larger time window into the operation of the machine than is possible with a conventional oscilloscope.

A typical logic analyzer can acquire data in either of two modes, which are typically called **state** and **timing modes.** To understand why the two modes are useful and the difference between them, it is important to remember that a logic analyzer trades reduced resolution on the signals for the longer time window. The measurement resolution of each signal is reduced in both voltage and time dimensions. The reduced voltage resolution is accomplished by measuring logic values (0, 1, x) rather than analog voltages. The reduction in timing resolution is accomplished by sampling the signal, rather than capturing a continuous waveform, as in an analog oscilloscope.

The state and timing modes represent different ways of sampling the values. Timing mode uses an internal clock that is fast enough to take several samples per clock period in a typical system. State mode, on the other hand, uses the system's own clock to control sampling; thus, it samples each signal only once per clock cycle. As a result, the timing mode requires more memory to store a given number of system clock cycles. On the other hand, it provides a greater resolution in the signal for detecting glitches. The timing mode is typically used for glitch-oriented debugging, whereas the state mode is used for sequentially oriented problems.

The internal architecture of a logic analyzer is shown in Fig. 4.26. The system's data signals are sampled at a latch within the logic analyzer; the latch is controlled by either the system clock or the internal logic analyzer sampling clock, depending on whether the analyzer is being used in the state or timing mode. Each sample is copied into a vector memory under the control of a state machine. The latch, timing circuitry, sample memory, and controller must be designed to run at high speed because several samples per system clock cycle may be required in timing mode.

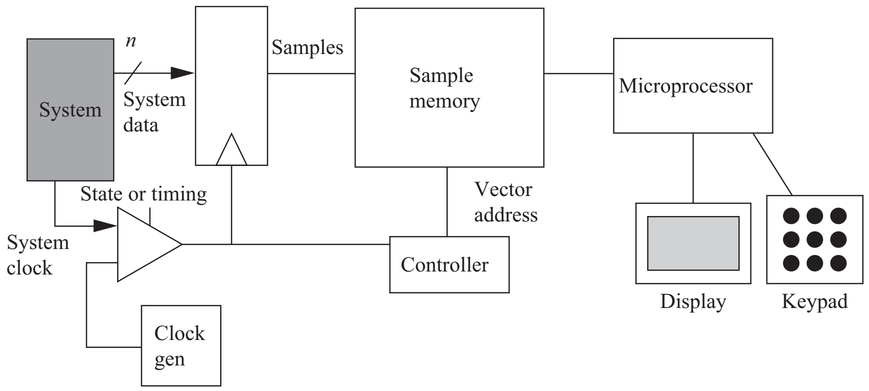


FIGURE 4.26

Architecture of a logic analyzer.

After the sampling is complete, an embedded microprocessor takes over to control the display of the data captured in the sample memory.

Logic analyzers typically provide several formats for viewing data. One format is a timing diagram. Many logic analyzers allow not only customized displays, such as giving names to signals, but also more advanced display options. For example, an inverse assembler can be used to turn vector values into microprocessor instructions. The logic analyzer does not provide access to the internal state of the components, but it does give a particularly good view of externally visible signals. This information can be used for both functional and timing debugging.

Some modern logic analyzers are structured as USB devices; the device includes the data acquisition circuitry, while the host computer serves as the user interface.

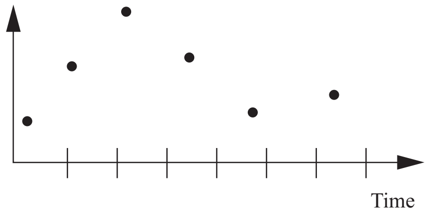
4.6.7 Debugging challenges

Logical errors in software can be hard to track, but errors in real-time code can create problems that are even harder to diagnose. Real-time programs are required to finish their work within a certain amount of time; if they run too long, they can create unexpected behavior.

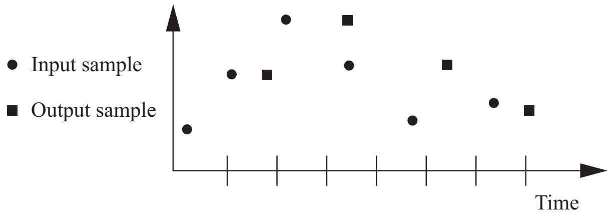
Example 4.8 demonstrates one of the problems that can arise.

Example 4.8: A Timing Error in Real-time Code

Let's consider a simple program that periodically takes an input from an analog/digital converter, does some computations on it, and then, outputs the result to a digital/analog converter. To make it easier to compare input to output and see the results of the bug, we assume that the computation produces an output equal to the input but that a bug causes the computation to run 50% longer than its given time interval. A sample input to the program over several sample periods looks like this:



If the program ran fast enough to meet its deadline, the output would simply be a time-shifted copy of the input. However, when the program runs over its allotted time, the output will become quite different. Exactly what happens depends, in part, on the behavior of the A/D and D/A converters. Therefore, let's make some assumptions. First, the A/D converter holds its current sample in a register until the next sample period, and the D/A converter changes its output whenever it receives a new sample. Next, a reasonable assumption about interrupt systems is that, when an interrupt is not satisfied and the device interrupts again, the device's old value will disappear and be replaced by the new value. The basic situation that develops when the interrupt routine runs too long is something like this:



1. The A/D converter is prompted by the timer to generate a new value, saves it in the register, and requests an interrupt.
2. The interrupt handler runs too long from the last sample.
3. The A/D converter gets another sample in the next period.
4. The interrupt handler finishes its first request, and then, immediately responds to the second interrupt. It never sees the first sample and only gets the second one.

Thus, assuming that the interrupt handler takes 1.5 times longer than it should, here is how it would process the sample input:

The output waveform is seriously distorted because the interrupt routine grabs the wrong samples and puts the results out at the wrong times.

The exact results of missing real-time deadlines depend on the detailed characteristics of the I/O devices and the nature of the timing violation. This makes debugging real-time problems especially difficult. Unfortunately, the best advice is that, if a system exhibits truly unusual behavior, missed deadlines should be suspected. ICEs, logic analyzers, and even LEDs can be useful tools for checking the execution time of real-time code to determine whether it in fact meets its deadline.

4.7 Embedded file systems

Flash memory

Many consumer electronics devices use **flash memory** for mass storage. Flash memory is a type of semiconductor memory that, unlike DRAM and SRAM, provides permanent storage. Values are stored in the flash memory cell as an electric charge using a specialized capacitor that can store the charge for years. Flash memory cell does not require an external power supply to maintain its value. Furthermore, the memory can be written electrically, and unlike previous generations of electrically erasable semiconductor memory, can be written using standard power supply voltages, and thus, does not need to be disconnected during programming.

SSDs

A solid-state drive (SSD) is a storage device built from flash memory or another form of nonvolatile solid-state device. An SSD is an I/O device with its own controller, which acts as an interface between the drive storage and the rest of the system.

Flash file systems

A flash memory has one important limitation that must be considered. Writing a flash memory cell causes mechanical stress that eventually wears out the cell. Today's flash memories can reliably be written a million times, but will, at some point, fail. Although a million write cycles may sound like a lot, creating a single file may require many write operations, particularly for the part of the memory that stores the directory information.

A wear-leveling flash file system [Ban95] manages the use of flash memory locations to equalize wear, while maintaining compatibility with existing file systems. A simple model of a standard file system has two layers: the bottom layer handles physical reads and writes on the storage device, and the top layer provides a logical view of the file system. A flash file system imposes an intermediate layer that allows the logical-to-physical mapping of files to be changed. This layer keeps track of how frequently different sections of the flash memory have been written and allocates data to equalize wear. It may also move the location of the directory structure while the file system is operating. Because the directory system receives the most wear, keeping it in one place may cause part of the memory to wear out before the rest, unnecessarily reducing the useful life of the memory device. Several flash file systems have been developed, such as the Yet Another Flash Filing System (YAFFS) [Yaf11].

