Chapter 3 *Standard single-purpose processors: Peripherals*

3.1 Introduction

A single-purpose processor is a digital system intended to solve a specific computation task. The processor may be a *standard* one, intended for use in a wide variety of applications in which the same task must be performed. The manufacturer of such an off-the-shelf processor sells the device in large quantities. On the other hand, the processor may be a *custom* one, built by a designer to implement a task specific to a particular application. An embedded system designer choosing to use a standard single-purpose, rather than a general-purpose, processor to implement part of a system's functionality may achieve several benefits.

First, performance may be fast, since the processor is customized for the particular task at hand. Not only might the task execute in fewer clock cycles, but also those cycles themselves may be shorter. Fewer clock cycles may result from many datapath components operating in parallel, from datapath components passing data directly to one another without the need for intermediate registers (chaining), or from elimination of program memory fetches. Shorter cycles may result from simpler functional units, less multiplexors, or simpler control logic. For standard single-purpose processors, manufacturers may spread NRE cost over many units. Thus, the processor's clock cycle may be further reduced by the use of custom IC technology, leading-edge IC's, and expert designers, just as is the case with general-purpose processors.

Second, size may be small. A single-purpose processor does not require a program memory. Also, since it does not need to support a large instruction set, it may have a simpler datapath and controller.

Third, a standard single-purpose processor may have low unit cost, due to the manufacturer spreading NRE cost over many units. Likewise, NRE cost may be low, since the embedded system designer need not design a standard single-purpose processor, and may not even need to program it.

There are of course tradeoffs. If we are already using a general-purpose processor, then implementing a task on an additional single-purpose processor rather than in software may add to the system size and power consumption.

In this chapter, we describe the basic functionality of several standard single-purpose processors commonly found in embedded systems. The level of detail of the description is intended to be enough to enable using such processors, but not necessarily to design one.

We often refer to standard single-purpose processors as *peripherals,* because they usually exist on the periphery of the CPU. However, microcontrollers tightly integrate these peripherals with the CPU, often placing them on-chip, and even assigning peripheral registers to the CPU's own register space. The result is the common term "on-chip peripherals," which some may consider somewhat of an oxymoron.

3.2 Timers, counters, and watchdog timers

A *timer* is a device that generates a signal pulse at specified time intervals. A time interval is a "real-time" measure of time, such as 3 milliseconds. These devices are extremely useful in systems in which a particular action, such as sampling an input signal or generating an output signal, must be performed every *X* time units.

Internally, a simple timer may consist of a register, counter, and an extremely simple controller. The register holds a count value representing the number of clock cycles that equals the desired real-time value. This number can be computed using the simple formula:

Number of clock cycles = Desired real-time value / Clock cycle

For example, to obtain a duration of 3 milliseconds from a clock cycle of 10 nanoseconds (100 MHz), we must count ($3x10^{-6}$ s / $10x10^{-9}$ s/cycle) = 300 cycles. The counter is initially loaded with the count value, and then counts down on every clock cycle until 0 is reached, at which point an output signal is generated, the count value is reloaded, and the process repeats itself.

To use a timer, we must configure it (write to its registers), and respond to its output signal. When we use a timer in conjunction with a general-purpose processor, we typically respond to the timer signal by assigning it to an interrupt, so we include the desired action in an interrupt service routine. Many microcontrollers that include built-in timers will have special interrupts just for its timers, distinct from external interrupts.

Note that we could use a general-purpose processor to implement a timer. Knowing the number of cycles that each instruction requires, we could write a loop that executed the desired number of instructions; when this loop completes, we know that the desired time passed. This implementation of a timer on a dedicated general-purpose processor is obviously quite inefficient in terms of size. One could alternatively incorporate the timer functionality into a main program, but the timer functionality then occupies much of the program's run time, leaving little time for other computations. Thus, the benefit of assigning timer functionality to a special-purpose processor becomes evident.

A *counter* is nearly identical to a timer, except that instead of counting clock cycles (pulses on the clock signal), a counter counts pulses on some other input signal.

