Building reliable applications with microcontroller

A world without Microcontrollers

chapter 1

What is an embedded system?

Reminder

1 second (s)	= 1.0 second (10^0 seconds) = 1000 ms.
1 millisecond (ms)	= 0.001 seconds (10^{-3} seconds) = 1000 μs.
1 microsecond (μs)	= 0.000001 seconds (10^{-6} seconds) = 1000 ns.
1 nanosecond (ns)	= 0.000000001 seconds (10^{-9} seconds).

1.4 Real-time systems

Users of most software systems like to have their applications respond quickly: the difference is that in most information systems and general desktop applications, a rapid response is a *useful* feature, while in many real-time systems it is an *essential* feature.

Consider, for example, the greatly simplified aircraft autopilot application illustrated schematically in Figure 1.4.

Here, we assume that the pilot has entered the required course heading and that the system must make regular and frequent changes to the rudder, elevator, aileron and engine settings (for example) in order to keep the aircraft following this path.

An important characteristic of this system is the need to process inputs and generate outputs very rapidly, on a time scale measured in milliseconds. In this case, even a slight delay in making changes to the rudder setting (for example) may cause the plane to oscillate very unpleasantly or, in extreme circumstances, even to crash. As a consequence of the need for rapid processing, few software engineers would argue with a claim that the autopilot system is representative of a broad class of real-time systems.

In order to be able to justify the use of the aircraft system in practice, it is not

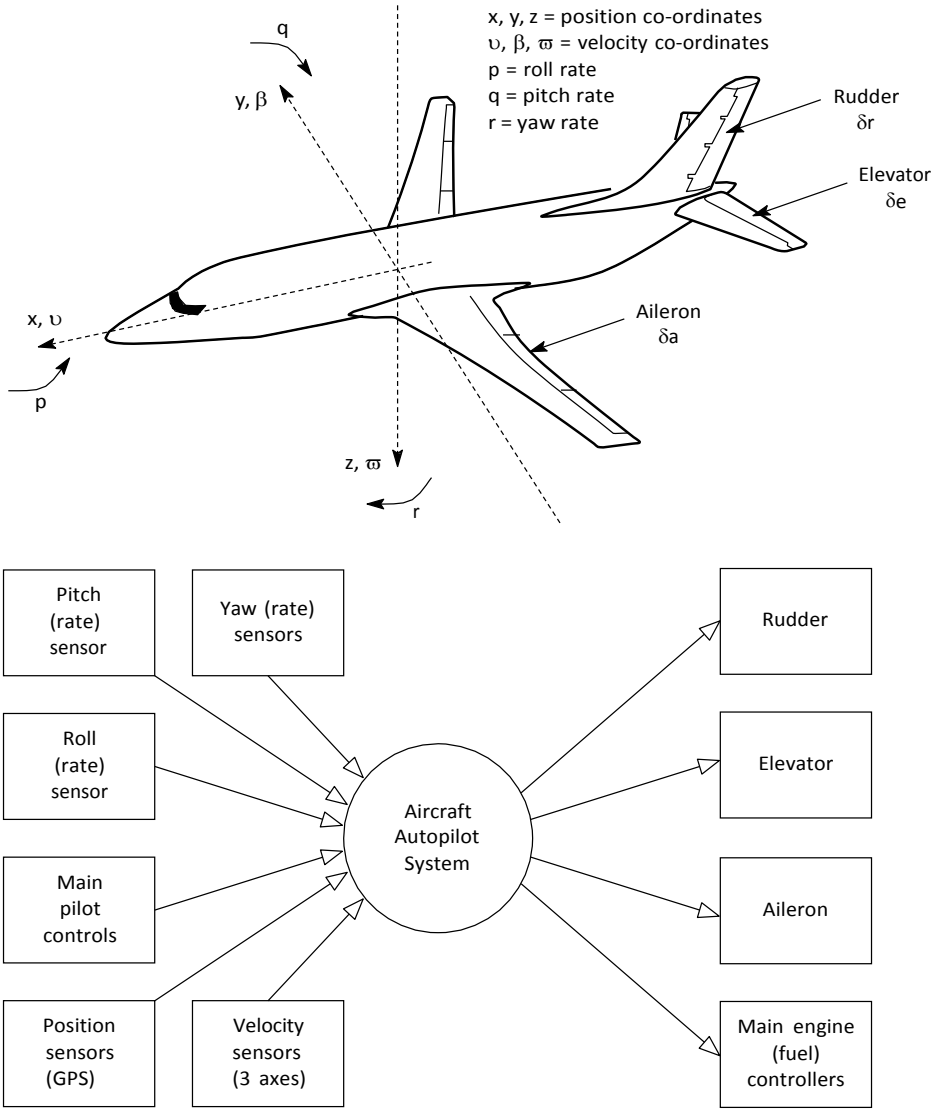


FIGURE 1.4 A high-level schematic view of a simple autopilot system

enough simply to ensure that the processing is 'as fast as we can make it': in this situation, as in many other real-time applications, the key characteristic is *deterministic* processing. What this means is that in many real-time systems we need to be able to *guarantee* that a particular activity will always be completed within (say) 2 ms, or at precisely 6 ms intervals: if the processing does not match this specification, then the application is not simply slower than we would like, it is *useless*.

Tom De Marco has provided a graphic description of this form of hard real-time requirement in practice, quoting the words of a manager on a software project:

'We build systems that reside in a small telemetry computer, equipped with all kinds of sensors to measure electromagnetic fields and changes in temperature, sound and physical disturbance. We analyze these signals and transmit the results back to a remote computer over a wide-band channel. Our computer is at one end of a one-meter long bar and at the other end is a nuclear device. We drop them together down a big hole in the ground and when the device detonates, our computer collects data on the leading edge of the blast. The first two-and-a-quarter milliseconds after detonation are the most interesting. Of course, long before millisecond three, things have gone down hill badly for our little computer. We think of *that* as a real-time constraint.

(De Marco, writing in the foreword to Hatley and Pirbhai, 1987)

In this case, it is clear that this real-time system must complete its recording on time: it has no opportunity for a 'second try'. This is an extreme example of what is sometimes referred to as a 'hard' real-time system.

Note that, unlike this military example, many applications (like the aircraft system outlined earlier), involve repeated sampling of data from the real world (via a transducer and analog-to-digital converter) and, after some (digital) processing, creating an appropriate analog output signal (via a digital-to-analog converter and an actuator). Assuming that we sample the inputs at 1000 Hz then, to qualify as a real-time system, we must be able to process this input and generate the corresponding output, before we are due to take the next sample (0.001 seconds later).

To summarize, consider the following 'dictionary' definition of a real-time system:

'[A] program that responds to events in the world as they happen. For example, an automatic-pilot program in an aircraft must respond instantly in order to correct deviations from its course. Process control, robotics, games, and many military applications are examples of real-time systems.'

(*Hutchinson New Century Encyclopedia* (CD ROM edition, 1996))

It is important to emphasize that a *desire* for rapid processing, either on the part of the designer or on the part of the client for whom the system is being developed, is not enough, on its own, to justify the description 'real time'. This is often misunderstood, even by developers within the software industry. For example, Waites and Knott have stated:

'Some business information systems also require real-time control ... Typical examples include airline booking and some stock control systems where rapid turnover is the norm.'

(Waites and Knott, 1996, p.194)

In fact, neither of these systems can sensibly be described as a real-time application.

1.5 Embedded systems

Although it is widely associated with real-time applications, the category 'embedded systems', like that of desktop systems includes, for example, both real-time and, less

commonly, information systems. The distinguishing characteristic of an embedded application is loosely summarized in the box:

> *An embedded system is an application that contains at least one programmable computer (typically in the form of a microcontroller, a microprocessor or digital signal processor chip) and which is used by individuals who are, in the main, unaware that the system is computer-based.*

Typical examples of such embedded applications include common domestic appliances, such as video recorders, microwave ovens and fridges. Other examples range from cars through combine harvesters to aircraft and numerous defence systems. Please note that this definition *excludes* applications such as 'palm computers' which – from a developer's perspective – are best viewed as a cut-down version of a desktop computer system.

While the desktop market is driven by a need to provide ever more performance, in order to support sophisticated operating systems and applications, the embedded market has rather different needs. For example, recent economic, legislative and technological developments in the automotive sector mean that an increasing number of road vehicles contain embedded systems. In some cases, such systems have been introduced primarily as a means of reducing production costs: for example, in modern vehicles, expensive (~£600.00) multi-wire looms have now been replaced by a two-wire controller area network (CAN) computer bus at a fraction of the cost (Perier and Coen, 1998). In other situations, such as the introduction of active suspension systems, the embedded systems have been introduced to improve ride quality and handling (Sharp, 1998).

Consider a very simple example which may help to illustrate some of the requirements of the embedded market: the indicator light circuit for a passenger car. In this application we need to be able to control six or more indicator lights from a switch behind the steering wheel, allowing the driver to tell other road users that he or she intends to turn a corner, change lane or park. For the US (and some other) markets, we expect the indicator circuit to interact with the rear lights (so that one light flashes to indicate the direction of a turn); in Europe, we expect indicator and rear lights to operate separately. Furthermore, in some countries, we wish to use the indicator lights as 'parking lights', to avoid having people run into our (parked) car at night.

The Volvo 131 ('Amazon') demonstrates the traditional solution to this problem. This classic European car from the 1960s uses a considerable amount of wire and some mechanical switches to provide indicator and braking behaviour: if we wanted to adjust this car to operate in the US style, we would have make substantial changes to the wiring loom. To avoid this type of expensive, labour-intensive conversion, more modern cars use a microcontroller to provide the required behaviour. Not only does the microcontroller solution result in a simpler, and cheaper, collection of wires, it can also be converted between US and European indicator standards by flicking a switch or changing a memory chip.

This simple application highlights four important features of many embedded applications.

- Like this indicator system, many applications employ microcontrollers not because the processing is complex, but because the microcontroller is flexible and, crucially, because it results in a cost-effective solution. As a result, many products have embedded microcontrollers sitting almost idle for much of their operational life. The fact that many commonly used microcontrollers are, by comparison with modern desktop microprocessors, often rather slow, is seldom of great concern.
- Unlike most microprocessors, microcontrollers are required to interact with the outside world, not via keyboards and graphical user interfaces, but via switches, small keypads, LEDs and so on. The provision of extensive I/O facilities is a key driving force for many microcontroller manufacturers.
- Like the indicator system, most embedded applications are required to execute particular tasks at precise time intervals or at particular instants of time. In this case, for example, the indicators lights must flash on at a precise frequency and duty cycle in order to satisfy legal requirements. This type of application is considered in greater detail in Chapter 2.
- Unlike many desktop applications (for example) many embedded applications have safety implications. For example, if the indicator lights fail while the car is in use, this could result in an accident. As a result, reliability is a crucial requirement in many embedded applications.

1.6 Event-triggered systems

Many applications are now described as 'event triggered' or 'event driven'. For example, in the case of modern desktop applications, the various running applications must respond to events such as mouse clicks or mouse movements. A key expectation of users is that such events will invoke an 'immediate' response.

In embedded systems, event-triggered behaviour is often achieved through the use of interrupts (see following box). To support these, event-triggered system architectures often provide multiple interrupt service routines.

What is an interrupt?

From a low-level perspective, an interrupt is a hardware mechanism used to notify a processor that an 'event' has taken place: such events may be 'internal' events (such as the overflow of a timer) or 'external' events (such as the arrival of a character through a serial interface).

Viewed from a high-level perspective, interrupts provide a mechanism for creating multitasking applications: that is applications which, apparently, perform more than one

task at a time using a single processor. To illustrate this, a schematic representation of interrupt handling in an embedded system is given in Figure 1.5.