DOS file systems

DOS file allocation table (**FAT**) file systems refer to the file system developed by Microsoft for early versions of the DOS [Mic00]. FAT32 file systems can interoperate with a wide range of operating systems and can be implemented in a relatively small amount of code. Wear-leveling algorithms for flash memory can be implemented without disturbing the basic operation of the file system.

4.8 Platform-level performance analysis

Bus-based systems add another layer of complications to performance analysis. Platform-level performance involves much more than the CPU. We often focus on the CPU because it processes instructions, but any part of the system can affect the

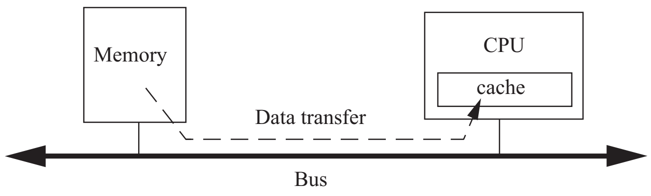


FIGURE 4.27

Platform-level data flows and performance.

total system performance. More precisely, the CPU provides an upper bound on performance, but any other part of the system can slow down the CPU. Merely counting instruction execution times is not enough.

Consider the simple system shown in Fig. 4.27. We want to move data from the memory to the CPU to process it. To get the data from the memory to the CPU, we must:

- read from the memory;
- transfer over the bus to the cache; and
- transfer from the cache to the CPU.

The time required to transfer from the cache to the CPU is included in the instruction execution time, but the other two times are not included.

Bandwidth as performance

The most basic measure of performance we are interested in is **bandwidth**—the rate at which we can move data. Ultimately, if we are interested in real-time performance, we are interested in real-time performance measured in seconds. However, often, the simplest way to measure performance is in units of clock cycles. However, different parts of the system run at different clock rates. We must ensure that we apply the right clock rate to each part of the performance estimate when we convert from clock cycles to seconds.

Bus bandwidth

Bandwidth questions often arise when we are transferring large blocks of data. For simplicity, let's start by considering the bandwidth provided by only one system component: the bus. Consider an image of 1920×1080 pixels with each pixel composed of 3 bytes of data. This gives a grand total of 6.2 megabytes. If these images are video frames, we want to check whether we can push one frame through the system within the 0.033 s that we have to process a frame before the next one arrives.

Let us assume that we can transfer 1 byte of data every microsecond, which implies a bus speed of 100 MHz. In this case, we would require 0.062 s to transfer one frame, or about half the rate required.

We can increase bandwidth in two ways: we can increase the clock rate of the bus, or we can increase the amount of data transferred per clock cycle. For example, if we increased the bus to carry 4 bytes or 32 bits per transfer, we would reduce the transfer time to 0.015 s at the original 100 MHz clock rate. Alternatively, if we could increase the bus clock rate to 200 MHz, then we would reduce the transfer time to 0.031 s, which is within our time budget for the transfer.

Bus bandwidth characteristics

How do we know how long it takes to transfer one unit of data? To determine this, we must look at the data sheet for the bus. A bus transfer generally takes more than one clock cycle. Burst transfers, which move blocks of data to contiguous locations, may be more efficient per byte. We also need to know the width of the bus—how many bytes per transfer. Finally, we need to know the bus clock period, which will generally be different from the CPU clock period.

Bus bandwidth formulas

We can write formulas that tell us how long a transfer of N words will take, given a bus with a transfer width of one word. We will write our basic formulas in units of bus cycles, T, and then, convert those bus cycle counts to real time t using the bus clock period, P:

$$t = TP \qquad \text{(Eq. 4.1)}$$

First, consider transferring one word at a time with nonburst bus transactions. As shown in Fig. 4.28, a basic bus transfer of N bytes transfers one word per bus transaction. A single transfer takes D clock cycles. Ideally, $D = 1$, but a memory that introduces wait states is one example of a transfer that could require $D > 1$ cycles. Addresses, handshaking, and other activities constitute overhead that may occur before (O_1) or after (O_2) the data. For simplicity, we lump the overhead into $O = O_1 + O_2$. This gives a total transfer time in clock cycles of

$$T_{basic}(N) = (O + D)N \qquad \text{(Eq. 4.2)}$$

Now consider a burst transaction with a burst transaction length of B words, as shown in Fig. 4.29. As before, each of those transfers will require D clock cycles, including any wait states. The bus also introduces O_B cycles of overhead per burst. This gives

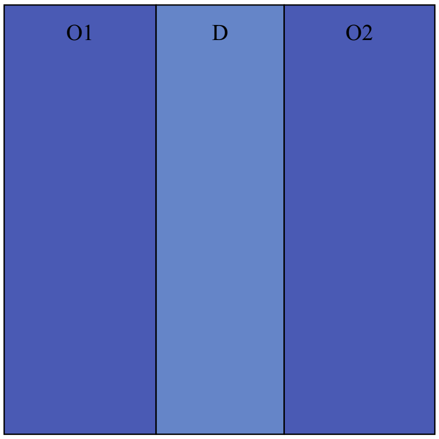


FIGURE 4.28

Times and data volumes in a basic bus transfer.

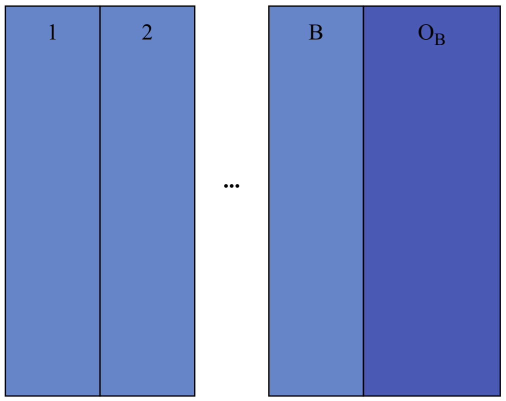


FIGURE 4.29

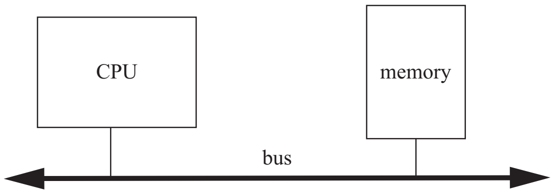
Times and data volumes in a burst bus transfer.

$$T_{burst}(N) = \frac{N}{B}(BD + O_B) \qquad \text{(Eq. 4.3)}$$

The next example applies our bus performance models to a simple example.

Example 4.9: Performance Bottlenecks in a Bus-based System

Consider a simple bus-based system.



We want to transfer data between the CPU and the memory over the bus. We need to be able to read an HDTV 1920 × 1080, 3 bytes per pixel video frame into the CPU at the rate of 30 frames/s, for a total of 6.2 MB/s. Which will be the bottleneck and limit system performance: the bus or the memory?

Let's assume that the bus has a 100 MHz clock rate (period of 10^{-8} s) and is two bytes wide, with D = 1 and O = 3. The 2-B bus allows us to cut the number of bus operations in half. This gives a total transfer time of

$$T_{basic}(1920 \times 1080) = (3+1) \cdot \left(\frac{6.2 \times 10^6}{2}\right) = 12.4 \times 10^6 \, cycles$$

$$t_{basic} = T_{basic} P = 0.124s$$

Because the total time to transfer 1-s worth of frames is more than 1 s, the bus is not fast enough for our application.

