10 Input and Output

Because cyber-physical systems integrate computing and physical dynamics, the mechanisms in processors that support interaction with the outside world are central to any design. A system designer has to confront a number of issues. Among these, the mechanical and electrical properties of the interfaces are important. Incorrect use of parts, such as drawing too much current from a pin, may cause a system to malfunction or may reduce its useful lifetime. In addition, in the physical world, many things happen at once.

Software, by contrast, is mostly sequential. Reconciling these two disparate properties is a major challenge, and is often the biggest risk factor in the design of embedded systems. Incorrect interactions between sequential code and concurrent events in the physical world can cause dramatic system failures. In this chapter, we deal with issues.

10.1 I/O Hardware

Embedded processors, be they microcontrollers, DSP processors, or general-purpose processors, typically include a number of input and output (**I/O**) mechanisms on chip, exposed to designers as pins of the chip. In this section, we review some of the more common interfaces provided, illustrating their properties through the following running example.

Example 10.1: Figure 10.1 shows an evaluation board for the Luminary Micro Stellaris® microcontroller, which is an ARM Cortex™ - M3 32-bit processor. The microcontroller itself is in the center below the graphics display. Many of the pins of the microcontroller are available at the connectors shown on either side of the microcontroller and at the top and bottom of the board. Such a board would typically be used to prototype an embedded application, and in the final product it would be replaced with a custom circuit board that includes only the hardware required by the application. An engineer will develop software for the board using an integrated development environment (**IDE**) provided by the vendor and load the software onto flash memory to be inserted into the slot at the bottom of the board. Alternatively, software might be loaded onto the board through the USB interface at the top from the development computer.

The evaluation board in the above example is more than a processor since it includes a display and various hardware interfaces (switches and a speaker, for example). Such a board is often called a **single-board computer** or a **microcomputer board**. We next discuss a few of the interfaces provided by a microcontroller or single-board computer. For a more comprehensive description of the many kinds of I/O interfaces in use, we recommend Valvano (2007) and Derenzo (2003).

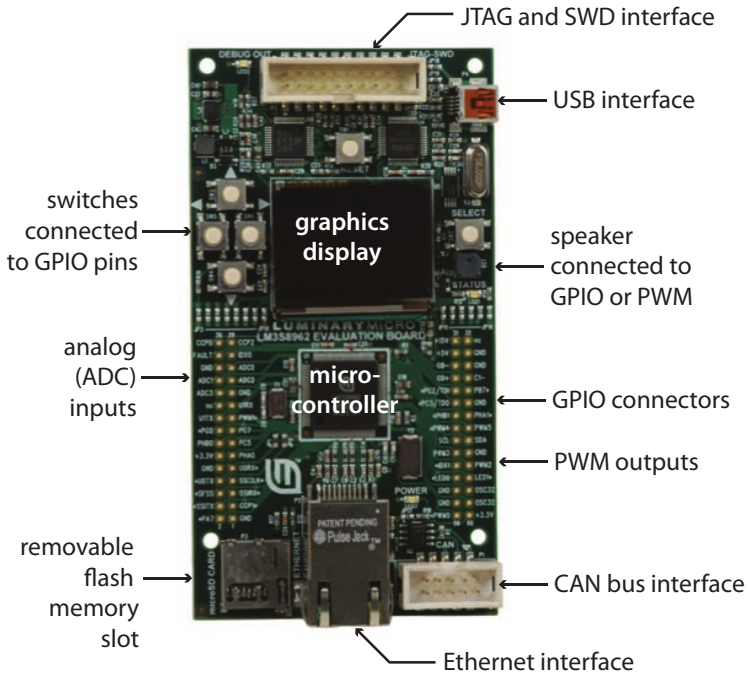


Figure 10.1: Stellaris® LM3S8962 evaluation board (Luminary Micro®, 2008a). (Luminary Micro was acquired by pointerTexas Instruments in 2009.)

10.1.1 Pulse Width Modulation

Pulse width modulation (PWM) is a technique for delivering a variable amount of power efficiently to external hardware devices. It can be used to control for example the speed of electric motors, the brightness of an LED light, and the temperature of a heating element. In general, it can deliver varying amounts of power to devices that tolerate rapid and abrupt changes in voltage and current.

PWM hardware uses only digital circuits, and hence is easy to integrate on the same chip with a microcontroller. Digital circuits, by design, produce only two voltage levels, high and low. A PWM signal rapidly switches between high and low at some fixed frequency, varying the amount of time that it holds the signal high. The duty cycle is the proportion

of time that the voltage is high. If the duty cycle is 100%, then the voltage is always high. If the duty cycle is 0%, then the voltage is always low.

Many microcontrollers provide PWM peripheral devices (see Figure 10.1). To use these, a programmer typically writes a value to a memory-mapped register to set the duty cycle (the frequency may also be settable). The device then delivers power to external hardware in proportion to the specified duty cycle.

PWM is an effective way to deliver varying amounts of power, but only to certain devices. A heating element, for example, is a resistor whose temperature increases as more current passes through it. Temperature varies slowly, compared to the frequency of a PWM signal, so the rapidly varying voltage of the signal is averaged out by the resistor, and the temperature will be very close to constant for a fixed duty cycle. Motors similarly average out rapid variations in input voltage. So do incandescent and LED lights. Any device whose response to changes in current or voltage is slow compared to the frequency of the PWM signal is a candidate for being controlled via PWM.

10.1.2 General-Purpose Digital I/O

Embedded system designers frequently need to connect specialized or custom digital hardware to embedded processors. Many embedded processors have a number of **general-purpose I/O** pins (**GPIO**), which enable the software to either read or write voltage levels representing a logical zero or one. If the processor **supply voltage** is V_{DD}, in **active high logic** a voltage close to V_{DD} represents a logical one, and a voltage near zero represents a logical zero. In **active low logic**, these interpretations are reversed.

In many designs, a GPIO pin may be configured to be an output. This enables software to then write to a memory-mapped register to set the output voltage to be either high or low. By this mechanism, software can directly control external physical devices.

However, caution is in order. When interfacing hardware to GPIO pins, a designer needs to understand the specifications of the device. In particular, the voltage and current levels vary by device. If a GPIO pin produces an output voltage of V_{DD} when given a logical one, then the designer needs to know the current limitations before connecting a device to it. If a device with a resistance of R ohms is connected to it, for example, then Ohm's law tells us that the output current will be

$$I = V_{DD}/R\,.$$

It is essential to keep this current within specified tolerances. Going outside these tolerances could cause the device to overheat and fail. A **power amplifier** may be needed to deliver adequate current. An amplifier may also be needed to change voltage levels.