A *watchdog timer* can be thought of as having the inverse functionality than that of a regular timer. We configure a watchdog timer with a real-time value, just as with a regular timer. However, instead of the timer generating a signal for us every X time units, *we* must generate a signal for the timer every X time units. If we fail to generate this signal in time, then the timer generates a signal indicating that we failed. We often connect this signal to the reset or interrupt signal of a general-purpose processor. Thus, a watchdog timer provides a mechanism of ensuring that our software is working properly; every so often in the software, we include a statement that generates a signal to the watchdog timer (in particular, that resets the timer). If something undesired happens in the software (e.g., we enter an undesired infinite loop, we wait for an input signal that never arrives, a part fails, etc.), the watchdog generates a signal that we can use to restart or test parts of the system. Using an interrupt service routine, we may record information as to the number of failures and the causes of each, so that a service technician may later evaluate this information to determine if a particular part requires replacement. Note that an embedded system often must recover from failures whenever possible, as the user may not have the means to reboot the system in the same manner that he/she might reboot a desktop system.

3.3 UART

A *UART* (Universal Asynchronous Receiver/Transmitter) receives serial data and stores it as parallel data (usually one byte), and takes parallel data and transmits it as serial data. The principles of serial communication appear in a later chapter.

Such serial communication is beneficial when we need to communicate bytes of data between devices separated by long distances, or when we simply have few available I/O pins. Principles of serial communication will be discussed in a later chapter. For our purpose in this section, we must be aware that we must set the transmission and reception rate, called the baud rate, which indicates the frequency that the signal changes. Common rates include 2400, 4800, 9600, and 19.2k. We must also be aware that an extra bit may be added to each data word, called parity, to detect transmission errors -- the parity bit is set to high or low to indicate if the word has an even or odd number of bits.

Internally, a simple UART may possess a baud-rate configuration register, and two independently operating processors, one for receiving and the other for transmitting. The transmitter may possess a register, often called a transmit buffer, that holds data to be sent. This register is a shift register, so the data can be transmitted one bit at a time by shifting at the appropriate rate. Likewise, the receiver receives data into a shift register,

and then this data can be read in parallel. Note that in order to shift at the appropriate rate based on the configuration register, a UART requires a timer.

To use a UART, we must configure its baud rate by writing to the configuration register, and then we must write data to the transmit register and/or read data from the received register. Unfortunately, configuring the baud rate is usually not as simple as writing the desired rate (e.g., 4800) to a register. For example, to configure the UART of an 8051, we must use the following equation:

$$Baudrate = (2^{s\bmod} / 32) * oscfreq / (12 * (256 - TH1)))$$

smod corresponds to 2 bits in a special-function register, *oscfreq* is the frequency of the oscillator, and *TH1* is an 8-bit rate register of a built-in timer.

Note that we could use a general-purpose processor to implement a UART completely in software. If we used a dedicated general-processor, the implementation would be inefficient in terms of size. We could alternatively integrate the transmit and receive functionality with our main program. This would require creating a routine to send data serially over an I/O port, making use of a timer to control the rate. It would also require using an interrupt service routine to capture serial data coming from another I/O port whenever such data begins arriving. However, as with the timer functionality, adding send and receive functionality can detract from time for other computations.

Knowing the number of cycles that each instruction requires, we could write a loop that executed the desired number of instructions; when this loop completes, we know that the desired time passed. This implementation of a timer on a dedicated general-purpose processor is obviously quite inefficient in terms of size. One could alternatively incorporate the timer functionality into a main program, but the timer functionality then occupies much of the program's run time, leaving little time for other computations. Thus, the benefit of assigning timer functionality to a special-purpose processor becomes evident.

3.4 Pulse width modulator

A *pulse-width modulator* (PWM) generates an output signal that repeatedly switches between high and low. We control the duration of the high value and of the low value by indicating the desired period, and the desired *duty cycle*, which is the percentage of time the signal is high compared to the signal's period. A *square wave* has a duty cycle of 50%. The pulse's width corresponds to the pulse's time high.

Again, PWM functionality could be implemented on a dedicated general-purpose processor, or integrated with another program's functionality, but the single-purpose processor approach has the benefits of efficiency and simplicity.

One common use of a PWM is to control the average current or voltage input to a device. For example, a DC motor rotates when power is applied, and this power can be turned on and off by setting an input high or low. To control the speed, we can adjust the input voltage, but this requires a conversion of our high/low digital signals to an analog signal. Fortunately, we can also adjust the speed simply by modifying the duty cycle of the motors on/off input, an approach which adjusts the average voltage. This approach works because a DC motor does not come to an immediate stop when power is turned off, but rather it coasts, much like a bicycle coasts when we stop pedaling. Increasing the duty cycle increases the motor speed, and decreasing the duty cycle decreases the speed. This duty cycle adjustment principle applies to the control other types of electric devices, such as dimmer lights.

Another use of a PWM is to encode control commands in a single signal for use by another device. For example, we may control a radio-controlled car by sending pulses of different widths. Perhaps a 1 ms width corresponds to a turn left command, a 4 ms width to turn right, and 8 ms to forward.