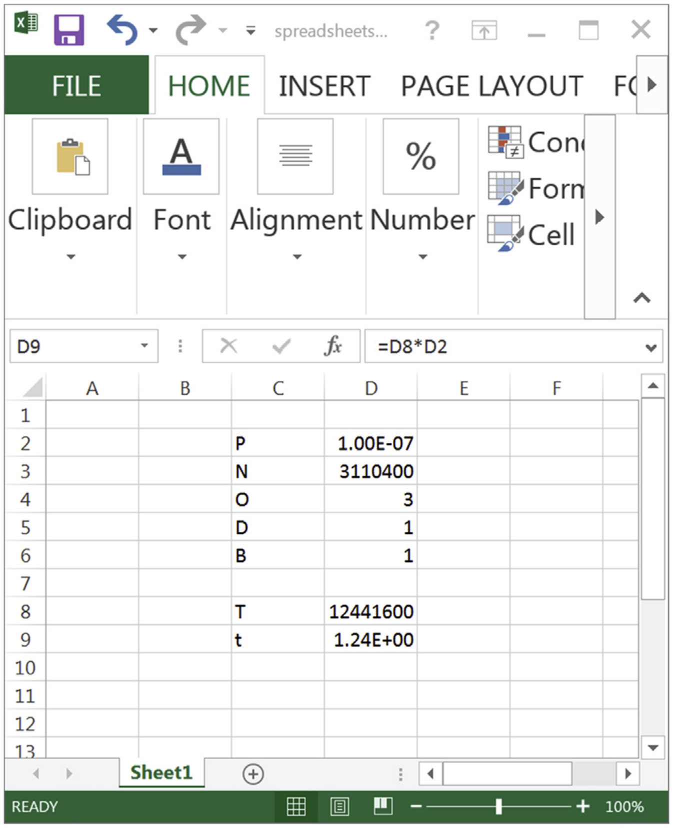
Alternatively, consider a memory that provides a burst mode with $B = 4$ and is two bytes wide. For this memory, $D = 1$ and $O = 4$, and assume that it runs at the same clock speed. The access time for this memory is 10 ns. Then,

$$T_{basic}(1920 \times 1080) = \frac{\left(6.2 \times 10^6/2\right)}{4}(4 \cdot 1 + 4) = 6.2 \times 10^6 cycles$$

$$t_{basic} = T_{basic}P = 0.062s$$

The memory is fast enough—it requires less than 1 s to transfer the 30 frames that must be transmitted in 1 s.

One way to explore design trade-offs is to build a spreadsheet with our bandwidth formulas.



	A	B	C	D	E	F
1						
2			P	1.00E-07		
3			N	3110400		
4			O	3		
5			D	1		
6			B	1		
7						
8			T	12441600		
9			t	1.24E+00		
10						
11						
12						
13						

We can change values, such as bus width and clock rate, and instantly see their effects on available bandwidth.

Component bandwidth

Bandwidth questions also come up in situations that we don't normally think of as communications. Transferring data into and out of components also raises questions of bandwidth. The simplest illustration of this problem is memory.

The width of a memory determines the number of bits that we can read from the memory in one cycle. This is a form of data bandwidth. We can change the types of memory components we use to change the memory bandwidth; we may also be able to change the format of our data to accommodate the memory components.

Memory aspect ratio

A single memory chip is not solely specified by the number of bits it can hold. As shown in Fig. 4.30, memories of the same size can have different **aspect ratios.** For example, a 1-Gbit memory that is 1-bit wide will use 30 address lines to present 2^{30} locations, each with 1-b of data. The same-size memory in a 4-bit-wide format will have 26 address lines, and an 8-bit-wide memory will have 22 address lines.

Memory chips do not come in extremely wide aspect ratios, but we can build wider memories by using several memories in parallel. By organizing memory chips into the proper aspect ratio for our application, we can build a memory system with the total amount of storage that we want, which presents the data width that we need.

The memory system width may also be determined by the memory modules we use. Rather than buying memory chips individually, we may buy memory as SIMMs or DIMMs. These memories are wide, but generally only come in standard widths.

Which aspect ratio is preferable for the overall memory system depends, in part, on the format of the data that we want to store in the memory and the speed with which it must be accessed, giving rise to bandwidth analysis.

Memory access times and bandwidth

We must also consider the time required to read or write a memory. Once again, we refer to the component data sheets to find these values. Access times depend quite a bit on the type of memory chip used. Page modes operate similarly to burst modes in buses. If the memory is not synchronous, we can still refer the times between events back to the bus clock cycle to determine the number of clock cycles required for access.

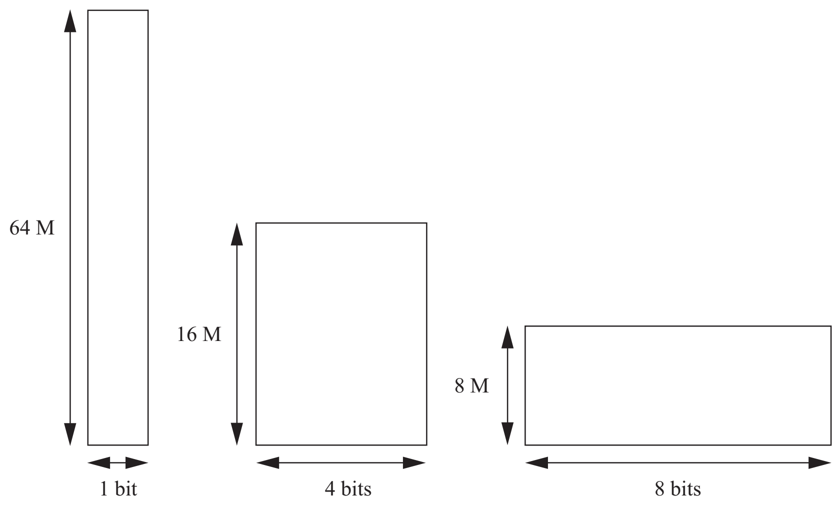


FIGURE 4.30

Memory aspect ratios.

4.9 Platform-level power management

ACPI

The **Advanced Configuration and Power Interface (ACPI)** [ACP13] is an open-industry standard for power management services. Initially targeted at PCs, it is designed to be compatible with a wide variety of operating systems. The role of ACPI in the system is illustrated in Fig. 4.31. The ACPI provides some basic power management facilities and abstracts the hardware layer. The operating system has its own power management module that determines the policy, and the operating system then uses ACPI to send the required controls to the hardware and to observe the hardware's state as input to the power manager.

The ACPI supports several basic global power states:

- G3, the mechanical off state, in which the system consumes no power;
- G2, the soft off state, which requires a full operating system reboot to restore the machine to working conditions;
- G1, the sleeping state, in which the system appears to be off, and the time required to return to working conditions is inversely proportional to power consumption;
- G0, the working state, in which the system is fully usable;

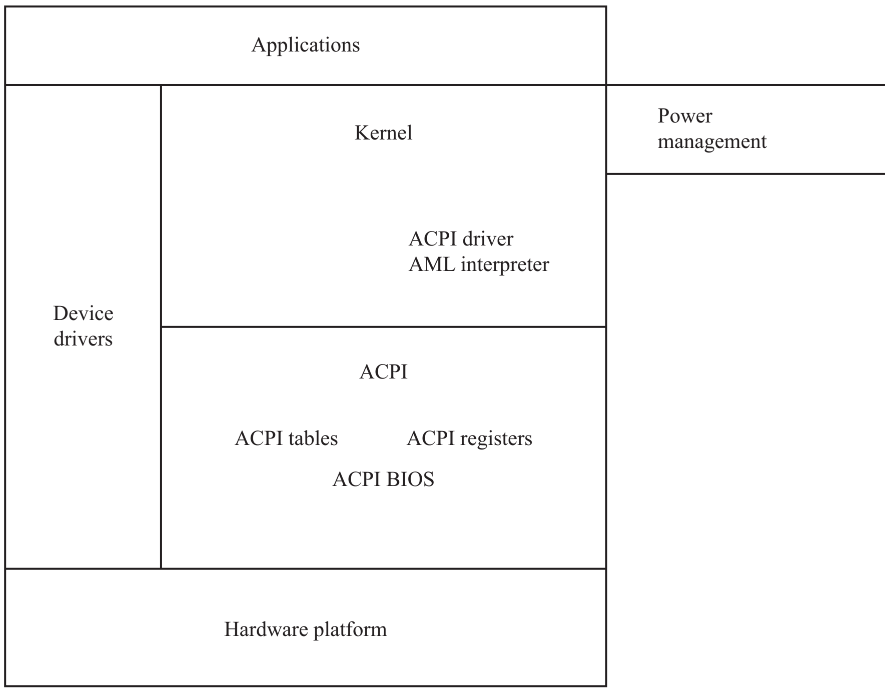


FIGURE 4.31

The Advanced Configuration and Power Interface and its relationship to a complete system.