Example 10.2: The GPIO pins of the Luminary Micro Stellaris® microcontroller shown in Figure 10.1 may be configured to source or sink varying amounts of current up to 18 mA. There are restrictions on what combinations of pins can handle such relatively high currents. For example, Luminary Micro® (2008b) states "The high-current GPIO package pins must be selected such that there are only a maximum of two per side of the physical package ... with the total number of high-current GPIO outputs not exceeding four for the entire package." Such constraints are designed to prevent overheating of the device.

In addition, it may be important to maintain **electrical isolation** between processor circuits and external devices. The external devices may have messy (noisy) electrical characteristics that will make the processor unreliable if the noise spills over into the power or ground lines of the processor. Or the external device may operate in a very different voltage or power regime compared to the processor. A useful strategy is to divide a circuit into **electrical domains**, possibly with separate power supplies, that have relatively little influence on one another. Isolation devices such as opto-isolators and transformers may be used to enable communication across electrical domains. The former convert an electrical signal in one electrical domain into light, and detect the light in the other electrical domain and convert it back to an electrical signal. The latter use inductive coupling between electrical domains.

GPIO pins can also be configured as inputs, in which case software will be able to react to externally provided voltage levels. An input pin may be **Schmitt triggered**, in which case they have hysteresis, similar to the thermostat of Example 3.5. A Schmitt triggered input pin is less vulnerable to noise. It is named after Otto H. Schmitt, who invented it in 1934 while he was a graduate student studying the neural impulse propagation in squid nerves.

Example 10.3: The GPIO pins of the microcontroller shown in Figure 10.1, when configured as inputs, are Schmitt triggered.

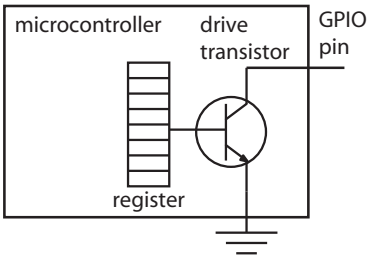


Figure 10.2: An open collector circuit for a GPIO pin.

In many applications, several devices may share a single electrical connection. The designer must take care to ensure that these devices do not simultaneously drive the voltage of this single electrical connection to different values, resulting in a short circuit that can cause overheating and device failure.

Example 10.4: Consider a factory floor where several independent microcontrollers are all able to turn off a piece of machinery by asserting a logical zero on an output GPIO line. Such a design may provide additional safety because the microcontrollers may be redundant, so that failure of one does not prevent a safety-related shutdown from occurring. If all of these GPIO lines are wired together to a single control input of the piece of machinery, then we have to take precautions to ensure that the microcontrollers do not short each other out. This would occur if one microcontroller attempts to drive the shared line to a high voltage while another attempts to drive the same line to a low voltage.

GPIO outputs may use **open collector** circuits, as shown in Figure 10.2. In such a circuit, writing a logical one into the (memory mapped) register turns on the transistor, which pulls the voltage on the output pin down to (near) zero. Writing a logical zero into the register turns off the transistor, which leaves the output pin unconnected, or "open."

A number of open collector interfaces may be connected as shown in Figure 10.3. The shared line is connected to a **pull-up resistor**, which brings the voltage of the line up to V_{DD} when all the transistors are turned off. If any one transistor is turned on, then it will bring the voltage of the entire line down to (near) zero without creating a short circuit with

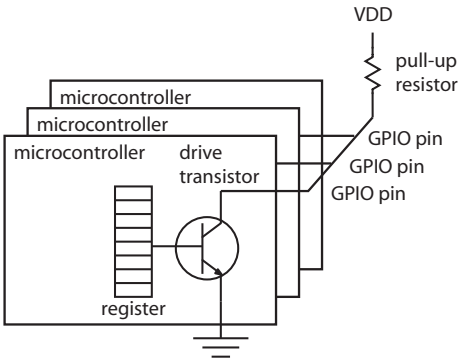


Figure 10.3: A number of open collector circuits wired together.

the other GPIO pins. Logically, all registers must have zeros in them for the output to be high. If any one of the registers has a one in it, then the output will be low. Assuming active high logic, the logical function being performed is NOR, so such a circuit is called a **wired NOR**. By varying the configuration, one can similarly create wired OR or wired AND.

The term "open collector" comes from the name for the terminal of a bipolar transistor. In CMOS technologies, this type of interface will typically be called an **open drain** interface. It functions essentially in the same way.

Example 10.5: The GPIO pins of the microcontroller shown in Figure 10.1, when configured as outputs, may be specified to be open drain circuits. They may also optionally provide the pull-up resistor, which conveniently reduces the number of external discrete components required on a printed circuit board.

GPIO outputs may also be realized with **tristate** logic, which means that in addition to producing an output high or low voltage, the pin may be simply turned off. Like an open-collector interface, this can facilitate sharing the same external circuits among multiple devices. Unlike an open-collector interface, a tristate design can assert both high and low voltages, rather than just one of the two.

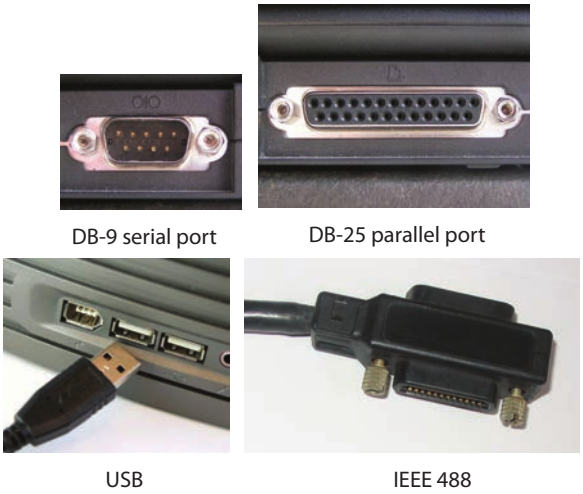


Figure 10.4: Connectors for serial and parallel interfaces.

10.1.3 Serial Interfaces

One of the key constraints faced by embedded processor designers is the need to have physically small packages and low power consumption. A consequence is that the number of pins on the processor integrated circuit is limited. Thus, each pin must be used efficiently. In addition, when wiring together subsystems, the number of wires needs to be limited to keep the overall bulk and cost of the product in check. Hence, wires must also be used efficiently. One way to use pins and wires efficiently is to send information over them serially as sequences of bits. Such an interface is called a **serial interface**. A number of standards have evolved for serial interfaces so that devices from different manufacturers can (usually) be connected.

An old but persistent standard, **RS-232**, standardized by the Electronics Industries Association (EIA), was first introduced in 1962 to connect teletypes to modems. This standard defines electrical signals and connector types; it persists because of its simplicity and because of continued prevalence of aging industrial equipment that uses it. The standard defines how one device can transmit a byte to another device asynchronously (meaning that the devices do not share a clock signal). On older PCs, an RS-232 connection may be provided via a DB-9 connector, as shown in Figure 10.4. A microcontroller will typically

use a **universal asynchronous receiver/transmitter** (**UART**) to convert the contents of an 8-bit register into a sequence of bits for transmission over an RS-232 serial link.