3.5 LCD controller

An *LCD (Liquid crystal display)* is a low-cost, low-power device capable of displaying text and images. LCDs are extremely common in embedded systems, since such systems often do not have video monitors standard for desktop systems. LCDs can be found in numerous common devices like watches, fax and copy machines, and calculators.

The basic principle of one type of LCD (reflective) works as follows. First, incoming light passes through a polarizing plate. Next, that polarized light encounters liquid crystal material. If we excite a region of this material, we cause the material's molecules to align, which in turn causes the polarized light to pass through the material. Otherwise, the light does not pass through. Finally, light that has passed through hits a mirror and reflects back, so the excited region appears to light up. Another type of LCD (absorption) works similarly, but uses a black surface instead of a mirror. The surface below the excited region absorbs light, thus appearing darker than the other regions.

One of the simplest LCDs is 7-segment LCD. Each of the 7 segments can be activated to display any digit character or one of several letters and symbols. Such an LCD may have 7 inputs, each corresponding to a segment, or it may have only 4 inputs to represent the numbers 0 through 9. An LCD driver converts these inputs to the electrical signals necessary to excite the appropriate LCD segments.

A dot-matrix LCD consists of a matrix of dots that can display alphanumeric characters (letters and digits) as well as other symbols. A common dot-matrix LCD has 5 columns and 8 rows of dots for one character. An LCD driver converts input data into the appropriate electrical signals necessary to excite the appropriate LCD bits.

Each type of LCD may be able to display multiple characters. In addition, each character may be displayed in normal or inverted fashion. The LCD may permit a character to be blinking (cycling through normal and inverted display) or may permit display of a cursor (such as a blinking underscore) indicating the "current" character. This functionality would be difficult for us to implement using software. Thus, we use an *LCD controller* to provide us with a simple interface, perhaps 8 data inputs and one enable input. To send a byte to the LCD, we provide a value to the 8 inputs and pulse the enable. This byte may be a control word, which instructs the LCD controller to initialize the LCD, clear the display, select the position of the cursor, brighten the display, and so on. Alternatively, this byte may be a data word, such as an ASCII character, instructing the LCD to display the character at the currently-selected display position.

3.6 Keypad controller

A *keypad* consists of a set of buttons that may be pressed to provide input to an embedded system. Again, keypads are extremely common in embedded systems, since such systems may lack the keyboard that comes standard with desktop systems.

A simple keypad has buttons arranged in an N-column by M-row grid. The device has N outputs, each output corresponding to a column, and another M outputs, each output corresponding to a row. When we press a button, one column output and one row output go high, uniquely identifying the pressed button. To read such a keypad from software, we must scan the column and row outputs.

The scanning may instead be performed by a *keypad controller* (actually, such a device decodes rather than controls, but we'll call it a controller for consistency with the other peripherals discussed). A simple form of such a controller scans the column and row outputs of the keypad. When the controller detects a button press, it stores a code corresponding to that button into a register and sets an output high, indicating that a button has been pressed. Our software may poll this output every 100 milliseconds or so, and read the register when the output is high. Alternatively, this output can generate an interrupt on our general-purpose processor, eliminating the need for polling.

3.7 Stepper motor controller

A *stepper motor* is an electric motor that rotates a fixed number of degrees whenever we apply a "step" signal. In contrast, a regular electric motor rotates continuously whenever power is applied, coasting to a stop when power is removed. We specify a stepper motor either by the number of degrees in a single step, such as 1.8E, or by the number of steps required to move 360E, such as 200 steps. Stepper motors obviously abound in embedded systems with moving parts, such as disk drives, printers, photocopy and fax machines, robots, camcorders, VCRs, etc.

Internally, a stepper motor typically has four coils. To rotate the motor one step, we pass current through one or two of the coils; the particular coils depends on the present orientation of the motor. Thus, rotating the motor 360E requires applying current to the coils in a specified sequence. Applying the sequence in reverse causes reversed rotation.

In some cases, the stepper motor comes with four inputs corresponding to the four coils, and with documentation that includes a table indicating the proper input sequence. To control the motor from software, we must maintain this table in software, and write a step routine that applies high values to the inputs based on the table values that follow the previously-applied values.

In other cases, the stepper motor comes with a built-in controller (i.e., a special-purpose processor) implementing this sequence. Thus, we merely create a pulse on an input signal of the motor, causing the controller to generate the appropriate high signals to the coils that will cause the motor to rotate one step.