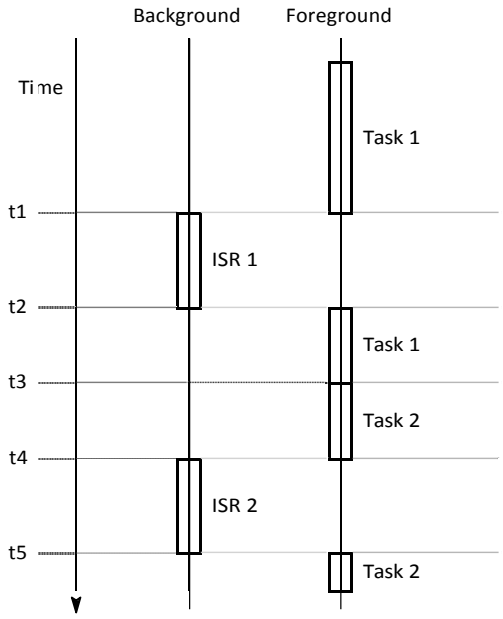


FIGURE 1.5 A schematic representation of interrupt handling in an embedded system

In Figure 1.5 the system executes two (background) tasks, Task 1 and Task 2. During the execution of Task 1, an interrupt is raised, and an 'interrupt service routine' (ISR1) deals with this event. During the execution of Task 2, another interrupt is raised, this time dealt with by ISR2.

Note that, from the perspective of the programmer, an ISR is simply a function that is 'called by the microcontroller', as a result of a particular hardware event.

1.7 Time-triggered systems

The main alternative to event-triggered systems architectures are time-triggered architectures (see, for example, Kopetz, 1997). As with event-triggered architectures, time-triggered approaches are used in both desktop systems and in embedded systems.

To understand the difference between the two approaches, consider that a hospital doctor must look after the needs of ten seriously ill patients overnight, with the support of some nursing staff. The doctor might consider two ways of performing this task:

- The doctor might arrange for one of the nursing staff to waken her, if there is a significant problem with one of the patients. This is the 'event-triggered' solution.

- The doctor might set her alarm clock to ring every hour. When the alarm goes off, she will get up and visit each of the patients, in turn, to check that they are well and, if necessary, prescribe treatment. This is the 'time-triggered' solution.

For most doctors, the event-triggered approach will seem the more attractive, because they are likely to get a few hours of sleep during the course of the night. By contrast, with the time-triggered approach, the doctor will inevitably suffer sleep deprivation.

However, in the case of many embedded systems – which do not need sleep – the time-triggered approach has many advantages. Indeed, within industrial sectors where safety is an obvious concern, such as the aerospace industry and, increasingly, the automotive industry, time-triggered techniques are widely used because it is accepted, both by the system developers and their certification authorities, that they help improve reliability and safety (see, for example, Allworth, 1981; MISRA, 1994; Storey, 1996; Nissanke, 1997; Bates, 2000 for discussion of these issues).

The main reason that time-triggered approaches are preferred in safety-related applications is that they result in systems which have very *predictable* behaviour. If we revisit the hospital analogy, we can begin to see why this is so.

Suppose, for example, that our 'event-triggered' doctor is sleeping peacefully. An apparently minor problem develops with one of the patients and the nursing staff decide not to awaken the doctor but to deal with the problem themselves. After another two hours, when four patients have 'minor' problems, the nurses decide that they will have to wake the doctor after all. As soon as the doctor sees the patients, she recognizes that two of them have a severe complications, and she has to begin surgery. Before she can complete the surgery on the first patient, the second patient is very close to death.

Consider the same example with the 'time-triggered' doctor. In this case, because the patient visits take place at hourly intervals, the doctor sees each patient before serious complications arise and arranges appropriate treatment. Another way of viewing this is that the workload is spread out evenly throughout the night. As a result, all the patients survive the night without difficulty.

In embedded applications, the (rather macabre) hospital situation is mirrored in the event-driven application by the occurrence of several events (that is, several interrupts) at the same time. This might indicate, for example, that two different faults had been detected simultaneously in an aircraft or simply that two switches had been pressed at the same time on a keypad.

To see why the simultaneous occurrence of two interrupts causes a problem, consider what happens in the 8051 architecture in these circumstances. Like many microcontrollers, the original 8051 architecture supports two different interrupt priority levels: low and high. If two interrupts (we will call them Interrupt 1 and Interrupt 2) occur in rapid succession, the system will behave as follows:

- If Interrupt 1 is a low-priority interrupt and Interrupt 2 is a high-priority interrupt:
  The interrupt service routine (ISR) invoked by a low-priority interrupt can be interrupted by a high-priority interrupt. In this case, the low-priority ISR will be paused, to allow the high-priority ISR to be executed, after which the operation of the low-priority ISR will be

completed. In most cases, the system will operate correctly (provided that the two ISRs do not interfere with one another).

- If Interrupt 1 is a low-priority interrupt and Interrupt 2 is also a low-priority interrupt:

  The ISR invoked by a low-priority interrupt cannot be interrupted by another low-priority interrupt. As a result, the response to the second interrupt will be at the very least delayed; under some circumstances it will be ignored altogether.

- If Interrupt 1 is a high-priority interrupt and Interrupt 2 is a low-priority interrupt:

  The interrupt service routine (ISR) invoked by a high-priority interrupt cannot be interrupted by a low-priority interrupt. As a result, the response to the second interrupt will be at the very least delayed; under some circumstances it will be ignored altogether.

- If Interrupt 1 is a high-priority interrupt and Interrupt 2 is also a high-priority interrupt:

  The interrupt service routine (ISR) invoked by a high-priority interrupt cannot be interrupted by another high-priority interrupt. As a result, the response to the second interrupt will be at the very least delayed; under some circumstances it will be ignored altogether.

Note carefully what this means! There is a common misconception among the developers of embedded applications that interrupt events will never be lost. This simply is not true. If you have multiple sources of interrupts that may appear at 'random' time intervals, interrupt responses can be missed: indeed, where there are several active interrupt sources, it is practically impossible to create code that will deal correctly with all possible combinations of interrupts.

It is the need to deal with the simultaneous occurrence of more than one event that both adds to the system complexity and reduces the ability to predict the behaviour of an event-triggered system under all circumstances. By contrast, in a time-triggered embedded application, the designer is able to ensure that only single events must be handled at a time, in a carefully controlled sequence.

As already mentioned, the predictable nature of time-triggered applications makes this approach the usual choice in safety-related applications, where reliability is a crucial design requirement. However, the need for reliability is not restricted to systems such as fly-by-wire aircraft and drive-by-wire passenger cars: even at the lowest level, an alarm clock that fails to sound on time or a video recorder that operates intermittently, or a data monitoring system that – once a year – loses a few bytes of data may not have safety implications but, equally, will not have high sales figures.

In addition to increasing reliability, the use of time-triggered techniques can help to reduce both CPU loads and memory usage: as a result, as we demonstrate throughout this book, even the smallest of embedded applications can benefit from the use of this form of system architecture.

chapter 4

Oscillator hardware

Introduction

All digital computer systems are driven by some form of oscillator circuit. This circuit is the 'heartbeat' of the system and is crucial to correct operation. For example, if the oscillator fails, the system will not function at all; if the oscillator runs irregularly, any timing calculations performed by the system will be inaccurate.

As a consequence, choice of an appropriate oscillator circuit is an important part of any hardware design. For most microcontroller-based systems, there are two main oscillator options, each of which is represented by a pattern in this chapter:

CRYSTAL OSCILLATOR

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

When and how should you use a quartz crystal to create an oscillator for use with members of the 8051 family of microcontrollers?

Background

Quartz is a common mineral and is the main component of most sand grains. It has the useful quality that it is piezoelectric in nature, which means that if we apply pressure to a piece of quartz, it will generate an electric current at a particular frequency. In some materials, the converse is also true: application of an electric field will cause a mechanical deflection in the material.

We can use this behaviour as the basis of a useful oscillator by using an electric field (generated by plating some contacts on the surface of the mineral and applying a current) to set up mechanical oscillations in the crystal which are, in turn, converted into measurable voltage fluctuations at the surface of the crystal. We can precisely control the frequency of these fluctuations by cutting the quartz to a particular size and shape: a particular form of cut, known as the 'AT' cut, is reasonably inexpensive to produce and can create high-frequency crystals with good temperature stability at reasonable cost.

To create a complete oscillator, some further components are required. Figure 4.1 shows how crystals may be used to generate a popular form of oscillator circuit known as a Pierce oscillator.

A variant of the Pierce oscillator is common in the 8051 family. To create such an oscillator, most of the components are included on the microcontroller itself: these components are, together, sometimes referred to as the oscillator inverter. The user of this device must generally only supply the crystal and two small capacitors to complete the oscillator implementation. We discuss this further in the solution section of this pattern.

Note that, in some circumstances, it may be preferable to use a complete, self-contained external crystal oscillator module (based on a circuit like that illustrated in Figure 4.1) and use this to drive the microcontroller. We discuss this possibility in 'Reliability and safety implications'.

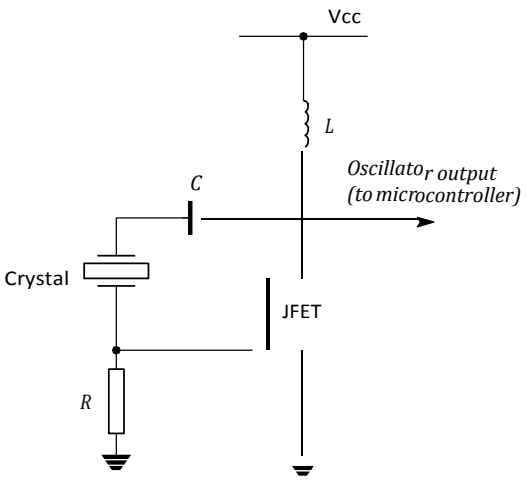


FIGURE 4.1 A Pierce oscillator circuit, driven by a quartz crystal (adapted from Horowitz and Hill, 1989)

The link between oscillator frequency and machine cycle period

When selecting an appropriate oscillator for an 8051–family device, the choice of oscillator frequency is really incidental to our real concern: the machine cycle period. That is, we are concerned with the speed at which instructions will execute.

As we discussed in Chapter 3, the various members of the 8051 family have different relationships between the oscillator cycle period and the machine cycle period. For example, in the original members of the 8051 family, the machine cycle takes 12 oscillator periods. In later family members, such as the Infineon C515C, a machine cycle takes six oscillator periods; in more recent devices such as the Dallas 89C420, only one oscillator period is required per machine cycle. As a result, the later members of the family operating at the same clock frequency execute instructions much more rapidly.

In general, the improved performance of modern implementations of the 8051 is 'A Good Thing': however, in situations where timing is critical, care must be taken to ensure that any timer-related calculations are implemented correctly on a particular device: see **HARDWARE DELAY** [page 194], and **CO-OPERATIVE SCHEDULER** [page 255] for further details.

Reliability and safety implications

We consider some reliability and safety issues related to the use of crystal oscillators in this section.

System heartbeat

The oscillator forms the 'heartbeat' of any digital computer. If this heartbeat stops, your system will stop. If this heartbeat varies, timing loops, delays, generated waveforms etc. will vary too. Correct operation of your embedded system relies therefore on the provision of a robust and regular clock input.

Heart of glass

Quartz is similar to glass in some physical characteristics: in particular, it is fragile.

If you require an oscillator that will operate in an environment where there is significant vibration, then quartz may not be the ideal choice. If you use a quartz crystal in these circumstances, you will need to package your application to avoid vibration influencing the operation of your system.

Time taken for oscillator to start

If the start of the (crystal) oscillator in your circuit is delayed, then the reset cycle may be completed before the oscillation begins. If this happens, the chip will not be reset.[9]

The time taken for a crystal oscillator to start operating depends on its being mounted correctly and having appropriate capacitors. Typical start-up times are 0.1 to 10 ms (Mariutti, 1999).

Using an external crystal oscillator module

As we noted in 'Background', it is possible to use a self-contained external crystal oscillator module (based on a circuit like that illustrated in Figure 4.1) to drive the microcontroller. This technique has the considerable advantage that the oscillator is guaranteed to start. This can make it a good solution if your system must operate very reliably.

Connecting an oscillator module is very straightforward. Figure 4.3 shows a circuit that will work with all members of the 8051 family. Note that, as shown in the figure, pin XTAL1 should be driven, while XTAL2 is left unconnected.

Particularly where higher clock frequencies (> 12 MHz) are being used, then modules may improve your system reliability. However, oscillator modules do have several drawbacks:

- Oscillator modules cost around twice the price of a crystal oscillator and four times as much as a ceramic resonator.
- Oscillator modules typically draw currents comparable to that of an 8051 microcontroller: 15–35 mA. This may represent a very significant power drain in battery-powered applications.