- S4, a nonvolatile sleep, in which the system state is written to nonvolatile memory for later restoration; and
- the legacy state, in which the system does not follow the ACPI.

The power manager typically includes an observer, who receives messages through the ACPI that describe the system behavior. It also includes a decision module that determines power management actions based on these observations.

4.10 Platform security

Security and safety cannot be bolted on; they must be baked in. A basic understanding of security issues is important for every embedded system designer. This section discusses some basic security techniques and their use in the computing platform. We discuss programs and security in Section 5.11.

Cryptography

We make use of a few basic concepts in cryptography throughout the book [Sch96]: cryptography, public-key cryptography, hashing, and digital signatures.

Secret key cryptography

Cryptography is designed to encode a message so that it cannot be read directly by someone who intercepts the message; the code should also be difficult to break. Traditional techniques are known as **secret key cryptography** because they rely on the secrecy of the key used to encrypt the messages. The advanced encryption standard (AES) is a widely used encryption algorithm [ISO10]. It encrypts data in blocks of 128 bits and can use keys of three different sizes: 128, 192, or 256 bits. The SIMON block cipher [Bea13] was developed as a lightweight cipher. It operates on blocks of several sizes ranging from 32 to 128 bits and with keys ranging from 64 to 256 bits.

Public-key cryptography

Public-key cryptography splits the key into two pieces: a private key and a public key. The two are related such that a message encrypted with the private key can be decrypted using the public key, but the private key cannot be inferred from the public key. Because the public key does not disclose information about how the message is encoded, it can be kept in a public place for anyone to use. The Rivest—Shamir—Adleman (RSA) algorithm is one widely used public-key algorithm.

Hash functions

A **cryptographic hash function** has a somewhat different purpose. It is used to generate a **message digest** from a message. The message digest is generally shorter than the message itself and doesn't directly reveal the message contents. The hash function is designed to minimize *collisions* so that two different messages should be very unlikely to generate the same message digest. As a result, the hash function can be used to generate short versions of more lengthy messages. SHA-3 [Dwo15] is the latest in a series of SHA standards.

Digital signatures

We can use a combination of public-key cryptography and hash functions to create a **digital signature**, a message that authenticates a message coming from a particular sender. The sender uses her own private key to sign either the message itself or the message digest. The receiver of the message then uses the sender's public key to decrypt the signed message. A digital signature ensures the identity of the person who encrypts the message; the signature is unforgeable and unalterable. We can also combine digital signatures with message encryption. In this case, both the sender and receiver have private and public keys. The sender first signs the message with her private key, and then, encrypts the signed message with the *receiver's* public key. Upon receipt, the receiver first decrypts using her private key, and then, verifies the signature using the *sender's* public key.

Cryptographic functions may be implemented in software or hardware. The next example describes an embedded processor with hardware security accelerators.

Application Example 1.1: Texas Instruments TM4C129x Microcontroller

Texas Instruments TM4C [Tex14] is designed for applications that require floating-point computation and low-power performance, such as building automation, security and access control, and data acquisition. The CPU is an Arm Cortex-M4. It includes accelerators for AES, data encryption standard, SHA, MD5 (message digest), and cyclic redundancy check (CRC).

Attestation

Signed code allows the platform to know that the code comes from a trusted source. A computing platform often interacts with other platforms, either over networks or over other types of links. When Alice talks to Bill, Bill needs to know that Alice has not been compromised before sharing secure information with Alice. **Attestation** [Cok11] provides a mechanism for Alice to demonstrate integrity to Bill and for Bill to **appraise** Alice's security. For example, Alice may generate a hash of certain important, predefined pieces of code or portions of memory, sign the hash, and send it to Bill. Because those hashed elements are known to Bill, Bill expects a certain set of hash codes. If the hash codes differ, Bill should assume that Alice has been compromised. If the received codes match the expected codes, Alice can be judged to be secure with high confidence.

4.11 Design example: alarm clock

Our first system design example is an alarm clock. We use a microprocessor to read the clock's buttons and update the time display. Because we now better understand I/O, we work through the steps of the methodology to go from a concept to a completed and tested system.

4.11.1 Requirements

The basic functions of an alarm clock are well understood and easy to enumerate. Fig. 4.32 illustrates the front panel design for the alarm clock. The time is shown as four digits in 12-h format; we use light to distinguish between AM and PM. We use several buttons to set the clock and alarm times. When we press the *hour* and *minute* buttons, we advance the hour and minute each by one. When setting the time, we must hold down the *set time* button while we hit the *hour* and *minute* buttons; the

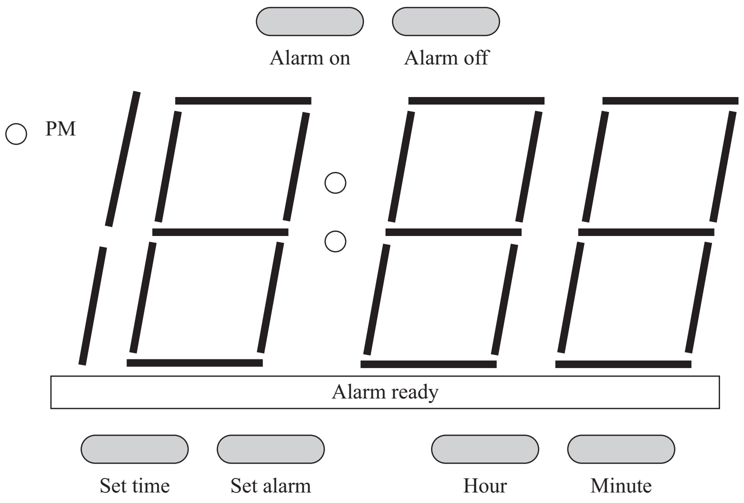


FIGURE 4.32

Front panel of the alarm clock.

set alarm button works in a similar fashion. We turn the alarm on and off with the *alarm on* and *alarm off* buttons. When the alarm is activated, the *alarm ready* light is on. A separate speaker provides the audible alarm.

We are now ready to create the following requirements table:

Name	**Alarm clock**
Purpose	A 24-h digital clock with a single alarm.
Inputs	Six pushbuttons: set time; set alarm, hour, minute, alarm on, alarm off.
Outputs	Four-digit, clock-style output. PM indicator light. Alarm ready light. Buzzer.
Functions	Default mode: The display shows the current time. PM light is on from noon to midnight. Hour and minute buttons are used to advance time and alarm, respectively. Pressing one of these buttons increments the hour/minute once. Depress set time button: This button is held down while hour/minute buttons are pressed to set time. New time is automatically shown on the display. Depress set alarm button: While this button is held down, display shifts to current alarm setting; depressing hour/minute buttons set alarm value in a manner like setting time. Alarm on: Puts clock in alarm-on state, causes clock to turn on buzzer when current time reaches alarm time, turns on alarm ready light. Alarm off: Turns off buzzer, takes clock out of alarm-on state, turns off alarm ready light.

Continued

—Continued

Name	Alarm clock
Performance	Displays hours and minutes, but not seconds. Should be accurate within the accuracy of a typical microprocessor clock signal. (Excessive accuracy may unreasonably drive up the cost of generating an accurate clock.)
Manufacturing cost	Consumer product range. Cost will be dominated by the microprocessor system, not the buttons or displays.
Power	Powered by AC through a standard power supply.
Physical size and weight	Small enough to fit on a nightstand with expected weight for an alarm clock.