3.8 Analog-digital converters

An *analog-to-digital* converter (ADC, A/D or A2D) converts an analog signal to a digital signal, and a *digital-to-analog* converter (DAC, D/A or D2A) does the opposite. Such conversions are necessary because, while embedded systems deal with digital values, an embedded system's surroundings typically involve many analog signals. Analog refers to continuously-valued signal, such as temperature or speed represented by a voltage between 0 and 100, with infinite possible values in between. "Digital" refers to discretely-valued signals, such as integers, and in computing systems, these signals are encoded in binary. By converting between analog and digital signals, we can use digital processors in an analog environment.

For example, consider the analog signal of Figure 3.1(a). The analog input voltage varies over time from 1 to 4 Volts. We sample the signal at successive time units, and encode the current voltage into a 4-bit binary number. Conversely, consider Figure 3.1(b). We want to generate an analog output voltage for the given binary numbers over time. We generate the analog signal shown.

We can compute the digital values from the analog values, and vice-versa, using the following ratio:

$$\frac{e}{V_{\max}} = \frac{d}{2^n - 1}$$

V_{max} is the maximum voltage that the analog signal can assume, n is the number of bits available for the digital encoding, d is the present digital encoding, and e is the present analog voltage. This proportionality of the voltage and digital encoding is shown graphically in Figure 3.1(c). In our example of Figure 3.1, suppose V_{max} is 7.5V. Then for e = 5V, we have the following ratio: 5/7.5 = d/15, resulting in d = 1010 (ten), as shown in Figure 3.1(c). The *resolution* of a DAC or ADC is defined as $V_{max}/(2^n-1)$, representing the number of volts between successive digital encodings. The above discussion assumes a minimum voltage of 0V.

Internally, DACs possess simpler designs than ADCs. A DAC has n inputs for the digital encoding d, a V_{max} analog input, and an analog output e. A fairly straightforward circuit (involving resistors and an op-amp) can be used to convert d to e.

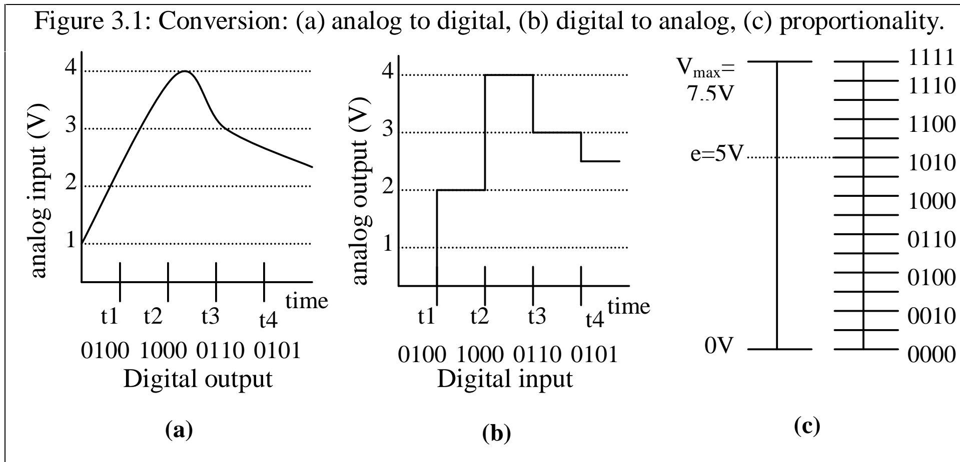

Figure 3.1: Conversion: (a) analog to digital, (b) digital to analog, (c) proportionality.

ADCs, on the other hand, require designs that are more complex, for the following reason. Given a V_{max} analog input and an analog input *e*, how does the converter know what binary value to assign in order to satisfy the above ratio? Unlike DACs, there is no simple analog circuit to compute *d* from *e*. Instead, an ADC may itself contain a DAC also connected to V_{max}. The ADC "guesses" an encoding *d*, and then evaluates its guess by inputting *d* into the DAC, and comparing the generated analog output *e'* with the original analog input *e* (using an analog comparator). If the two sufficiently match, then the ADC has found a proper encoding. So now the question remains: how do we guess the correct encoding?

This problem is analogous to the common computer-programming problem of finding an item in a list. One approach is sequential search, or "counting-up" in analog-digital terminology. In this approach, we start with an encoding of 0, then 1, then 2, etc., until we find a match. Unfortunately, while simple, this approach in the worst case (for high voltage values) requires 2^n comparisons, so it may be quite slow.