---

9. See **RC RESET** [page 68] for further details.

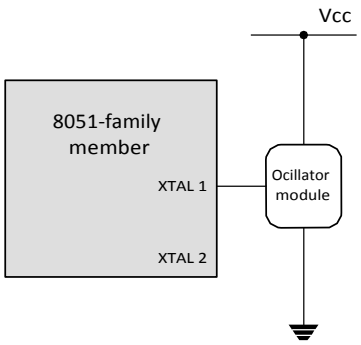


FIGURE 4.3 Using an external oscillator module

- Oscillator modules are not always easy to obtain in 'odd' frequencies, such as 11.059 MHz. This frequency is very useful in 8051-based designs involving a serial interface, as discussed in 'Related patterns and alternative solutions'.

Overall strengths and weaknesses

- ☺ **Crystal oscillators are stable. Typically ±20–100 ppm = ±50 mins per year (up to ~1 minute / week).**
- ☺ **The great majority of 8051-based designs use a variant of the simple crystal-based oscillator circuit presented here: developers are therefore familiar with crystal-based designs.**
- ☺ **Quartz crystals are available at reasonable cost for most common frequencies. The only additional components required are usually two small capacitors. Overall, crystal oscillators are more expensive than ceramic resonators.**
- ☹ Crystal oscillators are susceptible to vibration.
- ☹ The stability falls with age.

chapter 5

Reset hardware

Introduction

The process of starting any microcontroller is a non-trivial one. The underlying hardware is complex and a small, manufacturer-defined 'reset routine' must be run to place this hardware into an appropriate state before it can begin executing the user program. Running this reset routine takes time and requires that the microcontroller's oscillator is operating.

Where your system is supplied by a robust power supply, which rapidly reaches its specified output voltage when switched on, rapidly decreases to 0V when switched off, and – while switched on – cannot 'brown out' (drop in voltage), then you can safely use low-cost reset hardware based on a capacitor and a resistor: this form of reset circuit is addressed in **RC RESET** [page 68].

Where your power supply is less than perfect, and / or your application is safety related, the simple RC solution will not be suitable. **ROBUST RESET** [page 77] discusses a more reliable alternative.

RC RESET

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you create a low-cost reset circuit for your 8051 microcontroller?

Background

As discussed in the introduction to this chapter, the reset process which must be completed prior to the execution of any other code requires that the microcontroller's oscillator is operating. To trigger the reset operation, the original members of the 8051 family have a 'RESET' pin. When this is held at Logic 0, the chip will run normally. If, while the oscillator is running, this pin is held at Logic 1 for two (or more) machine cycles, the microcontroller will be reset.

Note that, if the reset operation is not completed correctly, the microcontroller will usually not operate at all: in rare circumstances, it may operate, but incorrectly. In either event, there is usually nothing that you can do, in software, to recover control of the system. Clearly, therefore, ensuring correct reset operation is a crucial part of any application.

Solution

Various techniques may be used to ensure that – when power is applied to your 8051-based application – the reset process is automatically carried out. The most widely used techniques are based on the use of an external capacitor and resistor: these techniques are considered in detail here.

RC reset circuits

A typical RC reset circuit is as shown in Figure 5.1.

The circuit in Figure 5.1 operates as follows. We assume that Vcc is initially at 0V (that is, the power has not been applied to the system) and that the capacitor C is fully discharged. When power is applied, the capacitor will begin to charge. Initially, the voltage across the capacitor will be 0V and – therefore – the voltage across the resistor (and the voltage at the RESET pin) will be Vcc: this is a Logic 1 value. Gradually, the capacitor will charge and its voltage will rise, eventually to Vcc: at this time, the voltage at the reset pin will be 0V.

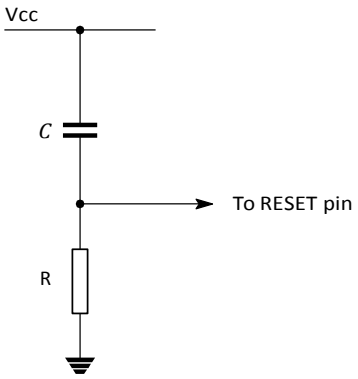


FIGURE 5.1 An (active high) RC reset circuit

In the real system, the microcontroller's input voltage threshold is around 1.1 – 1.3V[10]: input voltages below this level are interpreted as Logic 0 and voltages above this level are interpreted as Logic 1. Thus, the reset operation will continue until the voltage at the RESET pin falls to a level of around 1.2V.

We can use this information to calculate the required values of R and C. To make this calculation, we use the fact that the capacitor in Figure 5.1 will have a voltage (*Vcap*) at time (*t*) seconds after it begins charging, given by Equation 5.1.

$$V_{cap} = V_{cc}(1 - e^{-t/RC})$$

EQUATION 5.1 The voltage across the capacitor in Figure 5.1 as a function of time

Note that Equation 5.1 assumes that the capacitor begins charging at a voltage of 0 and that the power supply voltage increases from 0V to Vcc in an instantaneous 'step' (rather than a slow ramp): these assumptions, although often made, are frequently invalid: see 'Safety and reliability issues' for a discussion of these issues.

The Intel 8051 data sheet recommends values of 8.2K for R and 10uf for C when this form of reset circuit is used. Figure 5.2 substitutes these values into Equation 5.1 and plots the result over a period of 500 ms.

When looking at Figure 5.2, remember that all 8051s complete their reset operation in 24 oscillator periods or less: if we use a 12 MHz oscillator, this is a maximum period of 0.002 ms: by contrast, the recommended reset circuit takes around 100 ms to complete the reset operation. This may seem like an excessive reset period

10. The data sheet for your chosen microcontroller will provide a precise value, if you require it.

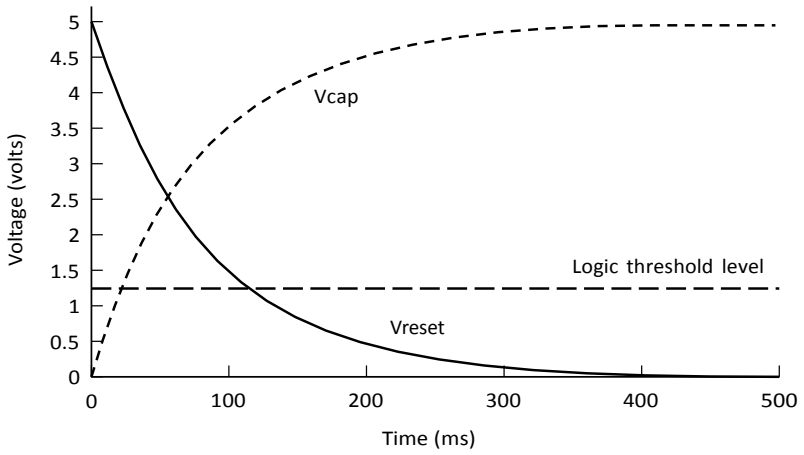


FIGURE 5.2 An example of the behaviour of an RC reset circuit using standard component values and an ideal power supply

but, for reasons discussed under 'Safety and reliability issues', allowing approximately 100 ms for the reset is generally good practice.

Choosing values of R and C

If, having reviewed all aspects of this pattern, you have decided to use an RC-based reset circuit, what values of R and C should you use?

Rather than trying to determine values of R and C directly from Equation 5.1, we can simplify matters by noting that the product of R (in Ohms) multiplied by C (in Farads) is known as the 'time constant' (in seconds) of this form of RC circuit. This time constant is the time taken for the capacitor to be charged to 60% of its final voltage. Thus, with a 5V supply and the circuit in Figure 5.1, this is the time taken for the capacitor voltage to reach 3V and, therefore, the voltage at the reset pin to reach 2V (that is, Vcc – 3V): this is still high enough (because it is greater than 1.2V, as already discussed) to ensure that the device is in reset mode. As long as the device is still in this mode until approximately 1 ms after the power supply reaches Vcc (typically around 100 ms after starting: see 'Safety and reliability issues'), the device will be reset correctly.

A basic rule of thumb, therefore, is that the RC time constant should be approximately 100 ms and values of R and C chosen to meet this requirement will usually ensure effective reset operation (Equation 5.2):

$$RC \geq 100 \text{ ms}$$

EQUATION 5.2 A 'rule of thumb' for calculating appropriate RC values

A suitable RC reset circuit satisfying these conditions is shown in Figure 5.3.

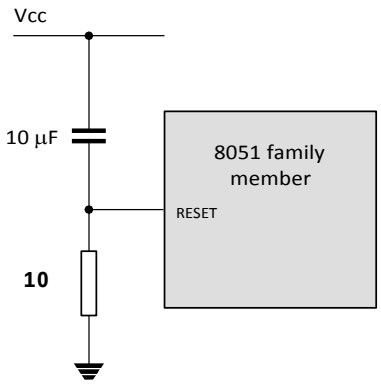


FIGURE 5.3 A suitable (active high) RC reset circuit

We can summarize the key material in this section as follows:

- A combination of a 10K resistor and a 10 μF capacitor in a RC reset circuit gives a 100 ms time constant. Bearing in mind the general limitations of RC reset circuits (see 'Safety and reliability issues'), this value is suitable for the majority of 8051-based systems.
- The standard 8K2, 10 μF RC reset combination gives a time constant of 82 ms: this is generally adequate.
- Values of 1K and 10 μF (which appear in some books) provide a time constant of only 10 ms: these values will not provide a reliable reset operation with all power supplies.

Adding a RESET button

In some systems, it is helpful to have a reset button, to force a hardware reset. This is easy to achieve. Figure 5.4 shows a suitable circuit.

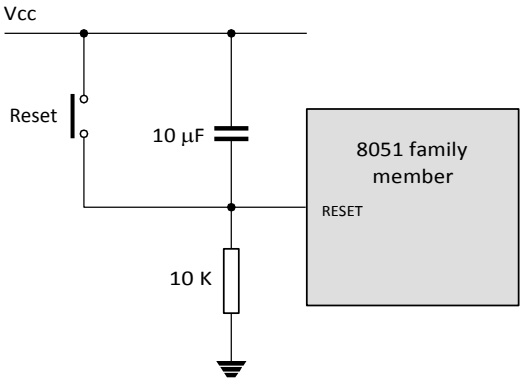


FIGURE 5.4 A reset circuit (active high) with reset switch

Note that the reset button pulls the RESET pin (assumed to be active high: see 'Portability') to Vcc. Note also that this button also discharges the capacitor, ensuring that – when the switch is released – the proper reset process will be carried out.

chapter 6

Memory issues

Introduction

All practical microcontroller-based systems require some form of non-volatile code memory (to store the program code) and some form of volatile memory (to store data and the stack).

In many cases, it is possible to create useful applications without adding external memory devices. The first pattern in this chapter (**ON-CHIP MEMORY** [page 82]) discusses how to do this by making effective use of the various memory areas available in members of the 8051 family.

In some applications, it is necessary to add external memory: the remaining patterns in this chapter (**OFF-CHIP DATA MEMORY** [page 94] and **OFF-CHIP CODE MEMORY** [page 100]) consider how best to add additional memory to your 8051-based application.

Please note that the material in this chapter is concerned primarily with devices using the Standard 8051 memory architecture. Some of the more recent 8051 devices, such as the Dallas 80C390, Analog Devices ADμC812 and Philips 80C51MX, provide support for much larger amounts of external memory than was possible in the original 8051 device: we briefly consider such extended memory devices in this chapter, but – as each manufacturer has an individual solution – we do not attempt to cover them in detail.

ON-CHIP MEMORY

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you create an 8051-based circuit that uses only internal memory?

Background

Some general background material on memory is presented in this section.

Direct vs. indirect addressing

You will often see the terms 'indirect addressing' and 'direct addressing' used in discussions about microcontroller memory. Although these terms often cause confusion, they are not difficult to understand.

You will recall that whatever language you write in (for example, C or assembly), your code must ultimately be translated into machine code instructions that can be executed by your chosen microcontroller. This set of possible machine instructions is defined by the hardware manufacturer. Through this process, even complex statements in a high-level language are eventually broken down into basic operations, such as 'Copy this piece of data from one memory location to another'. These, in turn, are implemented by machine instructions that take the form 'Move the contents of memory address X to register Y'.

There are essentially two ways in which such fundamental 'Move' instructions may be implemented in microcontrollers and microprocessors:

- Using **direct addressing**, the address of the memory location (that is, memory address X in the last example) is specifically given as part of the instruction.
- Using **indirect addressing**, the address of the memory location is not explicitly included as part of the instruction: instead the address (of another memory location or another register) that contains memory address X is included in the instruction.