4.11.2 Specification

The basic function of the clock is simple, but we need to create some classes and associated behaviors to clarify exactly how the user interface works.

Fig. 4.33 shows the basic classes for the alarm clock. Borrowing a term from mechanical watches, we call the class that handles the basic clock operation the *Mechanism* class. We have three classes that represent physical elements: *Lights** for all the digits and lights, *Buttons** for all the buttons, and *Speaker** for the sound output. The *Buttons** class can easily be used directly by *Mechanism*. As discussed below, the physical display must be scanned to generate the digit output, so we introduce the *Display* class to abstract the physical lights.

The details of the low-level user interface classes are shown in Fig. 4.34. The *Buttons** class provides read-only access to the current state of the buttons. The *Lights** class allows us to drive the lights. However, to save pins on the display, *Lights** provides signals for only one digit, along with a set of signals to indicate which digit is currently being addressed. We generate the display by scanning the digits periodically. That function is performed by the *Display* class, which makes the display appear as an unscanned, continuous display to the rest of the system.

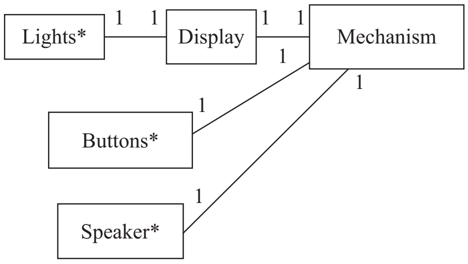


FIGURE 4.33

Class diagram for the alarm clock.

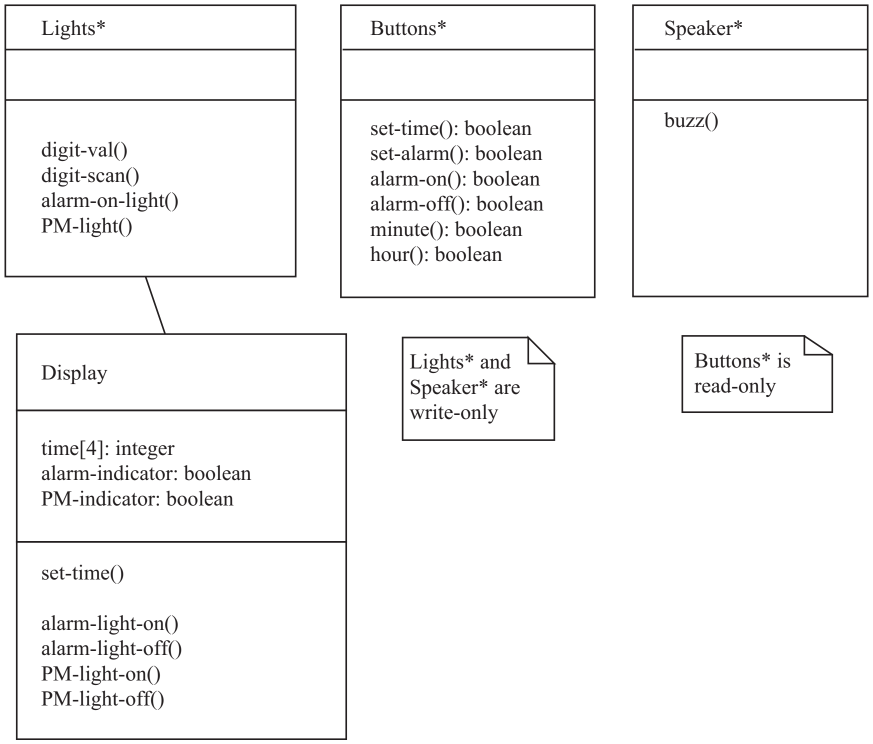


FIGURE 4.34

Details of user interface classes for the alarm clock.

The *Mechanism* class is presented in Fig. 4.35. This class keeps track of the current time, the current alarm time, whether the alarm has been turned on, and whether it is currently buzzing. The clock shows the time only to the minute, but it keeps the internal time to the second. The time is kept as discrete digits rather than as a single integer to simplify transferring the time to the display. The class provides two behaviors, both of which run continuously. First, the *scan-keyboard* handles looking at the inputs and updating the alarm and other functions as requested by the user. Second, *update-time* keeps the current time accurate.

Fig. 4.36 shows the state diagram for *update-time*. This behavior is straightforward, but it must do several things. It is activated once per second and must update the second clock. If it has counted 60 seconds, it must then update the displayed time; when it does so, it must roll over between digits and keep track of AM-to-PM and PM-to-AM transitions. It sends the updated time to the display object. It also compares the time with the alarm setting and sets the alarm buzzing under the proper conditions.

The state diagram for the *scan-keyboard* is shown in Fig. 4.37. This function is called periodically, frequently enough, so that all the user's button presses are

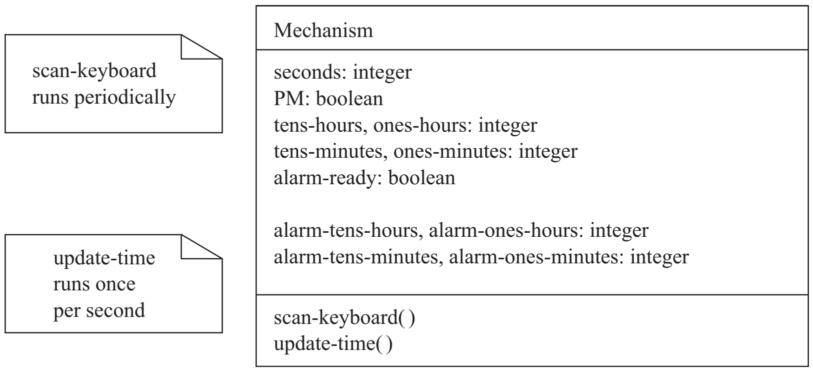

scan-keyboard runs periodically
update-time runs once per second
Mechanism
seconds: integer
PM: boolean
tens-hours, ones-hours: integer
tens-minutes, ones-minutes: integer
alarm-ready: boolean
alarm-tens-hours, alarm-ones-hours: integer
alarm-tens-minutes, alarm-ones-minutes: integer
scan-keyboard( )
update-time( )

FIGURE 4.35

The mechanism class.

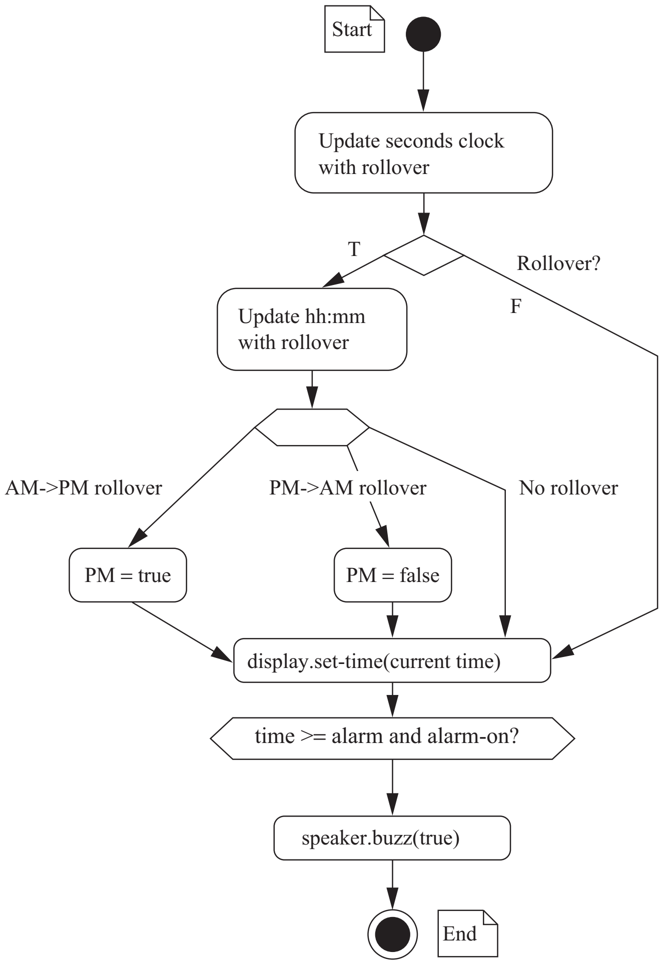

Start
Update seconds clock with rollover
T
Rollover?
F
Update hh:mm with rollover
AM->PM rollover
PM->AM rollover
No rollover
PM = true
PM = false
display.set-time(current time)
time >= alarm and alarm-on?
speaker.buzz(true)
End

FIGURE 4.36

State diagram for update-time.