For an embedded system designer, a major issue to consider is that RS-232 interfaces can be quite slow and may slow down the application software, if the programmer is not very careful.

Example 10.6: All variants of the Atmel AVR microcontroller include a UART that can be used to provide an RS-232 serial interface. To send a byte over the serial port, an application program may include the lines

```
1 while(!(UCSR0A & 0x20));
2 UDR0 = x;
```

where `x` is a variable of type `uint8_t` (a C data type specifying an 8-bit unsigned integer). The symbols `UCSR0A` and `UDR0` are defined in header files provided in the AVR IDE. They are defined to refer to memory locations corresponding to memory-mapped registers in the AVR architecture.

The first line above executes an empty `while` loop until the serial transmit buffer is empty. The AVR architecture indicates that the transmit buffer is empty by setting the sixth bit of the memory mapped register `UCSR0A` to 1. When that bit becomes 1, the expression `!(UCSR0A & 0x20)` becomes 0 and the `while` loop stops looping. The second line loads the value to be sent, which is whatever the variable `x` contains, into the memory-mapped register `UDR0`.

Suppose you wish to send a sequence of 8 bytes stored in an array `x`. You could do this with the C code

```
1 for(i = 0; i < 8; i++) {
2   while(!(UCSR0A & 0x20));
3   UDR0 = x[i];
4 }
```

How long would it take to execute this code? Suppose that the serial port is set to operate at 57600 baud, or bits per second (this is quite fast for an RS-232 interface). Then after loading `UDR0` with an 8-bit value, it will take $8/57600$ seconds or about 139 microseconds for the 8-bit value to be sent. Suppose that the frequency of the processor is operating at 18 MHz (relatively slow for a microcontroller).

Then except for the first time through the `for` loop, each `while` loop will need to consume approximately 2500 cycles, during which time the processor is doing no useful work.

To receive a byte over the serial port, a programmer may use the following C code:

```
1 while(!(UCSR0A & 0x80));
2 return UDR0;
```

In this case, the `while` loop waits until the UART has received an incoming byte. The programmer must ensure that there will be an incoming byte, or this code will execute forever. If this code is again enclosed in a loop to receive a sequence of bytes, then the `while` loop will need to consume a considerable number of cycles each time it executes.

For both sending and receiving bytes over a serial port, a programmer may use an interrupt instead to avoid having an idle processor that is waiting for the serial communication to occur. Interrupts will be discussed below.

The RS-232 mechanism is very simple. The sender and receiver first must agree on a transmission rate (which is slow by modern standards). The sender initiates transmission of a byte with a **start bit**, which alerts the receiver that a byte is coming. The sender then clocks out the sequence of bits at the agreed-upon rate, following them by one or two **stop bit**s. The receiver's clock resets upon receiving the start bit and is expected to track the sender's clock closely enough to be able to sample the incoming signal sequentially and recover the sequence of bits. There are many descendants of the standard that support higher rate communication, such as **RS-422**, **RS-423**, and more.

Newer devices designed to connect to personal computers typically use **universal serial bus** (**USB**) interfaces, standardized by a consortium of vendors. USB 1.0 appeared in 1996 and supports a data rate of 12 Mbits/sec. USB 2.0 appeared in 2000 and supports data rates up to 480 Mbits/sec. USB 3.0 appeared in 2008 and supports data rates up to 4.8 Gbits/sec.

USB is electrically simpler than RS-232 and uses simpler, more robust connectors, as shown in Figure 10.4. But the USB standard defines much more than electrical transport of bytes, and more complicated control logic is required to support it. Since modern peripheral devices such as printers, disk drives, and audio and video devices all include

microcontrollers, supporting the more complex USB protocol is reasonable for these devices.

Another serial interface that is widely implemented in embedded processors is known as **JTAG** (Joint Test Action Group), or more formally as the IEEE 1149.1 standard test access port and boundary-scan architecture. This interface appeared in the mid 1980s to solve the problem that integrated circuit packages and printed circuit board technology had evolved to the point that testing circuits using electrical probes had become difficult or impossible. Points in the circuit that needed to be accessed became inaccessible to probes. The notion of a **boundary scan** allows the state of a logical boundary of a circuit (what would traditionally have been pins accessible to probes) to be read or written serially through pins that are made accessible. Today, JTAG ports are widely used to provide a debug interface to embedded processors, enabling a PC-hosted debugging environment to examine and control the state of an embedded processor. The JTAG port is used, for example, to read out the state of processor registers, to set breakpoints in a program, and to single step through a program. A newer variant is **serial wire debug** (**SWD**), which provides similar functionality with fewer pins.

There are several other serial interfaces in use today, including for example **I^{2}C** (inter-integrated circuit), **SPI** (serial peripheral interface bus), **PCI Express** (peripheral component interconnect express), **FireWire**, **MIDI** (musical instrument digital interface), and serial versions of SCSI (described below). Each of these has its use. Also, network interfaces are typically serial.

10.1.4 Parallel Interfaces

A serial interface sends or receives a sequence of bits sequentially over a single line. A **parallel interface** uses multiple lines to simultaneously send bits. Of course, each line of a parallel interface is also a serial interface, but the logical grouping and coordinated action of these lines is what makes the interface a parallel interface.

Historically, one of the most widely used parallel interfaces is the IEEE-1284 printer port, which on the IBM PC used a DB-25 connector, as shown in Figure 10.4. This interface originated in 1970 with the Centronics model 101 printer, and hence is sometimes called a Centronics printer port. Today, printers are typically connected using USB or wireless networks.

With careful programming, a group of GPIO pins can be used together to realize a parallel interface. In fact, embedded system designers sometimes find themselves using GPIO pins to emulate an interface not supported directly by their hardware.

It seems intuitive that parallel interfaces should deliver higher performance than serial interfaces, because more wires are used for the interconnection. However, this is not necessarily the case. A significant challenge with parallel interfaces is maintaining synchrony across the multiple wires. This becomes more difficult as the physical length of the interconnection increases. This fact, combined with the requirement for bulkier cables and more I/O pins has resulted in many traditionally parallel interfaces being replaced by serial interfaces.

10.1.5 Buses

A **bus** is an interface shared among multiple devices, in contrast to a point-to-point interconnection linking exactly two devices. Busses can be serial interfaces (such as USB) or parallel interfaces. A widespread parallel bus is **SCSI** (pronounced scuzzy, for small computer system interface), commonly used to connect hard drives and tape drives to computers. Recent variants of SCSI interfaces, however, depart from the traditional parallel interface to become serial interfaces. SCSI is an example of a **peripheral bus** architecture, used to connect computers to peripherals such as sound cards and disk drives.