Since the use of indirect addressing means that two steps are required to find the address of the required memory location it may appear to be slower than direct addressing. However, universal use of direct addressing in an 8-bit architecture (with a 16-bit address space) would mean that all 'Move' instructions would need to include two bytes of address information, and would therefore take more time to fetch from

memory. A compromise is thus often made in devices (including the 8051) where a small area of memory can be directly addressed and most other memory areas must be indirectly addressed.

Note that the distinction between direct and indirect addressing also has other uses. For example, within members of the 8051 family, there is an area of 'special function register' memory and another area of general purpose memory. Both blocks of memory are the same size (128 bytes) and both blocks share the same address range. On the surface, having two areas of memory with the same address makes no sense; however, in this case, there is no problem. One block of memory can only be accessed indirectly, the other block can only be accessed directly. As a result, when our compiler translates a particular 'C' statement, appropriate machine-level instructions are selected to ensure that the correct memory area is accessed: in most circumstances, this process is completely hidden from the programmer.

Types of memory

On the desktop, most designers and programmers can safely ignore the type of memory they are using. This is seldom the case in embedded environments and we therefore briefly review some of the different types of memory.

First, a short history lesson, to explain the roots of an important acronym. On early mainframe and desktop computer systems, long-term data storage was carried out using computer tapes. Reading or writing to the tape took varying amounts of time, depending whether it involved, for example, rewinding the entire tape or simply rewinding a couple of centimetres. In this context, new memory devices appeared that could be used to store data while the computer was running, but which lost these data when the power was removed. These read-write memory devices were referred to as 'random access memory' (RAM) devices, because – unlike tape-based systems – accessing any element of memory 'chosen at random' took the same amount of time.

Tapes have now largely disappeared, but the acronym RAM has not and is still used to refer to memory devices that can be both read from and written to. However, since RAM was first introduced, new forms of memory devices have appeared, including various forms of ROM (read-only memory). Since these ROM devices are also 'random access' in nature, the acronym RAM is now best translated as 'read-write memory'.

Dynamic RAM (DRAM)

Dynamic RAM is a read-write memory technology that uses a small capacitor to store information. As the capacitor will discharge quite rapidly, it must be frequently refreshed to maintain the required information: circuitry on the chip takes care of this refresh activity. Like most current forms of RAM, the information is lost when power is removed from the chip.

In general, dynamic RAM is simple and comparatively cheap.

Static RAM (SRAM)

Static RAM is a read-write memory technology that uses a form of electronic flip-flop to store the information. No refreshing is required, but the circuitry is more complex and costs can be several times that of the corresponding size of DRAM. However, access times may be a third those of DRAM.

Mask read-only memory (ROM)

A true read-only memory (ROM) would be useless. The most basic kind of practical ROM is, from the designer's perspective, read only: however, the manufacturer is able to write to the memory, at the time the chip is manufactured, according to a 'mask' provided by the company for which the chips are being produced. Such devices are therefore sometimes referred to as 'factory-programmed ROM' or 'mask ROM'. Factory or mask programming is not cheap and is not a low-volume option: as a result, mistakes can be very expensive and providing code for your first mask can be a character-building process. Access times are often slower than RAM: roughly 1.5 times that of DRAM.

It should be noted that mask ROMs retain their contents even in environments with high levels of electromagnetic activity. This behaviour is in contrast to some of the erasable devices in which there is a risk that data corruption may occur due, for example, to high UV levels (UV–EPROMs: see later in this section) or strong electrical fields (EEPROMs: see later in this section).

Many members of the 8051 family are available with on-board mask-programmable ROM.

Programmable read-only memory (PROM)

The name PROM sounds like a contradiction and it is. This is, in fact, a form of write-once, read-many (WORM) or 'one-time programmable' (OTP) memory. Basically, we use a PROM programmer to blow tiny 'fuses' in the device. Once blown, these fuses cannot be repaired; however, the devices themselves are cheap.

Many modern members of the 8051 family are available with OTP ROM.

UV-erasable programmable read-only memory (UV-EPROM)

Like PROMs, UV-EPROMs are programmed electrically. Unlike PROMs, they also have a quartz window which allows the memory to be erased by exposing the internals of the device to UV light. The erasure process can take several minutes and, after erasure, the quartz window will be covered with a UV-opaque label. This form of EPROM can withstand thousands of program / erase cycles.

More flexible than PROMs and once very common, UV-EPROMs now seem rather primitive compared with EEPROMs. They can be useful for prototyping but are prohibitively expensive for use in production.

Many older members of the 8051 family are available with on-board UV-EPROM.

Electrically erasable programmable read-only memory (EEPROM, E^2PROM)

EEPROMs are a more user-friendly form of EPROM that can be both programmed and erased electrically. This does not mean they can simply be used in place of RAM for all purposes, not least because writing to the EEPROM is a very slow process and there is a limit to the number of write operations that may be performed.

Many members of the 8051 family are available with on-board EEPROM.

Flash ROM

Flash ROM is not only a relief from increasingly long and irrelevent acronyms, it is also the most civilized form of ROM currently available. As the name suggests, it can usually be programmed much more rapidly than an EEPROM. In addition, while many EEPROMs often require high (12V) programming voltages, Flash ROM devices can usually be programmed at standard (3V/5V) levels.

Members of the 8051 family are now available with on-board flash ROM.

chapter 7

Driving DC Loads

Introduction

The port pins on a typical microcontroller can be set at values of either 0V or 5V (or, in a 3V system, 0V and 3V) under software control. Each pin can typically sink (or source) a current of around 10 mA. In this chapter, we are concerned with hardware designs that will allow us to control a range of low- and high-power, direct current (DC) loads via these pins.

With care, the port may be used directly to drive low-power DC loads, such as LEDs (see **NAKED LED** [page 110]) or, for example, small warning buzzers (see **NAKED LOAD** [page 115]). However, while a limited number of such loads may be connected directly to the port, connecting, say, eight LEDs will generally exceed the total port or microcontroller capacity. In these circumstances, use of an IC-based buffer circuit can be a cost-effective solution: see **IC BUFFER** [page 118]. Indeed, even with small loads, the reliability of the application may be improved through the use of such a buffer.

Of course, many output devices require a higher voltage and a far greater power output than the naked ports can provide. For example, to drive even a small DC motor may require up to 1A of current at 12V. To control such devices, we need to provide appropriate driver (or interface) circuit to convert the microcontroller output to an appropriate level. We will consider three ways of driving such loads in this chapter:

- Using bipolar-junction transistors (see **BJT DRIVER** [page 124])
- Using a driver IC (see **IC DRIVER** [page 134])
- Using 'metal oxide silicon field effect transistors' (see **MOSFET DRIVER** [page 139])

Please note that in this chapter our concern will be with hardware details of the interface only. In **PORT I/O** [page 174] we consider software suitable for controlling such hardware.

NAKED LED

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

What is the cheapest way of driving a small number of LEDs with a microcontroller?

Background

Even in the most basic application, the presence of a constant red or green output from a light-emitting diode (LED) is reassuring, implying that the application is at least powered. In other applications requiring numerical outputs, LEDs are grouped into seven- or eight-segment combinations[12] to display, for example, the current time, the voltage or the number of calls received by a telephone answering machine ('voice mail') system. Overall, LEDs are the most widely used components in user interfaces for embedded systems.

Given the name it is hardly surprising that, electrically, an individual LED operates as a diode. LEDs have a forward voltage drop of about 2V, and typically require a current of around 5–15 mA for a bright display (Figure 7.1). Note that the forward voltage required to 'switch on' a conventional (silicon) diode is around 0.7 V: the difference arises because LEDs are generally manufactured from gallium arsenide phosphide (see Horowitz and Hill, 1989, for further details).

Solution

The focus in this pattern is on low-cost techniques for driving LEDs.

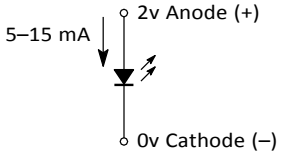


Figure 7.1 Lighting a single LED

12. Multi-segment LEDs are considered in **MX LED DISPLAY** [page 450].

Driving single LEDs

It is possible to drive small single LEDs directly from the port pins, as illustrated in Figure 7.2. Note that we usually need a resistor in series with the LED, between the supply and the port pin. This resistor is required to limit the current flow to the port when the LED is 'switched on'.

To understand why this resistor is necessary, you need to remember that – as we discussed in 'Background' – the voltage drop across the LED will be around 2V. Assuming Vcc is 5V, then the voltage drop across the resistor and diode, when the port pin is at 0V, will be around 5V. Thus, the voltage drop across the resistor needs to be 3V. If there is no resistor, then we need to drop 3V across a stretch of wire: this can cause a very strong current to flow (limited only by the supply capacity), which will almost instantly destroy the port, the LED and, possibly, the whole microcontroller.

The equation in Figure 7.2 (essentially Ohm's law) shows how to calculate the required resistor (R) value. Typical values of the required parameters are as follows:

- Supply voltage, $V_{cc} = 5V$
- LED forward voltage, $V_{diode} = 2V$
- Required diode current, I_{diode} = 10 mA (note that the data sheet for your chosen LED will provide this information)

This gives a required R value of 300Ω.

Use of pull-up resistors

Throughout this chapter, we will present examples of driver circuits like that shown in Figure 7.2.

This circuit will only work on port pins where the microcontroller has an internal pull-up resistor: this applies to most ports on the 8051 family, with the exception of Port 0. In addition, some other members of the 8051 family – notably the Atmel 89Cx051 devices – have small numbers of pins without internal pull-ups.

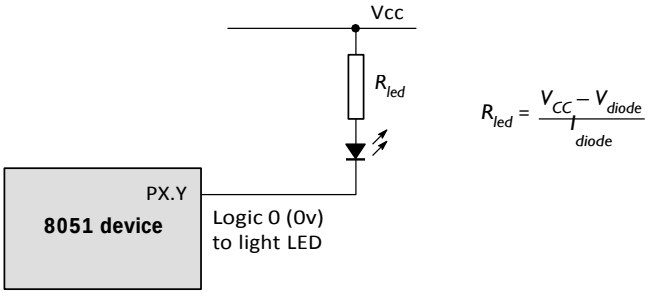


Figure 7.2 Connecting a single LED directly to a microcomputer port

[*Note*: When calculating the required value of R_{led}, the resistance values are in ohms, voltages in volts and current in amps.]

To adapt circuits such as that shown in Figure 7.2 for use on pins without internal pull-up resistors is straightforward: you simply need to add an external pull-up resistor, as illustrated in Figure 7.3. The value of the pull-up resistor should be between 1K and 10K. This requirement applies to all of the examples in this book.

In the unlikely event that you do not know whether a drive circuit will be used on a port with pull-up resistors, the best solution is to include 10K resistors in your circuit. If the port pin used already contains pull-ups, the extra resistor will have no discernible impact on the operation of the circuit.

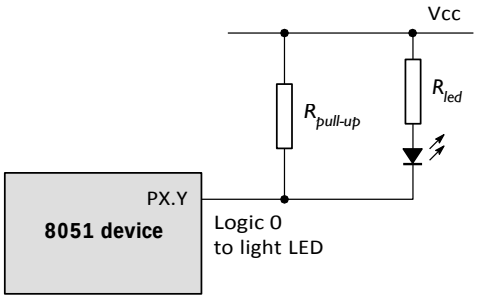


FIGURE 7.3 Using a pull-up resistor

Hardware resource implications

Every implementation of this pattern uses at least one port pin.

Reliability and safety implications

There are several reliability implications involved in the use of this pattern.

Connecting up LEDs

If you connect ordinary LEDs to a port, do not omit the resistor! The resulting high current flows may not cause the system to fail immediately, but will greatly reduce the life of the port pin and, potentially, the whole processor.

Note: There are '5V' LEDs available, at higher cost: these include series resistors. Provided they have suitable current requirements, they may be safely connected directly to the port pins.

Should you use a buffer?

Where more than two LEDs are connected to a single port, buffering is almost always essential, because – while the limit for an individual port pin may be (say) 10 mA – the port as a whole may have a limit of 20 mA or less. This issue is discussed in **IC BUFFER** [page 118].