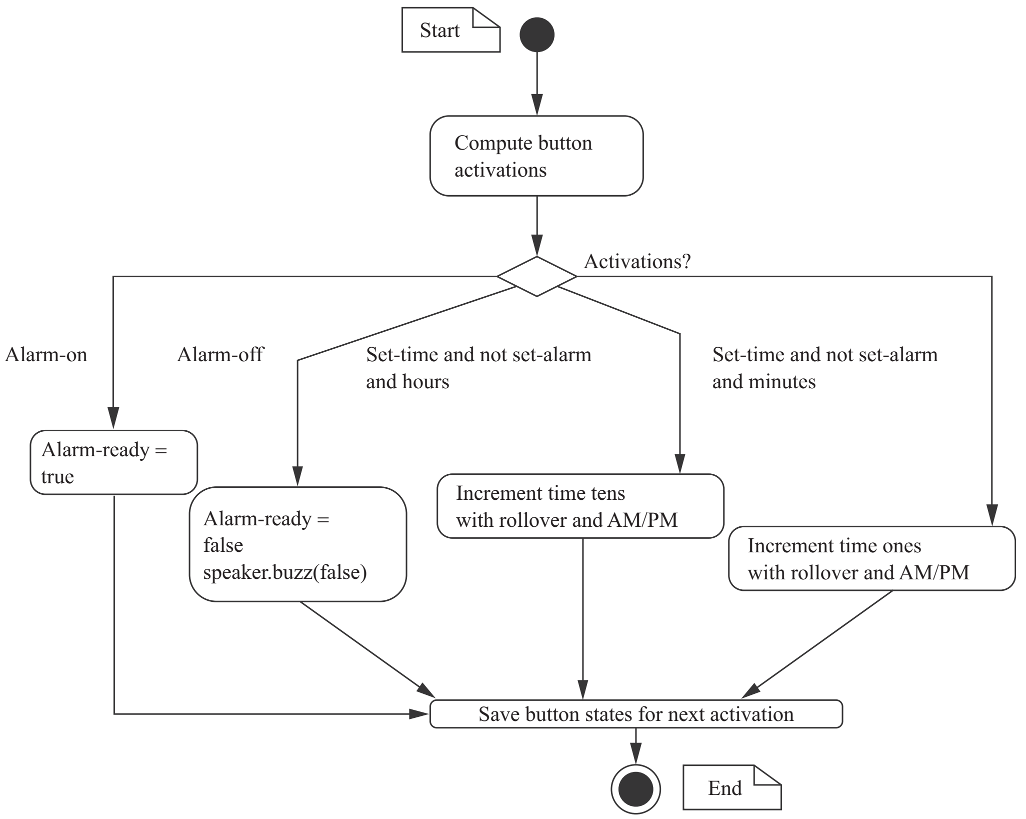

Start
Compute button activations
Activations?
Alarm-on
Alarm-off
Set-time and not set-alarm and hours
Set-time and not set-alarm and minutes
Alarm-ready = true
Alarm-ready = false speaker.buzz(false)
Increment time tens with rollover and AM/PM
Increment time ones with rollover and AM/PM
Save button states for next activation
End

FIGURE 4.37

State diagram for scan-keyboard.

caught by the system. Because the keyboard will be scanned several times per second, we don't want to register the same button press several times. If, for example, we advanced the minutes count on every keyboard scan when the *set time* and *minutes* buttons were pressed, the time would be advanced much too fast. To make the buttons respond more reasonably, the function computes button activations; it compares the current state of the button to the button's value on the last scan, and it considers the button activated only when it is on for this scan but was off for the last scan. Once computing the activation values for all the buttons, it looks at the activation combinations and takes the appropriate actions. Before exiting, it saves the current button values for computing activations the next time this behavior is executed.

4.11.3 System architecture

The software and hardware architectures of a system are always hard to completely separate, but let's first consider the software architecture, and then, its implications for the hardware.

The system has both periodic and aperiodic components; the current time must obviously be updated periodically, and the button commands occur occasionally.

It seems reasonable to have the following two major software components:

- An interrupt-driven routine can update the current time. The current time will be kept in a variable in the memory. A timer can be used to interrupt periodically and update the time. As seen in the subsequent discussion of the hardware architecture, the display must send the new value when the minute value changes. This routine can also maintain the PM indicator.
- A foreground program can poll the buttons and execute their commands. Because buttons are changed at a relatively slow rate, it makes no sense to add the hardware required to connect the buttons to interrupts. Instead, the foreground program will read the button values, and then, use simple conditional tests to implement the commands, including setting the current time, setting the alarm, and turning off the alarm. Another routine called by the foreground program will turn the buzzer on and off based on the alarm time.

An important question for the interrupt-driven current time handler is how often the timer interrupts occur. A one-minute interval would be very convenient for the software, but a one-minute timer would require many counter bits. It is more realistic to use a 1-s timer and a program variable to count the seconds in a minute.

The foreground code will be implemented as a while `loop`.

```
while (TRUE) {
    read_buttons(button_values); /*read inputs */
    process_command(button_values); /*do commands */
    check_alarm(); /*decide whether to turn on the alarm */
}
```

The loop first reads the buttons using `read_buttons()`. In addition to reading the current button values from the input device, this routine must preprocess the button values so that the user interface code responds properly. The buttons will remain depressed for many sample periods, because the sample rate is much faster than any person can push and release buttons. We want to make sure that the clock responds to this as a single depression of the button, not as one depression per sample interval. As shown in Fig. 4.38, this can be done by performing a simple edge detection on the button input; the button event value is one for one sample period when the button is depressed, and then, goes back to zero and does not return to one until the button is depressed, and then, released. This can be accomplished using a simple two-state machine.

The `process_command()` function handles responding to button events. The `check_alarm()` function checks the current time against the alarm time and decides

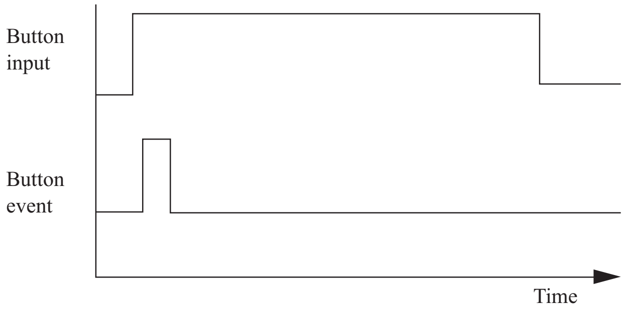


FIGURE 4.38

Preprocessing button inputs.

when to turn on the buzzer. This routine is kept separate from the command processing code because the alarm must go on when the proper time is reached, independent of the button inputs.

We have determined from the software architecture that we will need a timer connected to the CPU. We will also need logic to connect the buttons to the CPU bus. In addition to performing edge detection on the button inputs, we must, of course, debounce the buttons.

The final step before starting to write code and build hardware is to draw the state transition graph for the clock's commands. This diagram will be used to guide the implementation of the software components.