A faster solution uses what programmers call binary search, or "successive approximation" in analog-digital terminology. We start with an encoding corresponding half of the maximum. We then compare the resulting analog value with the original; if the resulting value is greater (less) than the original, we set the new encoding to halfway between this one and the maximum (minimum). We continue this process, dividing the possible encoding range in half at each step, until the compared voltages are equal. This technique requires at most *n* comparisons. However, it requires a more complex converter.

Because ADCs must guess the correct encoding, they require some time. Thus, in addition to the analog input and digital output, they include an input "start" that starts the conversion, and an ouput "done" to indicate that the conversion is complete.

3.9 Real-time clocks

Much like a digital wristwatch, a *real-time clock* (RTC) keeps the time and date in an embedded system. Read-time clocks are typically composed of a crystal-controlled oscillator, numerous cascaded counters, and a battery backup. The crystal-controlled oscillator generates a very consistent high-frequency digital pulse that feed the cascaded counters. The first counter, typically, counts these pulses up to the oscillator frequency, which corresponds to exactly one second. At this point, it generates a pulse that feeds the next counter. This counter counts up to 59, at which point it generates a pulse feeding the minute counter. The hour, date, month and year counters work in similar fashion. In addition, real-time clocks adjust for leap years. The rechargeable back-up battery is used to keep the real-time clock running while the system is powered off.

From the micro-controller's point of view, the content of these counters can be set to a desired value, (this corresponds to setting the clock), and retrieved. Communication

between the micro-controller and a real-time clock is accomplished through a serial bus, such as I^2C. It should be noted that, given a timer peripheral, it is possible to implement a real-time clock in software running on a processor. In fact, many systems use this approach to maintain the time. However, the drawback of such systems is that when the processor is shut down or reset, the time is lost.

3.10 Summary

Numerous single-purpose processors are manufactured to fulfill a specific function in a variety of embedded systems. These standard single-purpose processors may be fast, small, and have low unit and NRE costs. A timer informs us when a particular interval of time as passed, while a watchdog timer requires us to signal it within a particular interval to indicate that a program is running without error. A counter informs us when a particular number of pulses have occurred on a signal. A UART converts parallel data to serial data, and vice-versa. A PWM generates pulses on an output signal, with specific high and low times. An LCD controller simplifies the writing of characters to an LCD. A keypad controller simplifies capture and decoding of a button press. A stepper-motor controller assists us to rotate a stepper motor a fixed amount forwards or backwards. ADCs and DACs convert analog signals to digital, and vice-versa. A real-time clock keeps track of date and time. Most of these single-purpose processors could be implemented as software on a general-purpose processor, but such implementation can be burdensome. These processors thus simplify embedded system design tremendously. Many microcontrollers integrate these processors on-chip.

3.11 References and further reading

[1] Embedded Systems Programming. Includes information on a variety of single-purpose processors, such as programs for implementing or using timers and UARTs on microcontrollers.

[2] Microcontroller technology: the 68HC11. Peter Spasov. 2nd edition. ISBN 0-13-362724-1. Prentice Hall, Englewood Cliffs, NJ, 1996. Contains descriptions of principles and details for common 68HC11 peripherals.

3.12 Exercises

1. Given a clock frequency of 10 MHz, determine the number of clock cycles corresponding to a real-time interval of 100 ms.
2. A particular motor operates at 10 revolutions per second when its controlling input voltage is 3.7 V. Assume that you are using a microcontroller with a PWM whose output port can be set high (5 V) or low (0 V). (a) Compute the duty cycle necessary to obtain 10 revolutions per second. (b) Provide values for a pulse width and period that achieve this duty cycle.
3. Given a 120-step stepper motor with its own controller, write a C function *Rotate*(int degrees), which, given the desired rotation amount in degrees (between 0 and 360), pulses a microcontroller's output port the correct number of times to achieve the desired rotation.
4. Given an analog output signal whose voltage should range from 0 to 10 V, and a 8-bit digital encoding, provide the encodings for the following desired voltages: (a) 0 V, (b) 1 V, (c) 5.33 V, (d) 10 V. (e) What is the resolution of our conversion?
5. Given an analog input signal whose voltage ranges from 0 to 5 V, and an 8-bit digital encoding, calculate the correct encoding, and then trace the successive-approximation approach (i.e., list all the guessed encodings in the correct order) to finding the correct encoding.
6. Determine the values for *smod* and *TH1* to generate a baud rate of 9600 for the 8051 baud rate equation in the chapter, assuming an 11.981 MHz oscillator. Remember that *smod* is 2 bits and *TH1* is 8 bits.