Use of LEDs as warning devices

LEDs (particularly flashing LEDs) are frequently used as warning devices. Bear in mind that in bright sunlight, such warnings will be barely visible and that blind or partially sighted people will never be able to see them. Adding an additional or alternative audible output may be appropriate in some systems.

General use of LEDs

LEDs consume large amounts of power (compared, for example, to liquid crystal displays) and need to be used with care in many battery-powered designs.

Portability

The circuits presented here can be used with almost all microcontrollers and microprocessors. Please note, however, that most of the circuits we present for driving DC and AC loads involve some form of current 'sink', as in Figure 7.2. Here, the current flows 'in to' the processor port. This is despite the fact that most microcontroller ports, manufactured using some form of MOS technology, can also 'source' current, so that the load could be arranged as illustrated in Figure 7.4.

However, some of the other circuits used as buffers or drivers may use different manufacturing processes, including TTL technology. TTL devices can sink current in much the same way as MOS devices, but are poor current sources. As a result, hardware designs based on current sinking are generally more portable (across different logic families) than designs based on current sourcing and, for this reason, will be used throughout this pattern collection.

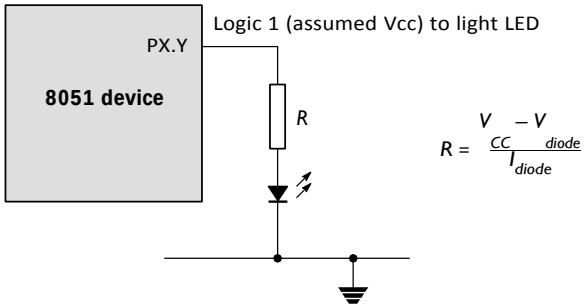


FIGURE 7.4 Connecting a single LED directly to a processor port

[*Note*: Here the port acts as a current source. See text for details and for a discussion of the drawbacks of this approach.]

Overall strengths and weaknesses

☺ **This pattern allows small numbers of LEDs to be driven from a microcontroller port with a minimum of external hardware.**

☺ **This is a low-cost solution.**

☹ Only applicable for small numbers of LEDs (typically two per port), otherwise a buffer will be required: see **IC BUFFER** [page 118].

Related patterns and alternative solutions

See **PORT I/O** [page 174] for software details.

See the remaining patterns in this chapter for techniques suitable for driving higher powered loads.

Example: Using low-current LEDs

To emphasize that all rules can be broken, we will consider in this example how you can connect eight 'naked' LEDs to the port of an 8051 microcontroller without using a buffer (see Figure 7.5).

Of course, for the reasons outlined in this pattern, it is not possible to connect up 'normal' (~10 mA) LEDs in this way without exceeding the current capacity of the port: however, if we use low-current (2 mA) LEDs, then – with most 8051 microcontrollers – this is possible.

Here, the required resistor values are calculated as follows:

$$R_{led} = \frac{V_{cc} - V_{diode}}{I_{diode}} = \frac{5V - 2V}{0.002A} = 1.5K\Omega$$

Note that you get 'nothing for nothing' in this world: the 2 mA LEDs will not be as bright as 10 mA or 20 mA LEDs.

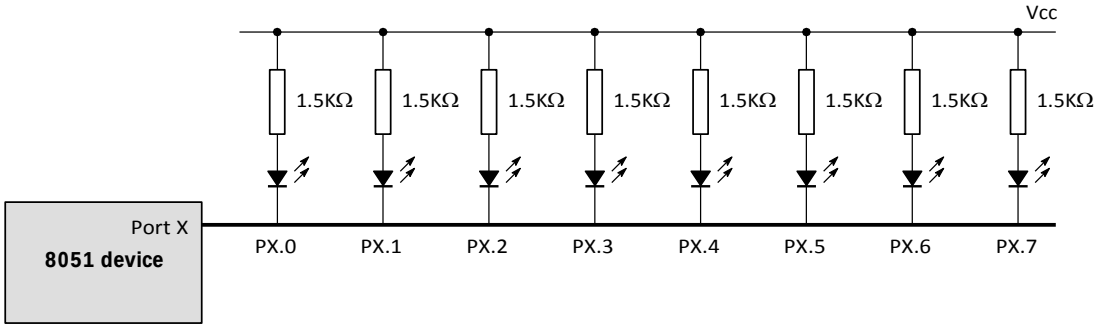


FIGURE 7.5 Using low-current 'naked' LEDs

Further reading

Horowitz, P. and Hill, W. (1989) *The Art of Electronics*, 2nd edn, Cambridge University Press, Cambridge, UK.

SSR DRIVER (DC)

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate hardware foundation for your application.

Problem

How do you 'switch on' or 'switch off' a piece of high-voltage (DC) electrical equipment using a microcontroller?

Background

A solid-state relay is a semiconductor device that was designed to take the place of a conventional electromagnetic (EM) relay. We do not recommend the use of EM relays for switching DC loads and therefore do not consider EM relays until we discuss the switching of AC loads in Chapter 8. If you are unfamiliar with EM relays, you may find it useful to refer to Chapter 8 (and specifically to **EMR DRIVER** [page 149]) for background information on EM relays before considering this pattern.

Solution

Unlike EM relays, solid-state relays (SSRs) are purely electronic in nature: they have no moving (mechanical) switch contacts. Both DC and AC solid-state relays are available: note that, unlike EM relays, a DC relay will **not** switch AC supplies and a DC relay will **not** switch AC supplies: we explain why this is the case later in the pattern.

SSRs provide very high levels of isolation between the 'control' and 'switching' circuits by optical techniques. The 'control' inputs will be connected to one (or more) LEDs. These will then, without any electrical link, control a phototransistor or photodiode array, which will, in turn, connect to further switching circuitry. In the case of a DC SSR, the switching circuit will typically be based on a MOSFET, and the current- and voltage-switching capabilities will generally be similar to MOSFETs.

Use of SSRs is generally straightforward: the inputs are directly compatible with microcontroller port voltages, and – because of the built-in opto-isolation – there is generally no need to add additional gates between the microcontroller and the SSR.

The examples below will illustrate the use of these devices.

Pull-up resistors

When using this pattern, you may need to incorporate pull-up resistors in your hardware design, at the input to the SSR. See **NAKED LED** [page 110] for further details.

Hardware resource implications

Every implementation of this pattern uses at least one port pin.

Reliability and safety issues

There are a number of general reliability and safety issues associated with the use of high-power DC loads: these are discussed in the pattern **BJT DRIVER** [page 124]. Please refer to this pattern for further information.

How robust are SSRs?

SSRs are less electrically robust than EM relays: in the presence of excessive voltages (due to back EMF from inductive loads, for example) or where the SSR is subjected to larger currents (possibly due to 'inrush'), the device will fail. If in doubt, over rate the device: that is, use a 300 V device where a 200 V device would probably do.

One other issue: we have seen people try to use more than one SSR in parallel in order to increase the current rating. This will only rarely work and is **never** reliable. The problem is that you have no way of ensuring that both relays switch on at **exactly the same time**. When one relay has turned on, the supply voltage drop will usually mean that the second (and subsequent) SSRs do not turn on – until the first relay fails. Thus, the likely result is that, within around a millisecond, all the relays will blow, in rapid succession.

What's in a name?

Despite the name, there are important differences between 'solid-state' and 'electromechanical' relays, particularly when it comes to circuit testing. If you are using an EM relay, you can check to make sure that the contacts are closing by using a multimeter to measure the resistance of the switch contacts: this resistance will be essentially zero when the contacts are closed and essentially infinite when they are open. This behaviour will be observed without connecting up the high-voltage side of the application.

You cannot test an SSR-based circuit in the same manner: most SSRs will always show an infinite resistance when measured with a multimeter. To test an SSR, you need to operate close to the specified operating voltage. Initial tests are therefore best performed (carefully) using a high-voltage load (we usually use an ordinary household lightbulb).

What happens when it goes wrong?

The typical failure mode of an SSR is an output short circuit: this can be dangerous.

Portability

SSRs can be used with any processor type. However, there are other portability issues to consider.

The most important (already briefly mentioned) is that an AC SSR cannot be used to switch DC. The reason for this is that the AC SSR contains zero-crossing detection circuits (see Chapter 8 for details). Because the DC supply never crosses zero, the SSR will never switch on.

Similarly, most DC SSRs are based on MOSFETs. Using a MOSFET to switch AC is ineffective: at best, the device will act as a form of rectifier for a short period, until it overheats.

Overall strengths and weaknesses

- ☺ **SSRs do not wear out (in normal use).**
- ☺ **SSRs are resistant to shock and vibration.**
- ☺ **SSRs have a high switching speed.**
- ☺ **SSRs have high levels of isolation between the 'control' and 'switching' circuits.**
- ☺ **SSRs generate only very low levels of electrical noise and generate no acoustic noise.**
- ☺ **SSRs do not exhibit switch bounce.**
- ☹ SSRs can be instantly and irrevocably damaged by excessive voltage and / or current.
- ☹ The typical failure mode of an SSR is an output short circuit: this can be dangerous.
- ☹ The 'switched' side has a minimum operating voltage and current: these may be quite high, complicating initial testing.
- ☹ EM relays can usually switch higher voltages and currents.
- ☹ Unlike an EM relay, the 'switched' side exhibits some leakage current in the off-state.
- ☹ The 'on' resistance is typically much larger than that of an EM relays: this translates directly into wasted heat and, hence, the need for heatsinks.

Related patterns and alternative solutions

See the other patterns in this chapter and Chapter 8, for alternative approaches.

Example: SSRs in telecommunication applications

Small DC semiconductor relays are commonly used in telecommunications equipment, such as modems, in place of larger EM relays. Indeed, the telecoms market is so important that special-purpose SSRs, intended to be used for telephone current line sensing, are also available (see, for example, products from Erg components).

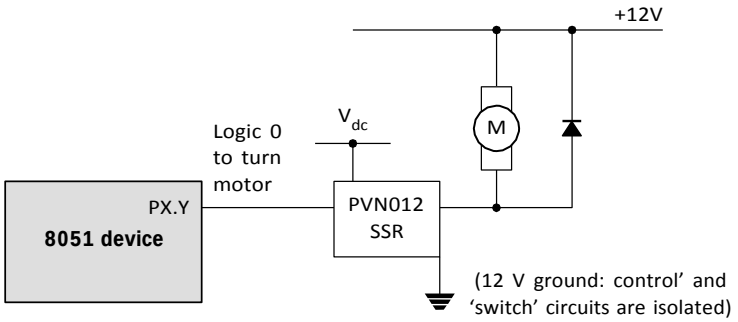
In modems and similar devices, these SSRs provide 200-300V (DC) output ratings at around 200 mA. They provide a low on resistance (typically 10Ω) and an 'off' resistance of some 500 MΩ, at voltages up to 4 kV.

Example: Open - loop DC motor control

In **MOSFET DRIVER** [page 139] we presented an example of a MOSFET used for open-loop DC motor control (see Figure 7.34 for details).

If we use an appropriate SSR we can simplify this circuit considerably. For example, our motor required 2A (continuous) at up to 12V for correct operation. Here we can use an IOR PVNO12 SSR. Unlike the majority of SSRs, this can switch AC or DC loads, of up to 20 V, 4.5A. It has no zero-crossing detection. The control current is a maximum of 10mA, which is compatible with our microcontroller. Figure 7.35 shows the required circuit.

Note that an important advantage of the SSR solution is that it provides a greater degree of isolation between the controller and the motor itself. Note also that, to control the speed of this motor, pulse-width modulation may be possible: see **HARDWARE PWM** [page 808] and **SOFTWARE PWM** [page 831].



chapter 9

A rudimentary software architecture

Introduction

In the first pattern in this chapter we consider the **minimum** software environment required to create a typical embedded application: this environment is called **SUPER LOOP** [page 162].

> Please note that we will use Super Loop in this book primarily to allow us to illustrate some introductory software patterns in Chapters 10, 11 and 12. In Chapter 13 we will demonstrate that a co-operative scheduler provides a more appropriate environment than a **SUPER LOOP** for the great majority of embedded applications.

The second pattern (**PROJECT HEADER** [page 169) is a practical implementation of a standard software design guideline: 'Do not duplicate information in numerous files; place the common information in a single file and refer to it where necessary.' Specifically, **PROJECT HEADER** pulls together the information connected with the particular microcontroller used in your application, along with other key pieces of information that are required by many of the files in your project.