Other widely used peripheral bus standards include the **ISA bus** (industry standard architecture, used in the ubiquitous IBM PC architecture), **PCI** (peripheral component interface), and **Parallel ATA** (advanced technology attachment). A somewhat different kind of peripheral bus standard is **IEEE-488**, originally developed more than 30 years ago to connect automated test equipment to controlling computers. This interface was designed at Hewlett Packard and is also widely known as **HP-IB** (Hewlett Packard interface bus) and **GPIB** (general purpose interface bus). Many networks also use a bus architecture.

Because a bus is shared among several devices, any bus architecture must include a **media-access control** (**MAC**) protocol to arbitrate competing accesses. A simple MAC protocol has a single bus master that interrogates bus slaves. USB uses such a mechanism. An alternative is a **time-triggered bus**, where devices are assigned time slots during which they can transmit (or not, if they have nothing to send). A third alternative is a **token ring**, where devices on the bus must acquire a token before they can use the shared medium, and the token is passed around the devices according to some pattern. A fourth alternative is to use a bus arbiter, which is a circuit that handles requests for

the bus according to some priorities. A fifth alternative is **carrier sense multiple access** (**CSMA**), where devices sense the carrier to determine whether the medium is in use before beginning to use it, detect collisions that might occur when they begin to use it, and try again later when a collision occurs.

In all cases, sharing of the physical medium has implications on the timing of applications.

Example 10.7: A peripheral bus provides a mechanism for external devices to communicate with a CPU. If an external device needs to transfer a large amount of data to the main memory, it may be inefficient and/or disruptive to require the CPU to perform each transfer. An alternative is **direct memory access** (**DMA**). In the DMA scheme used on the ISA bus, the transfer is performed by a separate device called a **DMA controller** which takes control of the bus and transfers the data. In some more recent designs, such as PCI, the external device directly takes control of the bus and performs the transfer without the help of a dedicated DMA controller. In both cases, the CPU is free to execute software while the transfer is occurring, but if the executed code needs access to the memory or the peripheral bus, then the timing of the program is disrupted by the DMA. Such timing effects can be difficult to analyze.

10.2 Sequential Software in a Concurrent World

As we saw in Example 10.6, when software interacts with the external world, the timing of the execution of the software may be strongly affected. Software is intrinsically sequential, typically executing as fast as possible. The physical world, however, is concurrent, with many things happening at once, and with the pace at which they happen determined by their physical properties. Bridging this mismatch in semantics is one of the major challenges that an embedded system designer faces. In this section, we discuss some of the key mechanisms for accomplishing this.

10.2.1 Interrupts and Exceptions

An **interrupt** is a mechanism for pausing execution of whatever a processor is currently doing and executing a pre-defined code sequence called an **interrupt service routine** (**ISR**) or **interrupt handler**. Three kinds of events may trigger an interrupt. One is a **hardware interrupt**, where some external hardware changes the voltage level on an interrupt request line. In the case of a **software interrupt**, the program that is executing triggers the interrupt by executing a special instruction or by writing to a memory-mapped register. A third variant is called an **exception**, where the interrupt is triggered by internal hardware that detects a fault, such as a segmentation fault.

For the first two variants, once the ISR completes, the program that was interrupted resumes where it left off. In the case of an exception, once the ISR has completed, the program that triggered the exception is not normally resumed. Instead, the program counter is set to some fixed location where, for example, the operating system may terminate the offending program.

Upon occurrence of an interrupt trigger, the hardware must first decide whether to respond. If interrupts are disabled, it will not respond. The mechanism for enabling or disabling interrupts varies by processor. Moreover, it may be that some interrupts are enabled and others are not. Interrupts and exceptions generally have priorities, and an interrupt will be serviced only if the processor is not already in the middle of servicing an interrupt with a higher priority. Typically, exceptions have the highest priority and are always serviced.

When the hardware decides to service an interrupt, it will usually first disable interrupts, push the current program counter and processor status register(s) onto the stack, and branch to a designated address that will normally contain a jump to an ISR. The ISR must store on the stack the values currently in any registers that it will use, and restore their values before returning from the interrupt, so that the interrupted program can resume where it left off. Either the interrupt service routine or the hardware must also re-enable interrupts before returning from the interrupt.

Example 10.8: The ARM Cortex™ - M3 is a 32-bit microcontroller used in industrial automation and other applications. It includes a system timer called SysTick. This timer can be used to trigger an ISR to execute every 1ms. Suppose for example that every 1ms we would like to count down from some initial count until the count

reaches zero, and then stop counting down. The following C code defines an ISR that does this:

```
1     volatile uint timerCount = 0;
2     void countDown(void) {
3         if (timerCount != 0) {
4             timerCount--;
5         }
6     }
```

Here, the variable `timerCount` is a global variable, and it is decremented each time `countDown()` is invoked, until it reaches zero. We will specify below that this is to occur once per millisecond by registering `countDown()` as an ISR. The variable `timerCount` is marked with the C **volatile keyword**, which tells the compiler that the value of the variable will change at unpredictable times during execution of the program. This prevents the compiler from performing certain optimizations, such as caching the value of the variable in a register and reading it repeatedly. Using a C API provided by Luminary Micro® (2008c), we can specify that `countDown()` should be invoked as an interrupt service routine once per millisecond as follows:

```
1     SysTickPeriodSet(SysCtlClockGet() / 1000);
2     SysTickIntRegister(&countDown);
3     SysTickEnable();
4     SysTickIntEnable();
```

The first line sets the number of clock cycles between "ticks" of the SysTick timer. The timer will request an interrupt on each tick. `SysCtlClockGet()` is a library procedure that returns the number of cycles per second of the target platform's clock (e.g., 50,000,000 for a 50 MHz part). The second line registers the ISR by providing a **function pointer** for the ISR (the address of the `countDown()` procedure). (Note: Some configurations do not support run-time registration of ISRs, as shown in this code. See the documentation for your particular system.) The third line starts the clock, enabling ticks to occur. The fourth line enables interrupts.

The timer service we have set up can be used, for example, to perform some function for two seconds and then stop. A program to do that is:

```
1 int main(void) {
2     timerCount = 2000;
```

```
3       ... initialization code from above ...
4       while(timerCount != 0) {
5           ... code to run for 2 seconds ...
6       }
7   }
```

Processor vendors provide many variants of the mechanisms used in the previous example, so you will need to consult the vendor's documentation for the particular processor you are using. Since the code is not **portable** (it will not run correctly on a different processor), it is wise to isolate such code from your application logic and document carefully what needs to be re-implemented to target a new processor.