4.11.4 Component design and testing

The two major software components, the interrupt handler and the foreground code, can be implemented relatively straightforwardly. Because most of the functionality of the interrupt handler is in the interruption process itself, that code is best tested on the microprocessor platform. The foreground code can be more easily tested on the PC or workstation used for code development. We can create a testbench for this code that generates button depressions to exercise the state machine. We will also need to simulate the advancement of the system clock. Trying to directly execute the interrupt handler to control the clock is probably a bad idea; not only would that require some type of emulation of interrupts, but it would also require us to count interrupts second by second. A better testing strategy is to add testing code that updates the clock, perhaps once per four iterations of the foreground `while` loop.

The timer will probably be a stock component, so we would then focus on implementing logic to interface with the buttons, display, and buzzer. The buttons will require debouncing logic. The display will require a register to hold the current display value to drive the display elements.

4.11.5 System integration and testing

Because this system has a small number of components, system integration is relatively easy. The software must be checked to ensure that the debugging code has been turned off. Three types of tests can be performed. First, the clock's accuracy can be checked against a reference clock. Second, commands can be exercised from the buttons. Finally, the buzzer's functionality should be verified.

4.12 Design example: jet engine controller

A jet engine requires real-time controls to ensure that it responds properly to the pilot's actions. Balancing the computations required for control with sensing and actuation using the I/O devices gives us a good example of performance analysis in bus-based systems.

4.12.1 Theory of operation and requirements

A jet engine's operation is remarkably simple [Rol15]. As shown in Fig. 4.39, air enters the engine through a fan, which pushes the air through a compressor. The highly compressed air flows into a combustion chamber, where fuel is applied. The burning exhaust flows out the back of the engine through a turbine. The turbine is turned by the expelled air. Shafts couple that rotation back to the compressor and fan to keep the engine fed with air. The fan also provides airflow that bypasses the combustion chamber; this airflow provides additional thrust. The compressor runs continuously.

The digital controller for a jet engine is known as **full authority digital electronic control** (**FADEC**). A traditional jet engine has one control: the throttle [Gar, Liu12]. Engine thrust cannot be directly sensed, but can be inferred. Thrust is a function of shaft speed that can be directly measured. Sensors can also be installed to measure variables related to the health of the engine, such as pressure and temperature.

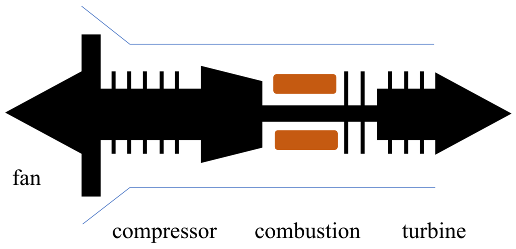


FIGURE 4.39

Cross-section of a jet engine.

4.12.2 Specifications

The pilot's command input to the engine is the throttle. The throttle position is known as the **throttle resolver angle** (**TRA**). A typical engine uses a pair of shafts, one for low-pressure (N1) and another for high-pressure (N2) sections. Sensors can directly measure the rotational speed of these shafts. These measurements allow for feedback control of the engine; the shaft speeds, and therefore, the thrust can be compared to the commanded thrust.

We will concentrate on a very simple control algorithm for the jet engine shown in Fig. 4.40:

- The throttle resolver angle is the pilot's command input to the jet engine.
- Fuel flow (WF) is computed based on TRA and the current computed thrust; that WF is sent to the fuel system.
- The shaft speeds N1 and N2 are sensed.
- Thrust is computed from the shaft speeds. The results are stored for the next control cycle.

A typical sample rate for the engine controller is in the $10 - 100\ Hz$ range.

4.12.3 System architecture

As shown in Fig. 4.41, the computing platform for a FADEC typically consists of a CPU and a separate I/O bus such as the controller area network (CAN) bus that

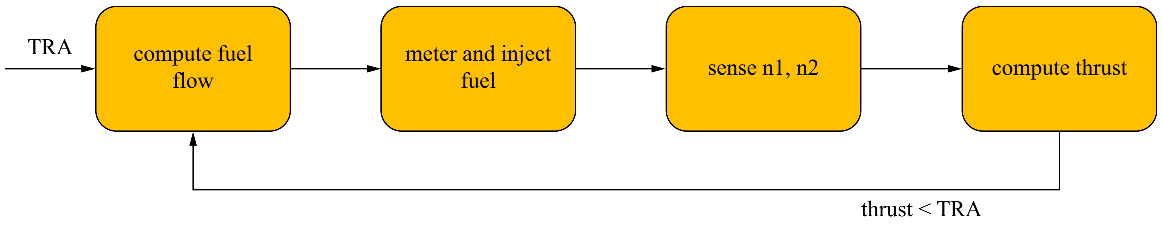


FIGURE 4.40

Jet engine control algorithm.

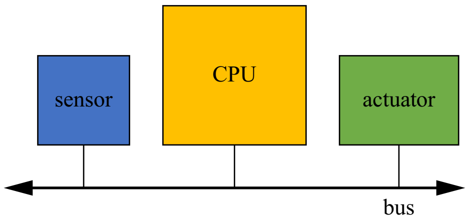


FIGURE 4.41

Simple computing platform for jet engine control.

will be discussed in Chapter 10. The CPU bus is not used directly to connect to devices; the CPU bus is connected to the CAN bus, which in turn connects to the sensors and actuators. However, the scheduling principles for this architecture are the same as those for a CPU bus.

As shown in Fig. 4.42, the control computation is divided into two tasks: computing WF from the TRA and computing thrust from shaft speeds. The WF computation cannot begin until after the TRA has been received. However, the bus can read the shaft speeds N1 and N2, while the CPU performs that computation. Similarly, the bus can send WF to the fuel system, while the thrust is computed from the shaft speeds.

4.12.4 Component design

Control algorithms can be evaluated using a model of the jet engine. C-MAPSS [Liu12], for example, is a Simulink model of a jet engine. FADEC software will often be coded in C language. The numerical behavior of the control algorithms is important for proper operation. The required numerical precision can be evaluated using jet engine models, and the execution time required for various number formats can be determined by software performance analysis.

4.12.5 System integration and testing

A test harness could be built to test the software on the target computing platform. The harness would provide sensor data and log actuator data. The results could be evaluated after each run to assess the software's correctness.

Jet engine test stands must be carefully designed. A test stand, for example, may allow the engine to operate remotely. The test bay can be separated from the interior of the building by a wall and a shatter-resistant observation window. A large set of louvers fill most of the outside wall of the test bay; in case of an explosion, the louvers blow open and vent to the outside.

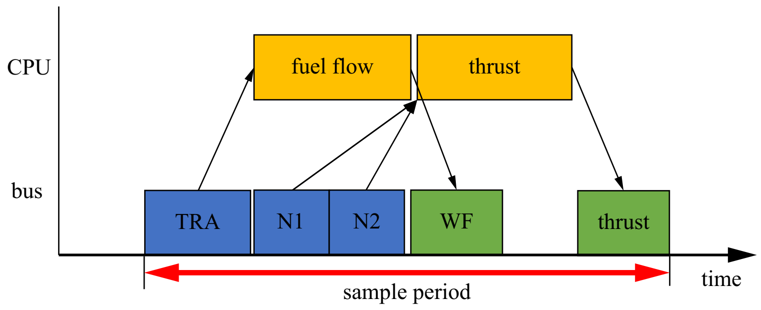


FIGURE 4.42

Schedule for I/O and computation in a jet engine controller.

4.13 Summary

The microprocessor is only one component in an embedded computing system; memory and I/O devices are equally important. The microprocessor bus serves as the glue that binds all these components together. Hardware platforms for embedded systems are often built around common platforms with appropriate amounts of memory and I/O devices added on; low-level monitor software also plays an important role in these systems.

What we learned

- CPU buses are built on handshaking protocols.
- A variety of memory components are available, which vary widely in speed, capacity, and other capabilities.
- An I/O device uses logic to interface to the bus so that the CPU can read and write the device's registers.
- Embedded systems can be debugged using a variety of hardware and software methods.
- System-level performance depends not just on the CPU, but the memory and bus as well.
- Security features of the platform enable the development of secure applications.