SUPER LOOP

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontroller.
- You are designing an appropriate software foundation for your application.

Problem

What is the minimum software environment you need to create an embedded C program?

Background

—

Solution

One key difference between embedded systems and desktop computer systems is that the vast majority of embedded systems are required to run only one program. This program will start running when the microcontroller is powered up and will stop running when the power is removed.

A software architecture that is frequently used to generate the required behaviour is illustrated in Listings 9.1 to 9.3.

```
/*------------------------------------------------------------------ *-

   Main.C

  ------------------------------------------------------------------

   Architecture of a simple Super Loop application

   [Compiles and runs but does nothing useful]

-*------------------------------------------------------------------*/

#include "X.h"

/*------------------------------------------------------------------*/
void main(void)
   {
   // Prepare for task X
   X_Init();
   while(1) // 'for ever' (Super Loop)
```

```
      {
      X();  // Perform the task
      }
   }
/* ------------------------------------------------------------------*-
   -------END OF FILE ----------------------------------------
-*------------------------------------------------------------------*/
```

Listing 9.1 Part of a simple Super Loop demonstration

```
/*------------------------------------------------------------------

   *- X.H

  ------------------------------------------------------------------

   - see X.C for details.

-*------------------------------------------------------------------*/

// Function prototypes
void X_Init(void);
void X(void);

/*------------------------------------------------------------------*-
  ----- END OF FILE ----------------------------------------------
   -*------------------------------------------------------------------*/
```

Listing 9.2 Part of a simple Super Loop demonstration

```
/*------------------------------------------------------------------

   *- X.C

  ------------------------------------------------------------------

   'Task' for a demonstration Super Loop application

   [Compiles and runs but does nothing useful]

-*------------------------------------------------------------------*/

/*------------------------------------------------------------------*/
void X_Init(void)
   {
   // Prepare for task X
   // User code here ...
   }
/*------------------------------------------------------------------*/
```

```
void X(void)
   {
   // Perform task X
   // User code here ...
   }

/*---------------------------------------------------------------*-
  -----END OF FILE ----------------------------------------------
-*---------------------------------------------------------------*/
```

Listing 9.3 Part of a simple Super Loop demonstration

Listings 9.1 to 9.3 illustrate a simple embedded architecture, capable of running a single task (the function `X()`). After performing some system initialization (through the function `Init_System()`), the application runs the task 'X' repeatedly, until power is removed from the system.

Crucially, the 'Super Loop', or 'endless loop', is required because we have no operating system to return to: our application will keep looping until the system power is removed.

Hardware resource implications

SUPER LOOP has no significant hardware resource implications. It uses no timers, ports or other facilities. It requires only a few bytes of program code. It is impossible to create, in C, a working environment requiring fewer system resources.

Reliability and safety implications

Applications based on **SUPER LOOP** can be both reliable and safe, because the overall architecture is very simple and easy to understand and no aspect of the underlying hardware is hidden from the original developer or from the person who subsequently has to maintain the system. If, by contrast, you are programming for Windows or a similarly complex desktop environment (including Linux or Unix), you are not in complete control: if someone else wrote poor code in a library, it may crash your program. With a 'super looping' application, there is nobody else to blame. This can be particularly attractive in safety-related applications.

Please note, however, that just because an application is based on a Super Loop does not mean that it is safe. Indeed, in general, a Super Loop does not provide the facilities needed in an embedded application: in particular, it does not provide a mechanism for calling functions at predetermined time intervals. As we discussed in Chapter 1, these are key characteristics of most embedded applications: if you need such facilities, a scheduler (see Chapter 13) is almost always a more reliable environment.

Portability

Any 'C' compiler intended for embedded applications will compile a Super Loop program: the loop is based entirely on ISO/ANSI 'C'. The code is therefore inherently portable.

Overall strengths and weaknesses

- ☺ **The main strength of Super Loop systems is their simplicity. This makes them (comparatively) easy to build, debug, test and maintain.**
- ☺ **Super Loops are highly efficient: they have minimal hardware resource implications.**
- ☺ **Super Loops are highly portable.**
- ☹ If your application requires accurate timing (for example, you need to acquire data precisely every 2 ms), then this framework will not provide the accuracy or flexibility you require.
- ☹ The basic Super Loop operates at 'full power' (normal operating mode) at all times. This may not be necessary in all applications, and can have a dramatic impact on system power consumption. Again, a scheduler can address this problem.

Related patterns and alternative solutions

In most circumstances, a scheduler will be a more appropriate choice: see Chapter 13.

Example: Central-heating controller

Suppose we wish to develop a microcontroller-based control system to be used as part of the central-heating system in a building. The simplest version of this system might consist of a gas-fired boiler (which we wish to control), a sensor (measuring room temperature), a temperature dial (through which the desired temperature is specified) and the control system itself (Figure 9.1).

We assume that the boiler, temperature sensor and temperature dial are connected to the system via appropriate ports. We further assume that the control system is to be implemented in 'C'.

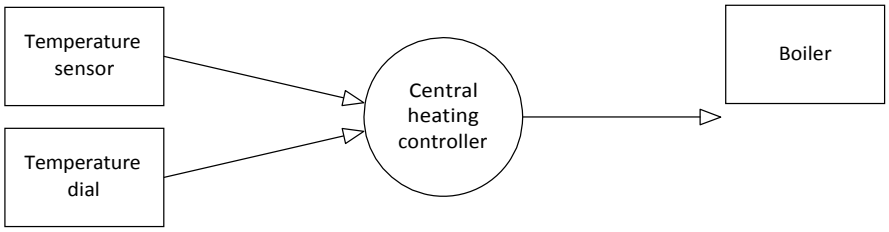


FIGURE 9.1 An overview of the required central heating controller

Here, precise timing is not required, and a Super Loop framework similar to that shown in Listings 9.4 to 9.6 may be appropriate.

```
/*-------------------------------------------------------------------*-

   Main.C

  ------------------------------------------------------------------

   Framework for a central heating system using 'Super Loop'.

   [Compiles and runs but does nothing useful]

-*-------------------------------------------------------------------*/

#include "Cen_Heat.h"

/*-------------------------------------------------------------------*/
void main(void)
   {
   // Init the system
   C_HEAT_Init();

   while(1) // 'for ever' (Super Loop)
      {
      // Find out what temperature the user requires
      // (via the user interface)
      C_HEAT_Get_Required_Temperature();

      // Find out what the current room temperature is
      // (via temperature sensor)
      C_HEAT_Get_Actual_Temperature();

      // Adjust the gas burner, as required
      C_HEAT_Control_Boiler();
      }
   }

/*-------------------------------------------------------------------*-
  ----- END OF FILE -------------------------------------------------
-*-------------------------------------------------------------------*/
```

Listing 9.4 Part of the code for a simple central-heating system

```
/*-------------------------------------------------------------------*-

   Cen_Heat.H

  ------------------------------------------------------------------
```

```
   - see Cen_Heat.C for details.
-*---------------------------------------------------------------------*/

// Function prototypes
void C_HEAT_Init(void);
void C_HEAT_Get_Required_Temperature(void);
void C_HEAT_Get_Actual_temperature(void);
void C_HEAT_Control_Boiler(void);

/*---------------------------------------------------------------------*-
 ---- END OF FILE ------------------------------------------------
-*---------------------------------------------------------------------*/
```

Listing 9.5 Part of the code for a simple central-heating system

```
/*---------------------------------------------------------------------*-

   Cen_Heat.C

---------------------------------------------------------------------

   Framework for a central heating system using 'Super Loop'.

   [Compiles and runs but does nothing useful]

-*---------------------------------------------------------------------*/

/*---------------------------------------------------------------------*/

void C_HEAT_Init(void)
   {
   // User code here ...
   }

/*---------------------------------------------------------------------*/

void C_HEAT_Get_Required_Temperature(void)
   {
   // User code here ...
   }

/*---------------------------------------------------------------------*/

void C_HEAT_Get_Actual_temperature(void)
   {
   // User code here ...
   }

/*---------------------------------------------------------------------*/
```

```
void C_HEAT_Control_Boiler(void)
   {
   // User code here ...
   }

/*---------------------------------------------------------------------*-
  -------- END OF FILE ------------------------------------------
-*---------------------------------------------------------------------*/
```

Listing 9.6 Part of the code for a simple central-heating system

Further reading

—

PROJECT HEADER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate software foundation for your application.

Problem

How do you group together all the information relating to the hardware platform used in your project?

Background

—

Solution

As we saw in Chapter 3, the 8051 family shares a common set of core facilities. However, it is a *family* – rather than a group of clones – and the different family members have different features and facilities. For example, some devices require 12 oscillator cycles per instruction, while others perform the same instruction in 6, 4 or even 1 oscillator cycle (see Chapters 3 and 4).

If you create an application using a particular 8051 device operating at a particular oscillator frequency, this information will be required when compiling many of the different source files in your project. This information will also be required by anyone who wishes to use your code.

The 'Project Header' is simply a header file, included in all projects, that groups all of this information in one place. As such, it is a practical implementation of a standard software design guideline: 'Do not duplicate information in numerous files; place the common information in a single file, and refer to it where necessary.'

In the case of the great majority of the examples in this book, we use a Project Header file. This is always called `Main.H`. An example of a typical project header file is included in Listing 9.7. Please note that this is a real example and not all of the features of this file have yet been considered in this book.

```
/*-------------------------------------------------------------------*-

   Main.H (v1.00)

  ------------------------------------------------------------------

   Project Header for project LCD_KEY (see Chapter 22)
```

```
-*------------------------------------------------------------------*/
#ifndef _MAIN_H
#define _MAIN_H

//------------------------------------------------------------------------
// WILL NEED TO EDIT THIS SECTION FOR EVERY PROJECT
//------------------------------------------------------------------------

// Must include the appropriate microcontroller header file here
#include <AT89×52.h>

// Must include oscillator / chip details here if delays are used
// -
// Oscillator / resonator frequency (in Hz) e.g. (11059200UL)
#define OSC_FREQ (11059200UL)
// Number of oscillations per instruction (6 or 12)
#define OSC_PER_INST (12)

//------------------------------------------------------------------------
// SHOULD NOT NEED TO EDIT THE SECTIONS BELOW
//------------------------------------------------------------------------
typedef unsigned char tByte;
typedef unsigned int  tWord;
typedef unsigned long tLong;

// Misc #defines
#ifndef TRUE
#define FALSE 0
#define TRUE (!FALSE)
#endif

#define RETURN_NORMAL (bit) 0
#define RETURN_ERROR (bit) 1

//------------------------------------------------------------------------
// Interrupts
// – see Chapter 12.
//------------------------------------------------------------------------

// Generic 8051 timer interrupts (used in most schedulers)
#define INTERRUPT_Timer_0_Overflow 1
#define INTERRUPT_Timer_1_Overflow 3
#define INTERRUPT_Timer_2_Overflow 5

// Additional interrupts (used in shared-clock schedulers)
#define INTERRUPT_UART Rx_Tx 4
#define INTERRUPT_CAN_c515c 17

// ------------------------------------------------------------------------
```

```
// Error codes
// – see Chapter 13.
//---------------------------------------------------------------------

#define ERROR_SCH_TOO_MANY_TASKS (1)
#define ERROR_SCH_CANNOT_DELETE_TASK (2)

#define ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK (0xAA)
#define ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER (0xAA)

#define ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START (0xA0)
#define ERROR_SCH_LOST_SLAVE (0x80)

#define ERROR_I2C_WRITE_BYTE_AT24C64 (11)
#define ERROR_I2C_READ_BYTE_AT24C64 (12)
#define ERROR_I2C_WRITE_BYTE (13)
#define ERROR_I2C_READ_BYTE (14)

#define ERROR_USART_TI (21)
#define ERROR_USART_WRITE_CHAR (22)

#endif
//=====================================================================
```

Listing 9.7 An example of a typical project headerfile (Main. H)

Hardware resource implications

There are no hardware resource implications.

Reliability and safety implications

Use of **Project Header** can help to improve reliability, not least because it helps to make your code more readable, because anyone using your projects knows where to find key information, such as the model of microcontroller and the oscillator frequency.

Use of **Project Header** can help to improve the reliability of applications which are subsequently ported to a different microcontroller, as discussed in the remainder of this chapter.