10.2.2 Atomicity

An interrupt service routine can be invoked between any two instructions of the main program (or between any two instructions of a lower priority ISR). One of the major challenges for embedded software designers is that reasoning about the possible interleavings of instructions can become extremely difficult. In the previous example, the interrupt service routine and the main program are interacting through a **shared variable**, namely `timerCount`. The value of that variable can change between any two **atomic opera-**

Basics: Timers

Microcontrollers almost always include some number of peripheral devices called **timer**s. A **programmable interval timer** (**PIT**), the most common type, simply counts down from some value to zero. The initial value is set by writing to a memory-mapped register, and when the value hits zero, the PIT raises an interrupt request. By writing to a memory-mapped control register, a timer might be set up to trigger repeatedly without having to be reset by the software. Such repeated triggers will be more precisely periodic than what you would get if the ISR restarts the timer each time it gets invoked. This is because the time between when the count reaches zero in the timer hardware and the time when the counter gets restarted by the ISR is difficult to control and variable. For example, if the timer reaches zero at a time when interrupts happen to be disabled, then there will be a delay before the ISR gets invoked. It cannot be invoked before interrupts are re-enabled.

tions of the main program. Unfortunately, it can be quite difficult to know what operations are atomic. The term "atomic" comes from the Greek work for "indivisible," and it is far from obvious to a programmer what operations are indivisible. If the programmer is writing assembly code, then it may be safe to assume that each assembly language instruction is atomic, but many ISAs include assembly level instructions that are not atomic.

Example 10.9: The ARM instruction set includes a LDM instruction, which loads multiple registers from consecutive memory locations. It can be interrupted part way through the loads (ARM Limited, 2006).

At the level of a C program, it can be even more difficult to know what operations are atomic. Consider a single, innocent looking statement

```
timerCount = 2000;
```

On an 8-bit microcontroller, this statement may take more than one instruction cycle to execute (an 8-bit word cannot store both the instruction and the constant 2000; in fact, the constant alone does not fit in an 8-bit word). An interrupt could occur part way through the execution of those cycles. Suppose that the ISR also writes to the variable `timerCount`. In this case, the final value of the `timerCount` variable may be composed of 8 bits set in the ISR and the remaining bits set by the above line of C, for example. The final value could be very different from 2000, and also different from the value specified in the interrupt service routine. Will this bug occur on a 32-bit microcontroller? The only way to know for sure is to fully understand the ISA and the compiler. In such circumstances, there is no advantage to having written the code in C instead of assembly language.

Bugs like this in a program are extremely difficult to identify and correct. Worse, the problematic interleavings are quite unlikely to occur, and hence may not show up in testing. For safety-critical systems, programmers have to make every effort to avoid such bugs. One way to do this is to build programs using higher-level concurrent models of computation, as discussed in Chapter 6. Of course, the implementation of those models of computation needs to be correct, but presumably, that implementation is constructed by experts in concurrency, rather than by application engineers.

When working at the level of C and ISRs, a programmer must carefully reason about the *order* of operations. Although many interleavings are possible, operations given as a

sequence of C statements must execute in order (more precisely, they must behave as if they had executed in order, even if out-of-order execution is used).

Example 10.10: In example 10.8, the programmer can rely on the statements within `main()` executing in order. Notice that in that example, the statement

```
timerCount = 2000;
```

appears before

```
SysTickIntEnable();
```

The latter statement enables the SysTick interrupt. Hence, the former statement cannot be interrupted by the SysTick interrupt.

10.2.3 Interrupt Controllers

An **interrupt controller** is the logic in the processor that handles interrupts. It supports some number of interrupts and some number of priority levels. Each interrupt has an **interrupt vector**, which is the address of an ISR or an index into an array called the **interrupt vector table** that contains the addresses of all the ISRs.

Example 10.11: The Luminary Micro LM3S8962 controller, shown in Figure 10.1, includes an ARM Cortex™ - M3 core microcontroller that supports 36 interrupts with eight priority levels. If two interrupts are assigned the same priority number, then the one with the lower vector will have priority over the one with the higher vector.

When an interrupt is asserted by changing the voltage on a pin, the response may be either **level triggered** or **edge triggered**. For level-triggered interrupts, the hardware asserting the interrupt will typically hold the voltage on the line until it gets an acknowledgement,

which indicates that the interrupt is being handled. For edge-triggered interrupts, the hardware asserting the interrupt changes the voltage for only a short time. In both cases, open collector lines can be used so that the same physical line can be shared among several devices (of course, the ISR will require some mechanism to determine which device asserted the interrupt, for example by reading a memory-mapped register in each device that could have asserted the interrupt).

Sharing interrupts among devices can be tricky, and careful consideration must be given to prevent low priority interrupts from blocking high priority interrupts. Asserting interrupts by writing to a designated address on a bus has the advantage that the same hardware can support many more distinct interrupts, but the disadvantage that peripheral devices get more complex. The peripheral devices have to include an interface to the memory bus.

10.2.4 Modeling Interrupts

The behavior of interrupts can be quite difficult to fully understand, and many catastrophic system failures are caused by unexpected behaviors. Unfortunately, the logic of interrupt controllers is often described in processor documentation very imprecisely, leaving many possible behaviors unspecified. One way to make this logic more precise is to model it as an FSM.

Example 10.12: The program of Example 10.8, which performs some action for two seconds, is shown in Figure 10.5 together with two finite state machines that model the ISR and the main program. The states of the FSMs correspond to positions in the execution labeled A through E, as shown in the program listing. These positions are between C statements, so we are assuming here that these statements are atomic operations (a questionable assumption in general).

We may wish to determine whether the program is assured of always reaching position C. In other words, can we assert with confidence that the program will eventually move beyond whatever computation it was to perform for two seconds? A state machine model will help us answer that question.

The key question now becomes how to compose these state machines to correctly model the interaction between the two pieces of sequential code in the procedures `ISR` and `main`. It is easy to see that asynchronous composition is not the right

```
    volatile uint timerCount = 0;
    void ISR(void) {
       … disable interrupts
D→  if(timerCount != 0) {
E→     timerCount--;
       }
       … enable interrupts
    }
    int main(void) {
       // initialization code
       SysTickIntRegister(&ISR);
       … // other init
       timerCount = 2000;
A→  while(timerCount != 0) {
B→    … code to run for 2 seconds
       }
    }
C→  … whatever comes next
```

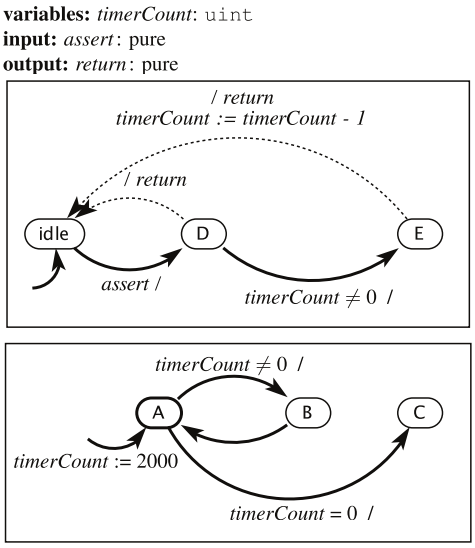


Figure 10.5: State machine models and main program for a program that does something for two seconds and then continues to do something else.

choice because the interleavings are not arbitrary. In particular, `main` can be interrupted by `ISR`, but `ISR` cannot be interrupted by `main`. Asynchronous composition would fail to capture this asymmetry.