Further reading

Shanley and Anderson [Min95] describe the PCI bus in detail. Dahlin [Dah00] describes how to interface to a touchscreen. Collins [Col97] describes the design of microprocessor ICEs. Earnshaw et al. [Ear97] describe an advanced debugging environment for the Arm architecture.

Questions

Q4-1 Name three major hardware components of a generic computing platform.

Q4-2 Name three major software components of a generic computing platform.

Q4-2 What role does the HAL play in the platform?

Q4-4 Draw UML state diagrams for device 1 and device 2 in a four-cycle handshake.

Q4-5 Describe the role of these signals in a bus:
- **a.** R/W'
- **b.** data-ready
- **c.** clock

Q4-6 Draw a UML sequence diagram that shows a four-cycle handshake between a bus controller and a device.

Q4-7 Define these signal types in a timing diagram:
- **a.** changing
- **b.** stable

Q4-8 Draw a timing diagram with the following signals (where $[t_1,t_2]$ is the time interval, starting at t_1 and ending at t_2).
- **a.** Signal A is stable [0,10], changing [10,15], stable [15,30].
- **b.** Signal B is 1 [0,5], falling [5,7], 0 [7,20], changing [20,30].
- **c.** Signal C is changing [0,10], 0 [10,15], rising [15,18], 1 [18,25], changing [25,30].

Q4-9 Draw a timing diagram for a write operation with no wait states.

Q4-10 Draw a timing diagram for a read operation on a bus in which the read includes two wait states.

Q4-11 Draw a timing diagram for a write operation on a bus in which the write takes two wait states.

Q4-12 Draw a timing diagram for a burst write operation that writes four locations.

Q4-13 Draw a UML state diagram for a burst read operation with wait states. One state diagram is for the bus controller and the other is for the device being read.

Q4-14 Draw a UML sequence diagram for a burst read operation with wait states.

Q4-15 Draw timing diagrams for
- **a.** a device becoming bus controller; and
- **b.** the device returning control of the bus to the CPU.

Q4-16 Draw a timing diagram that shows a complete DMA operation, including handing off the bus to the DMA controller, performing the DMA transfer, and returning bus control back to the CPU.

Q4-17 Draw UML state diagrams for a bus controllership transaction, in which one side shows the CPU as the default bus controller and the other shows the device that can request bus controllership.

Q4-18 Draw a UML sequence diagram for a bus controllership request, grant, and return.

Q4-19 Draw a UML sequence diagram that shows a DMA bus transaction and concurrent processing on the CPU.

Q4-20 Draw a UML sequence diagram for a complete DMA transaction, including the DMA controller requesting the bus, the DMA transaction itself, and returning control of the bus to the CPU.

Q4-21 Draw a UML sequence diagram showing a read operation across a bus bridge.

Q4-22 Draw a UML sequence diagram showing a write operation with wait states across a bus bridge.

Q4-23 Draw a UML sequence diagram for a read transaction that includes a DRAM refresh operation. The sequence diagram should include the CPU, the DRAM interface, and the DRAM internals to show the refresh itself.

Q4-24 Draw a UML sequence diagram for a DRAM read operation. Show the activity of each of the DRAM signals.

Q4-25 What is the role of a memory controller in a computing platform?

Q4-26 What hardware factors might be considered when choosing a computing platform?

Q4-27 What software factors might be considered when choosing a computing platform?

Q4-28 Write ARM assembly language code that handles a breakpoint. It should save the necessary registers, call a subroutine to communicate with the host, and upon return from the host, cause the breakpointed instruction to be properly executed.

Q4-29 Assume an A/D converter is supplying samples at 44.1 kHz.

a. How much time is available per sample for CPU operations?

b. If the interrupt handler executes 100 instructions obtaining the sample and passing it onto the application routine, how many instructions can be executed on a 20-MHz RISC processor that executes 1 instruction per cycle?

Q4-30 If an interrupt handler executes for too long and the next interrupt occurs before the last call to the handler has finished, what happens?

Q4-31 Consider a system in which an interrupt handler passes on samples to a finite impulse response (FIR) filter program that runs in the background.

a. If the interrupt handler takes too long, how does the FIR filter's output change?

a. If the FIR filter code takes too long, how does its output change?

Q4-32 Assume that your microprocessor implements an ICE instruction that asserts a bus signal that causes a microprocessor ICE to start. Additionally, assume that the microprocessor allows all internal registers to be observed and controlled through a boundary scan chain. Draw a UML sequence diagram of the ICE operation, including execution of the ICE instruction, uploading the microprocessor state to the ICE, and returning control to the microprocessor's program. The sequence diagram should include the microprocessor, the microprocessor ICE, and the user.

Q4-33 Why might an embedded computing system want to implement a DOS-compatible file system?

Q4-34 Name two example embedded systems that implement a DOS-compatible file system.

Q4-35 You are given a memory system with an overhead O=2 and a single-word transfer time of 1 (no wait states.) You will use this memory system to perform a transfer of 1,024 locations. Plot total number of clock cycles T as a function of burst size B for $1 <= B <= 8$.

Q4-36 You are given a bus that supports single-word and burst transfers. A single transfer takes one clock cycle (no wait states). The overhead of the single-word transfer is 1 clock cycles (O = 1). The overhead of a burst is 3 clock cycles ($O_B = 3$) Which performs a two-word transfer faster: a pair of single transfers or a single burst of two words?

Q4-37 You are given a 2-byte wide bus that supports single-byte, dual word (same clock cycle) and burst transfers of up to 8 bytes (four byte pairs per burst). The overhead of each of these types of transfers is 1 clock cycle ($O = O_B = 1$) and a data transfer takes one clock cycle per single or dual word (D=1). You want to send a 1080P video frame at a resolution of 1920×1080 pixels with 3 bytes per pixel. Compare the difference in bus transfer times if the pixels are packed vs. sending a pixel as a 2-byte followed by a single-byte transfer.

Q4-38 Determine the design parameters for an audio system:

a. Determine the total bytes/s required for an audio signal of 16 bits/sample per channel, two channels, sampled at 44.1 kHz.

b. Given a clock period P = 20 MHz for a bus, determine the bus width required assuming that nonburst mode transfers are used and D = O = 1.

c. Given a clock period P = 20 MHz for a bus, determine the bus width required assuming burst transfers of length four and $D = O_B = 1$.

d. Assume the data signal now contains both the original audio signal and a compressed version of the audio at a bit rate of 1/10 the input audio signal. Assume bus bandwidth for burst transfers of length four with P = 20 MHz and $D = O_B = 1$. Will a bus of width 1 be sufficient to handle the combined traffic?

Q4-39 You are designing a system a bus-based computer: the input device I1 sends its data to program P1; P1 sends its output to device O1. Is there any way to overlap bus transfers and computations in this system?

Q4-40 What hardware modules might be used to create a digital signature?

Lab exercises

L4-1 Use a logic analyzer to view system activity on your bus.

L4-2 If your logic analyzer is capable of on-the-fly disassembly, use it to display bus activity in the form of instructions, rather than simply ones and zeroes.

L4-3 Attach LEDs to your peripheral bus so that you can monitor its activity. For example, use an LED to monitor the read/write line on the bus.

L4-4 Design logic to interface an I/O device to your microprocessor.

L4-5 Have someone else deliberately introduce a bug into one of your programs, and then use the appropriate debugging tools to find and correct the bug.

L4-6 Identify the different bus transaction types in your platform. Compute the best-case bus bandwidth.

L4-7 Construct a simple program to access memory in widely separated places. Measure the memory system bandwidth and compare to the best-case bandwidth.

L4-8 Construct a simple program to perform some memory accesses. Use a logic analyzer to study the bus activity. Determine what types of bus modes are used for the transfers.