Portability

The use of a project header can help to make your code more easily portable, by placing some of the key rnicrocontroller-dependent data in one place.

In addition, the `typedef` statements in the file create three key user-defined types which are used in all of the projects in this book:

```
typedef unsigned char tByte;
typedef unsigned int  tWord;
typedef unsigned long tLong;
```

Thus, in the projects you will see code like this:

```
tWord Temperature;
```

Rather than:

```
unsigned int Temperature;
```

If the code is ported into – say – a 16-bit environment, changes to only three `typedef` statements are required in order to adapt the variable sizes to a new compiler. Without the use of these user-defined types, porting the code becomes more complicated and error prone.

Overall strengths and weaknesses

- **PROJECT HEADER can help to make your code more readable, not least because anyone using your projects knows where to find key information, such as the model of microcontroller and the oscillator frequency.**
- **PROJECT HEADER can help to make your code more easily portable.**

Related patterns and alternative solutions

See **PORT HEADER** [page 184].

Examples

Almost every example project on the CD includes a project header file. Search for the file `Main.H`.

chapter 10

Using the ports

Introduction

The first pattern in this chapter (**PORT I/O** [page 174]) is concerned with basic software techniques for interacting with the digital ports on an 8051 microcontroller.

The second pattern (**PORT HEADER** [page 184]) encapsulates a design guideline that helps you cope with the fact that many different components in a larger project will each require port access: specifically, **PORT HEADER** demonstrates how the port access for the whole project can be integrated into a single file. Use of these techniques can ease project development, maintenance and porting.

The following points should also be noted:

- This chapter does not consider the hardware that will be connected to the port: see Chapters 7 and 8 for relevant hardware details.
- This chapter does not consider analog input and output: for this, see Part G.

PORT I/O

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate software foundation for your application.

Problem

How do you write software to read from and /or write to the ports on an (8051) microcontroller?

Background

The Standard 8051s have four 8-bit ports. All of the ports are bidirectional: that is, they may be used for both input and output.

To limit the size of the device, some of the port pins have alternative functions. For example, as we saw in Chapter 6, Ports 0, 2 (and part of Port 3) together provide the address and data bus used to support access to external memory. Similarly, two further pins on Port 3 (pins 0 and 1) also provide access to the on-chip USART (see Chapter 18). When in their 'alternative roles', these pins cannot be used for ordinary input or output. As a result, on the original members of the 8051 family, where external memory is used, only Port 1 is available for general-purpose I/O operations.

These comments all refer to the Standard 8051: the number of available ports on 8051 microcontrollers varies enormously: the Small 8051s have the equivalent of approximately two ports and the Extended 8051s have up to ten ports (see Chapter 3). Despite these differences, the control of ports on all members of the 8051 family is carried out in the same way.

Solution

Control of the 8051 ports through software is carried out using what are known as 'special function registers' (SFRs). The SFRs are 8-bit latches: in practical terms, this means that the values written to the port are held there until a new value is written or the device is reset.

Each of the four basic ports on the Standard 8051 family, as well as any additional ports, is represented by an SFR: these are named, appropriately, P0, P1, P2, P3 and so on. Physically, the SFR is a area of memory in the upper areas of internal RAM: P0 is at address 0x80, P1 at address 0x90, P2 at address 0xA0 and P3 at address 0xB0.

If we want to read from the ports, we need to read from these addresses. Assuming that we are using a C compiler, the process of writing to an address is usually by

means of a SFR variable 'declaration', hidden in a header file. Thus, a typical SFR header file for an 8051 family device will contain the lines:

```
sfr P0 = 0x80;
sfr P1 = 0x90;
sfr P2 = 0xA0;
sfr P3 = 0xB0;
```

Having declared the SFR variables, we can write to the ports in a straightforward manner. For example, we can send some data to Port 1 as follows:

```
unsigned char Port_data:

Port_data = 0x0F;

P1 = Port_data; // Write 00001111 to Port 1
```

Similarly, we can read from (for example) Port 1 as follows:

```
unsigned char Port_data;

P1 = 0xFF;        // Set the port to 'read mode'
Port_data = P1;   // Read from the port
```

Note that, in order to read from a pin, we need to ensure that the last thing written to the pin was a '1'. Because the reset value of the ports is 0xFF (see 'Port reset values', page 178), it is tempting to assume that writing this value is unnecessary and that we can get away with the following version:

```
unsigned char Port_data;

// Assume nothing written to port since reset
// - DANGEROUS!!!
Port_data = P1;
```

The problem with this code is that, in simple test programs it works: this can lull the developer into a false sense of security. If, at a later date, someone modifies the program to include a routine for writing to all or part of the same port, this code will not generally work as required:

```
unsigned char Port-data:

P1 = 0x00;

...

// Assumes nothing written to port since reset
// - WON'T ALWAYS WORK
Port_data = P1;
```

In general, we use initialization functions to set the ports to a known state at the start of the program. Where this is not possible, it is safer to always write '1' to any port pin before reading from it, as was illustrated in the first example.

Note that, in the last example, we assume that we wished to read from or write to an entire port. More commonly, we might wish (for example) to control an LED is connected to a single output pin. For example, assuming that the LED is connected to Pin 0 on Port 3 of an 8051-family microcontroller, we can flash the diode by controlling the whole port, as follows:

```
P3 = 0xFF:
... // delay
P3 = 0x00;
... // delay
P3 = 0xFF ;
... // etc
```

Alternatively, we can make use of an `sbit` variable in the C51 compiler to provide a finer level of control. At the same time, we consider the fact that – depending on the hardware (see Chapter 7) – the LED may be lit using a logic 1 or a logic 0 output on the port pin and we make the software flexible enough to deal easily with subsequent hardware changes:

```
#define LED_PORT P3

#define LED_ON 0        // Easy to change the logic here
#define LED_OFF 1

sbit Warning_led = LED_PORT^0;  // LED is connected to 4.0

...

Warning_led = LED_ON;
... // delay
Warning_led = LED_OFF;
... // delay
Warning_led = LED_ON:
... // etc
```

Reliability and safety implications

Port reset values

After the system is reset, the contents of the various port special function registers (SFRs) are set to 0xFF. This fact has very important safety and reliability implications.

Consider, for example, that you have connected a motorized device to a port and that the device is activated by a 'logic 1' output. When the microcontroller is reset, the motorized device will be activated. Even if you change the port outputs to 0 at the start of your program, the motor will be 'pulsed' briefly. This can, in some systems, lead to the injury or even death of users of the system or those in the immediate vicinity.

Because the output pins are 'reset high' it is important to ensure that any devices which have safety implications are connected to the microcontroller in such a way

that they are 'active low': that is, that an output of '0' on the relevant port pin will activate the device.

Port I/O and memory access

One of the most common errors made by inexperienced 8051 developers is to continue to use P0, P2 or P3 as normal I/O ports when using external memory (see Chapter 8 for details of memory usage).

If you use external memory, you cannot safely use P0 and P2 for any other purpose and you must also take care when writing to Port 3.

For example, any statement similar to this:

```
P3 = AD_data;
```

is potentially catastrophic if external memory is being used.

Instead, make use of `sbit` variables to ensure you only write to 'safe' port pins (see 'Example: Reading and writing bits' for details).

chapter 11

Delays

Introduction

The creation of accurate delays are key requirements in many embedded applications.

In this chapter we will consider two different techniques that may be used to provide such facilities:

- **HARDWARE DELAY** [page 194] which is capable of producing precise delays through the use of one of the on-chip timers. Particularly suitable for generating delays of around 0.1 ms or more.
- **SOFTWARE DELAY** [page 206] which is a simple technique that requires no hardware resources. The most flexible form of delay mechanism that is particularly suitable for generating short delays (measured in microseconds) or where timer resources are not available.

HARDWARE DELAY

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate software foundation for your application.

Problem

You need to wait for a fixed period of time (measured in milliseconds) before taking some action.

Background

—

Solution

All members of the 8051 family have at least two 16-bit timer / counters, known as Timer 0 and Timer 1. These timers can be used to generate accurate delays.

We begin by providing brief information on these timers.

Timer 0 and Timer 1

Timer 0 and Timer 1 have much in common and we will consider them together.

SOFTWARE DELAY

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate software foundation for your application.

Problem

How do you create a simple delay without using any hardware (timer) resources?

Background

See **HARDWARE DELAY** [page 194] for background information.

Solution

Suppose we want to flash the LEDs connected to Port 1 on an 8051 microcontroller with a two second cycle time (so that they are on for 1 second then off for 1 second, *ad infinitum*). The basic program structure we require could be based on a **SUPER LOOP** [page 162] as follows:

```
...
while (1)
   {
   P1 = 0xFF;
   // Delay one second
   P1 = 0x00;
   // Delay one second
   }
```

If we wish to implement these delays and have no available timers (see **HARDWARE DELAY** [page 194]), we could use a 'Software Delay', implemented as follows:

```
Loop_Delay()
   {
   unsigned int x;

   for (x=0; x <= 65535; x++);
   }
```

We could then measure the pulse frequency we obtained, using an oscilloscope or a software simulator. If we found that these delays were not long enough, we could easily extend them by adding additional layers, as shown in `Longer_Loop_Delay()`

```
Longer_Loop_Delay()
   {
   unsigned int x;
   unsigned int y;

   for (x=0; x<=65535; x++)
      {
      for (y=0; y<=65535; y++);
      }
   }
```

Hardware resource implications

Unlike **HARDWARE DELAY** [page 194], **SOFTWARE DELAY** uses no timer resources. Note, however, that the CPU time spent in the delay calculation is wasted: using a scheduler (see Chapter 13) can, in many circumstances, avoid the waste of CPU time.

Reliability and safety implications

Software delays are not suitable for use in applications where precise timing is required.

Portability

Software delays can be used even on a microcontroller / microprocessor without a built-in timer. However, the precise delay duration obtained varies (enormously) with differences in hardware and software.

Overall strengths and weaknesses

- ☺ **SOFTWARE DELAY can be used to produce very short delays.**
- ☺ **SOFTWARE DELAY requires no hardware timers.**
- ☺ **SOFTWARE DELAY will work on any microcontroller.**
- ☹ It is very difficult to produce precisely timed delays.
- ☹ The loops must be returned if you decide to use a different processor, change the clock frequency or even change the compiler optimization settings.

Related patterns and alternative solutions

In most circumstances, it is better to avoid using delays at all: see **CO-OPERATIVE SCHEDULER** [page 255] for a delay-free alternative that will work in many circumstances.

If you do require delays then **HARDWARE DELAY** [page 194] is often a better alternative.

chapter 12

Watchdogs

Introduction

Suppose there is a hungry dog guarding a house (Figure 12.1), and someone wishes to break in. If the burglar's accomplice repeatedly throws the guard dog small pieces of meat at 2-minute intervals, then the dog will be so busy concentrating on the food that he will ignore his guard duties and will not bark. However, if the accomplice run out of meat or forgets to feed the dog for some other reason, the animal will start barking, thereby alerting the neighbours, property occupants or police.

FIGURE 12.1 The origins of the 'watchdog' analogy

This same basic approach is followed in computerized 'watchdog timers'. Very simply, these are timers which, if not refreshed at regular intervals, will overflow. In most cases, overflow of the timer will reset the system. Such watchdogs are intended to deal with the fact that, even with meticulous planning and careful design, embedded systems can 'hang' due to unexpected problems. The use of a watchdog can be used to recover from this situation, in certain circumstances.

The pattern **HARDWARE WATCHDOG** [page 217] considers how to apply these techniques to good effect in your embedded application.

HARDWARE WATCHDOG

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- You are designing an appropriate software foundation for your application.

Problem

How can you ensure that – if your application 'hangs' due to an unexpected software or hardware error – the system will automatically reset itself?

Background

See the introduction to this chapter for an explanation of the watchdog analogy.

Solution

Working with **HARDWARE WATCHDOG** means using either an internal or external (hardware) timer.

We have seen in many previous cases that, where available, the use of on-chip components is to be preferred to the use of equivalent off-chip components. Specifically, on-chip components generally offer the following benefits:

- Reduced hardware complexity, which tends to result in increased system reliability.
- Reduced application cost.
- Reduced application size.

In the case of watchdog timers, the situation is more complex, because external watchdog chips typically provide some useful facilities that are not available in most on-chip versions.