Assuming that the interrupt is always serviced immediately upon being requested, we wish to have a model something like that shown in Figure 10.6. In that figure, a two-state FSM models whether an interrupt is being serviced. The transition from Inactive to Active is triggered by a pure input *assert*, which models the timer hardware requesting interrupt service. When the ISR completes its execution, another pure input *return* triggers a return to the Inactive state. Notice here that the transition from Inactive to Active is a preemptive transition, indicated by the small circle at the start of the transition, suggesting that it should be taken immediately when *assert* occurs, and that it is a reset transition, suggesting that the state refinement of Active should begin in its initial state upon entry.

If we combine Figures 10.5 and 10.6 we get the hierarchical FSM in Figure 10.7. Notice that the *return* signal is both an input and an output now. It is an output produced by the state refinement of Active, and it is an input to the top-level FSM, where it triggers a transition to Inactive. Having an output that is also an input provides a mechanism for a state refinement to trigger a transition in its container state machine.

To determine whether the program reaches state C, we can study the flattened state machine shown in Figure 10.8. Studying that machine carefully, we see that in fact there is no assurance that state C will be reached! If, for example, *assert* is present on every reaction, then C is never reached.

Could this happen in practice? With this program, it is improbable, but not impossible. It could happen if the ISR itself takes longer to execute than the time between interrupts. Is there any assurance that this will not happen? Unfortunately, our only assurance is a vague notion that processors are faster than that. There is no guarantee.

In the above example, modeling the interaction between a main program and an interrupt service routine exposes a potential flaw in the program. Although the flaw may be unlikely to occur in practice in this example, the fact that the flaw is present at all is disturbing. In any case, it is better to know that the flaw is present, and to decide that the risk is acceptable, than to not know it is present.

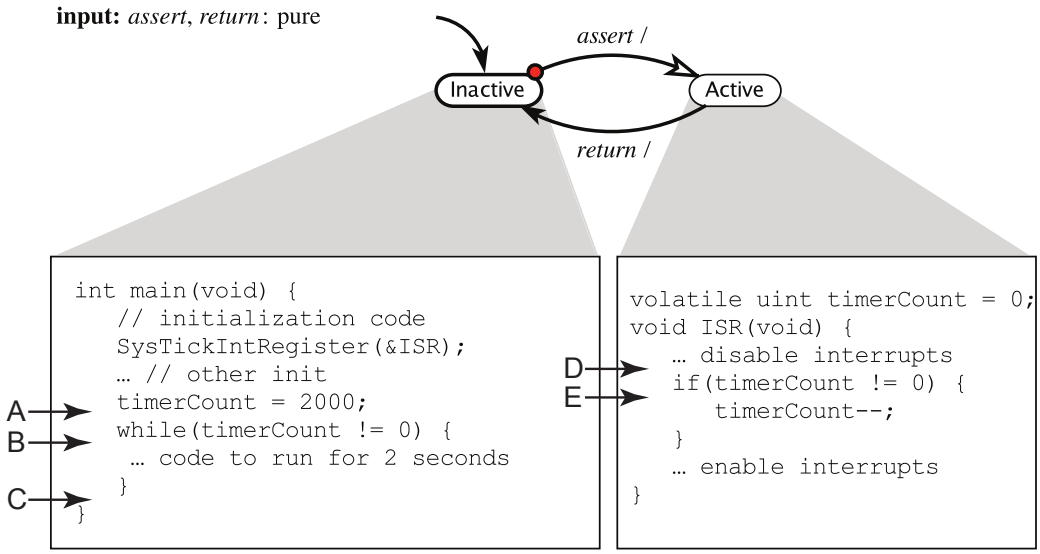


Figure 10.6: Sketch of a state machine model for the interaction between an ISR and the main program.

Interrupt mechanisms can be quite complex. Software that uses these mechanisms to provide I/O to an external device is called a **device driver**. Writing device drivers that are correct and robust is a challenging engineering task requiring a deep understanding of the architecture and considerable skill reasoning about concurrency. Many failures in computer systems are caused by unexpected interactions between device drivers and other programs.

10.3 Summary

This chapter has reviewed hardware and software mechanisms used to get sensor data into processors and commands from the processor to actuators. The emphasis is on understanding the principles behind the mechanisms, with a particular focus on the bridging between the sequential world of software and the parallel physical world.

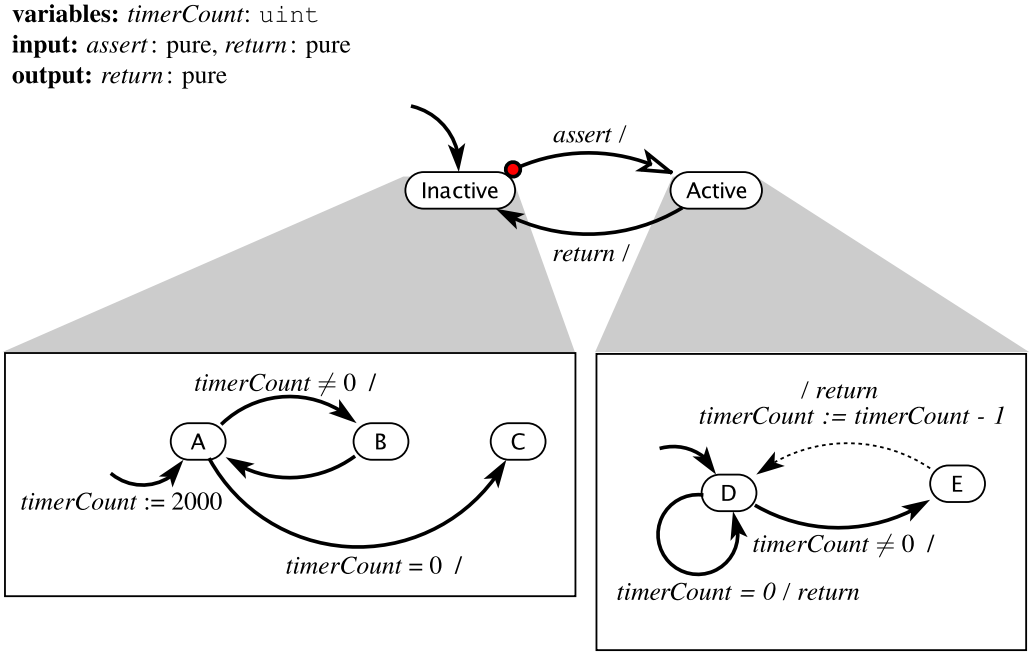


Figure 10.7: Hierarchical state machine model for the interaction between an ISR and the main program.

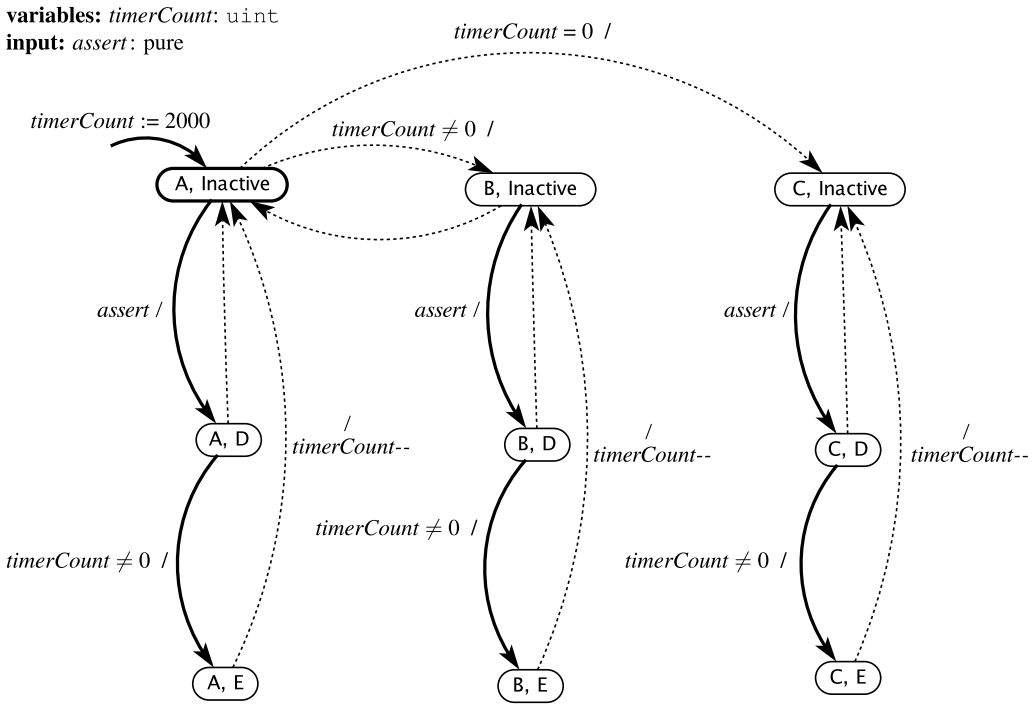


Figure 10.8: Flattened version of the hierarchical state machine in Figure 10.7.

Exercises

1. Similar to Example 10.6, consider a C program for an Atmel AVR that uses a UART to send 8 bytes to an RS-232 serial interface, as follows:

```
1 for(i = 0; i < 8; i++) {
2   while(!(UCSR0A & 0x20));
3   UDR0 = x[i];
4 }
```

   Assume the processor runs at 50 MHz; also assume that initially the UART is idle, so when the code begins executing, `UCSR0A & 0x20 == 0x20` is true; further, assume that the serial port is operating at 19,200 baud. How many cycles are required to execute the above code? You may assume that the `for` statement executes in three cycles (one to increment `i`, one to compare it to 8, and one to perform the conditional branch); the `while` statement executes in 2 cycles (one to compute `!(UCSR0A & 0x20)` and one to perform the conditional branch); and the assigment to `UDR0` executes in one cycle.

2. Figure 10.9 gives the sketch of a program for an Atmel AVR microcontroller that performs some function repeatedly for three seconds. The function is invoked by calling the procedure foo(). The program begins by setting up a timer interrupt to occur once per second (the code to do this setup is not shown). Each time the interrupt occurs, the specified interrupt service routine is called. That routine decrements a counter until the counter reaches zero. The main() procedure initializes the counter with value 3 and then invokes foo() until the counter reaches zero.

   (a) We wish to assume that the segments of code in the grey boxes, labeled **A**, **B**, and **C**, are atomic. State conditions that make this assumption valid.

   (b) Construct a state machine model for this program, assuming as in part (a) that **A**, **B**, and **C**, are atomic. The transitions in your state machine should be labeled with "guard/action", where the action can be any of **A**, **B**, **C**, or nothing. The actions **A**, **B**, or **C** should correspond to the sections of code in the grey boxes with the corresponding labels. You may assume these actions are atomic.

   (c) Is your state machine deterministic? What does it tell you about how many times foo() may be invoked? Do all the possible behaviors of your model correspond to what the programmer likely intended?

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;

// Interrupt service routine.
SIGNAL(SIG_OUTPUT_COMPARE1A) {
  if(timer_count > 0) {
    timer_count--;                    A
  }
}

// Main program.
int main(void) {
  // Set up interrupts to occur
  // once per second.
  ...
  // Start a 3 second timer.
  timer_count = 3;                    B

  // Do something repeatedly
   // for 3 seconds.
  while(timer_count > 0) {
    foo();                            C
  }
}
```

Figure 10.9: Sketch of a C program that performs some function by calling procedure foo() repeatedly for 3 seconds, using a timer interrupt to determine when to stop.

Note that there are many possible answers. Simple models are preferred over elaborate ones, and complete ones (where everything is defined) over incomplete ones. Feel free to give more than one model.

3. In a manner similar to example 10.8, create a C program for the ARM Cortex™ - M3 to use the SysTick timer to invoke a system-clock ISR with a jiffy interval of 10 ms that records the time since system start in a 32-bit int. How long can this program run before your clock overflows?

4. Consider a dashboard display that displays "normal" when brakes in the car operate normally and "emergency" when there is a failure. The intended behavior is that once "emergency" has been displayed, "normal" will not again be displayed. That is, "emergency" remains on the display until the system is reset.

   In the following code, assume that the variable `display` defines what is displayed. Whatever its value, that is what appears on the dashboard.

```
1  volatile static uint8_t alerted;
2  volatile static char* display;
3  void ISRA() {
4      if (alerted == 0) {
5          display = "normal";
6      }
7  }
8  void ISRB() {
9      display = "emergency";
10     alerted = 1;
11 }
12 void main() {
13     alerted = 0;
14     ...set up interrupts...
15     ...enable interrupts...
16     ...
17 }
```

   Assume that `ISRA` is an interrupt service routine that is invoked when the brakes are applied by the driver. Assume that `ISRB` is invoked if a sensor indicates that the brakes are being applied at the same time that the accelerator pedal is depressed. Assume that neither ISR can interrupt itself, but that `ISRB` has higher priority than

`ISRA`, and hence `ISRB` can interrupt `ISRA`, but `ISRA` cannot interrupt `ISRB`. Assume further (unrealistically) that each line of code is atomic.

(a) Does this program always exhibit the intended behavior? Explain. In the remaining parts of this problem, you will construct various models that will either demonstrate that the behavior is correct or will illustrate how it can be incorrect.

(b) Construct a determinate extended state machine modeling `ISRA`. Assume that:

- `alerted` is a variable of type $\{0, 1\} \subset$ `uint8_t`,
- there is a pure input *A* that when present indicates an interrupt request for `ISRA`, and
- `display` is an output of type `char*`.

(c) Give the size of the state space for your solution.

(d) Explain your assumptions about when the state machine in (b) reacts. Is this time triggered, event triggered, or neither?

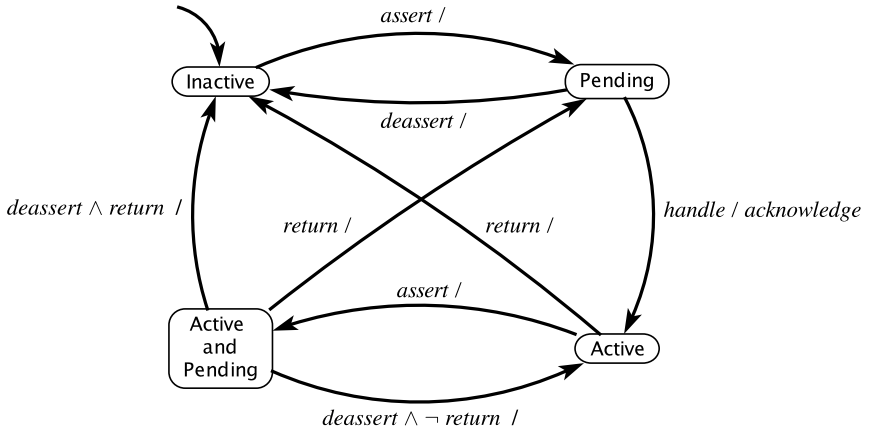
(e) Construct a determinate extended state machine modeling `ISRB`. This one has a pure input *B* that when present indicates an interrupt request for `ISRB`.

(f) Construct a flat (non-hierarchical) determinate extended state machine describing the joint operation of the these two ISRs. Use your model to argue the correctness of your answer to part (a).

(g) Give an equivalent hierarchical state machine. Use your model to argue the correctness of your answer to part (a).

5. Suppose a processor handles interrupts as specified by the following FSM:

input: *assert*, *deassert*, *handle*, *return*: pure
output: *acknowledge*



Here, we assume a more complicated interrupt controller than that considered in Example 10.12, where there are several possible interrupts and an arbiter that decides which interrupt to service. The above state machine shows the state of one interrupt. When the interrupt is asserted, the FSM transitions to the Pending state, and remains there until the arbiter provides a *handle* input. At that time, the FSM transitions to the Active state and produces an *acknowledge* output. If another interrupt is asserted while in the Active state, then it transitions to Active and Pending. When the ISR returns, the input *return* causes a transition to either Inactive or Pending, depending on the starting point. The *deassert* input allows external hardware to cancel an interrupt request before it gets serviced.

Answer the following questions.

(a) If the state is Pending and the input is *return*, what is the reaction?

(b) If the state is Active and the input is *assert* ∧ *deassert*, what is the reaction?

(c) Suppose the state is Inactive and the input sequence in three successive reactions is:

i. *assert* ,
ii. *deassert* ∧ *handle* ,
iii. *return* .

What are all the possible states after reacting to these inputs? Was the interrupt handled or not?

(d) Suppose that an input sequence never includes *deassert*. Is it true that every *assert* input causes an *acknowledge* output? In other words, is every interrupt request serviced? If yes, give a proof. If no, give a counterexample.

6. Suppose you are designing a processor that will support two interrupts whose logic is given by the FSM in Exercise 5. Design an FSM giving the logic of an arbiter that assigns one of these two interrupts higher priority than the other. The inputs should be the following pure signals:

$$assert1, return1, assert2, return2$$

to indicate requests and return from interrupt for interrupts 1 and 2, respectively. The outputs should be pure signals *handle1* and *handle2*. Assuming the *assert* inputs are generated by two state machines like that in Exercise 5, can you be sure that this arbiter will handle every request that is made? Justify your answer.

7. Consider the following program that monitors two sensors. Here `sensor1` and `sensor2` denote the variables storing the readouts from two sensors. The actual read is performed by the functions `readSensor1()` and `readSensor2()`, respectively, which are called in the interrupt service routine `ISR`.

```
1  char flag = 0;
2  volatile char* display;
3  volatile short sensor1, sensor2;
4
5  void ISR() {
6    if (flag) {
7       sensor1 = readSensor1();
8    } else {
9       sensor2 = readSensor2();
10   }
11 }
12
13 int main() {
14   // ... set up interrupts ...
15   // ... enable interrupts ...
16   while(1) {
17      if (flag) {
18         if isFaulty2(sensor2) {
19            display = "Sensor2 Faulty";
20         }
```

```
21        } else {
22           if isFaulty1(sensor1) {
23              display = "Sensor1 Faulty";
24           }
25        }
26        flag = !flag;
27     }
28  }
```

Functions `isFaulty1()` and `isFaulty2()` check the sensor readings for any discrepancies, returning 1 if there is a fault and 0 otherwise. Assume that the variable `display` defines what is shown on the monitor to alert a human operator about faults. Also, you may assume that `flag` is modified only in the body of `main`.

Answer the following questions:

(a) Is it possible for the `ISR` to update the value of `sensor1` while the main function is checking whether `sensor1` is faulty? Why or why not?

(b) Suppose a spurious error occurs that causes `sensor1` or `sensor2` to be a faulty value for one measurement. Is it possible for that this code would not report "Sensor1 faulty" or "Sensor2 faulty"?

(c) Assuming the interrupt source for `ISR()` is timer-driven, what conditions would cause this code to never check whether the sensors are faulty?

(d) Suppose that instead being interrupt driven, `ISR` and `main` are executed concurrently, each in its own thread. Assume a microkernel that can interrupt any thread at any time and switch contexts to execute another thread. In this scenario, is it possible for the `ISR` to update the value of `sensor1` while the main function is checking whether `sensor1` is faulty? Why or why not?