For example, the popular '1232' watchdogs (available, in various versions, from Dallas Semiconductors, Maxim, Linear Technology and Analog Devices) are low-cost, low-power devices. In addition to functioning as a watchdog timer, they also provide power system monitoring capabilities (see **ROBUST RESET** [page 77] for details of this). If, as in many designs, you intend to use an external 'robust reset' circuit anyway, then the 1232 chips allow you to incorporate an external watchdog facility for minimal addition cost and only a very minor increase in hardware complexity.

Another beneficial feature of external watchdogs is that they are inherently portable: you can generally use the same external watchdog with any member of the 8051 family. By contrast, code written to work with an internal watchdog will generally have to be rewritten for use with a different hardware.

One situation in which on-chip watchdogs (such as those in the Infineon c515x devices) can be beneficial is where they allow you to determine whether the system has undergone a normal reset or a reset caused by a watchdog overflow. This may allow you to modify the system behaviour to match these circumstances. Without this information (which is not generally available through external watchdogs without some complex coding) your system may be continually reset by the watchdog timer overflow.

We can summarize by saying that, if you require watchdog facilities, you need to consider both internal and external solutions carefully. There is no single 'ideal' solution and – considering the issues mentioned earlier – you need to find the best match to your requirements.

Reliability and safety implications

Before using either an internal or external watchdog, you need to be sure that the use of such a timer will increase (rather than decrease) the reliability of your application.

The first thing to bear in mind is that watchdog behaviour should be for **disaster recovery**. In a well-designed system the occurrence of a watchdog reset should be a noteworthy event that occurs rarely. If you think of the use of watchdogs in terms of 'if all else falls, then we will have to let the watchdog reset the system', then you are taking a realistic view of the capabilities of this approach.

Used without due care at the design phase and/or adequate testing, watchdogs can reduce the system reliability dramatically. A particular problem with a badly designed watchdog can occur in the presence of sustained hardware faults. In these circumstances, a badly implemented watchdog can mean that your system constantly resets itself. This can be extremely dangerous.

You also need to appreciate that watchdogs are unsuitable for many applications, because the time taken to react to an error is too long. Suppose, for example, the braking system in an automotive application uses a 500 ms watchdog and the vehicle encounters a problem when it is travelling at 70 miles per hour (110 km per hour). In these circumstances, the vehicle and its passengers will have travelled some 16 yards / 15 metres – right into the car in front – before the vehicle even begins to reset the braking system. In short, where fast recovery is required, watchdogs are rarely the best solution.

Portability

As already noted, internal watchdogs are based on hardware that is not part of the 8051/52 core. As a result, different forms of watchdog now exist on the various different 8051 derivatives and code written for one on-chip watchdog will generally need to be adapted for use with a different device. By contrast, software written for external watchdogs can be more portable.

Overall strengths and weaknesses

- ☺ **Watchdogs can provide a 'last resort' form of error recovery. If you think of the use of watchdogs in terms of 'if all else fails, then reset the system', then you are taking a realistic view of the capabilities of this approach.**
- ☺ **In the presence of intermittent faults, e.g. rare bursts of EMI, watchdogs can be very effective.**
- ☹ Watchdogs with long timeout periods are unsuitable for many applications.
- ☹ Used without due care at the design phase and / or adequate testing, watchdogs can reduce the system reliability dramatically.
- ☹ In the presence of sustained hardware faults, badly implemented watchdogs can mean that your system constantly resets itself. This can be very dangerous.

The user interface

In this part, we will consider some key patterns suitable for assisting in the creation of user interfaces for a wide range of embedded applications.

In Chapter 18, we present **PC LINK (RS-232)** [page 364]. This pattern considers how, using the universal 'RS-232' standard, we can transfer data between an embedded device and a desktop, notebook or similar PC. In addition to being a useful pattern in its own right, this pattern illustrates many of the features required in software developed for a co-operatively scheduled environment.

In Chapter 19, we will explore techniques for reading inputs from switches: here we will be concerned with both software-only and hardware-based interfaces. In Chapter 20, we will see how the same basic techniques may be extended to work efficiently with keypads. In Chapter 21, we turn our attention to the creation of LED displays. In particular, we will be concerned with multiplexed LED interfaces, which are the only cost-effective solution in most applications.

Finally, in Chapter 22, we will consider how to control LCD (text) displays. In this chapter we will be specifically concerned with the interaction with displays based on the ubiquitous Hitachi HD44780 LCD controller chip.

chapter 18

Communicating with PCs via RS-232

Introduction

In this chapter we present **PC LINK (RS-232)**. This pattern describes how, using the universal 'RS-232' standard, we can transfer data between an embedded device and a desktop, notebook or similar PC.

Note: We are **not** concerned here with the process of transferring data between two (or more) microcontrollers: this is discussed in Part F.

PC LINK (RS-232)

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you link your embedded application to a desktop (or notebook or hand-held) PC using 'RS-232'?

Background

This pattern is concerned with the use of what is commonly referred to as the RS-232 communication protocol, to transfer data between an 8051 microcontroller and some form of personal computer (PC, notebook or similar).

We provide some important background information on RS-232 in this section. We begin, however, by considering some important terminology used in the communications area.

What are 'simplex', 'half-duplex' and 'duplex' serial communications?

In a **simplex** serial communication system, we typically require at least two wires ('signal' and 'ground'). Data transfer is only in one direction: for example, we might transfer data from a simple data-monitoring system to a PC using simplex data transfer.

In a **half-duplex** serial communication system we again (typically) require at least two wires. Here, data transfer can be in both directions. However, transmission can only be in one direction at a time. In a variation of the data-monitoring example, we might use a half-duplex communication strategy to allow us to send information to the embedded module (to set parameters, such as sampling rate). We might also, at other times, use the same link to download the data from the embedded board to a PC.

In a **full-duplex** serial communication system, we typically require at least three wires (Signal A, Signal B, Ground). The line Signal A will carry data in one direction **at the same time that** Signal B carries data in the other direction.

What is 'RS-232'?

In 1997 the Telecommunications Industry Association released what is formally known as TIA-232 Version F, a serial communication protocol which has been

universally referred to as 'RS-232' since its first 'Recommended Standard' appeared in the 1960s. Similar standards (V.28) are published by the International Telecommunications Union (ITU) and by CCITT (Consultative Committee International Telegraph and Telephone).

The 'RS-232' standard includes details of:

- The protocol to be used for data transmission.
- The voltages to be used on the signal lines.
- The connectors to be used to link equipment together.

Overall, the standard is comprehensive and widely used, at data transfer rates of up to around 115 or 330 kbits / second (115 / 330 k baud). Data transfer can be over distances of 15 metres or more.

Note that RS-232 is a peer-to-peer communication standard. Unlike the multi-drop RS-485 standard (which we consider in Chapter 27), RS-232 is intended to link only two devices together.

Basic RS-232 protocol

RS-232 is a character-oriented protocol. That is, it is intended to be used to send single 8-bit blocks of data. To transmit a byte of data over an RS-232 link, we generally encode the information as follows:

- We send a 'Start' bit.
- We send the data (8 bits).
- We send a 'Stop' bit (or bits).

We consider each of these stages here.

Quiescent state

When no data are being sent on an RS-232 'transmit' line, the line is held at a Logic 1 level.

Start bit

To indicate the start of a data transmission we pull the 'transmit' line low.

Data

Data are often encoded in ASCII (American Standard Code for Information Interchange), in 7-bit form. The bits are sent least significant bit first. If we are sending 7-bit data, the eighth data bit is often used as a simple parity check bit and is transmitted in order to provide a rudimentary error detection facility on a character-by-character basis.

Note that none of the code presented here uses parity bits: we use all 8 bits for data transfer.

Stop bit(s)

The stop bits consist of a Logic 1 output. These can be 1 or, less commonly, 1G or 2 pulses wide.

Note that we will use a single stop bit in all code example.

Asynchronous data transmission and baud rates

RS-232 uses an asynchronous protocol. This means that no clock signal is sent with the data. Instead, both ends of the communication link have an internal clock, running at the same rate. The data (in the case of RS-232, the 'Start' bit) is then used to synchronize the clocks, if necessary, to ensure successful data transfer.

RS-232 generally operates at one of a (restricted) range of baud rates. Typically these are: 75, 110, 300, 1,200, 2,400, 4,800, 9,600, 14,400, 19,200, 28,800, 33,600, 56,000, 115,000 and (rarely) 330,000 baud. Of these, 9,600 baud is a very 'safe' choice, as it is very widely supported.

RS-232 voltage levels

The threshold levels used by the receiver are +3V and -3V and the lines are inverted. The maximum voltage allowed is +/- 15V.

Note that these voltages cannot be obtained directly from the naked microcontroller port pins: some form of interface hardware is required. For example, the Maxim Max232 and Max233 are popular and widely used line driver chips; we consider how to use such chips in 'Solution'.

Flow control

RS-232 is often used with some form of flow control. This is a protocol, implemented through software or hardware, that allows a receiver of data to tell the transmitter to pause the dataflow. This might be necessary, for example, if we were sending data to a PC and the PC had filled a RAM buffer: the PC would then tell our embedded application to pause the data transfer until the buffer contents had been stored on disk.

Although hardware handshaking can be used, this requires extra signal lines. The most common flow control technique is 'Xon / Xoff' control. This requires a half- or full-duplex communication link, and can operate as follows:

1. Transmitter sends a byte of data.
2. The receiver is able to receive more data: it does nothing.
3. The transmitter sends another byte of data.
4. Steps 1–3 continue until the receiver cannot accept any more data: it then sends a 'Control s' (Xoff) character back to the transmitter.
5. The transmitter receives the 'Xoff' command and pauses the data transmission.

6 When the receiver node is ready for more data, it sends a 'Control q' (Xon) character to the transmitter.

7 The transmitter resumes the data transmission.

8 The process continues from Step 1.

Solution

In this section we consider how to implement an RS-232 link from an embedded 8051 microcontroller to a PC.

Transceiver hardware

As noted in 'Background', the voltages used in RS-232 communications are incompatible with the voltages used on the microcontroller itself. Therefore, you will require some form of voltage level conversion circuitry between the microcontroller board and the PC cable.

Usually the most cost-effective way of achieving this is to use a 'transceiver' (transmitter-receiver) chip created for this purpose. Of these, the Max232 (from Maxim) is frequently used; however, in new designs the more recent Max233 is a better choice as it requires fewer external components.

The required link to a Max233 is illustrated in Figure 18.1, which also shows the standard connections (9-pin D-type socket: 'DB-9') used for most recent designs.

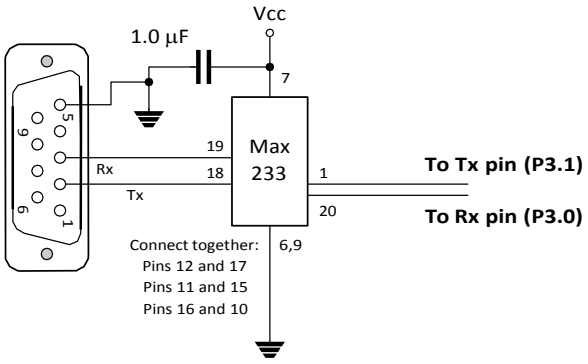


FIGURE 18.1 Using a Max233 as an RS-232 transceiver

Cable connections

To connect to the PC using the software presented here, you need a 3-wire cable.

Use DB-9 socket ('female') connector at the PC end and DB-9 plug ('male') connector at the microcontroller end.

The required cable connections are a 'straight through' cable as described in Table 18.1.

TABLE 18.1 Connections required for a 'straight through' cable needed to link the microcontroller to the desktop PC (or similar terminal)

PC (COM1, COM2) DB-9 connector		Microcontroller DB-9 connector
RxD – Pin 2	to	TxD – Pin 2
TxD – Pin 3	to	RxD – Pin 3
Ground – Pin 5	to	Ground – Pin 5

The software architecture

Suppose we wish to transfer data to a PC at a standard 9,600 baud rate; that is, 9,600 bits per second. As we discussed in 'Background', transmitting each byte of data, plus stop and start bits, involves the transmission of ten bits of information (assuming a single stop bit is used). As a result, each byte takes approximately 1 ms to transmit.

This has important implications in all applications, not least those using a scheduler. If, for example, we wish to send this information to the PC:

```
Current core temperature is 36.678 degrees
```

then the task sending these 42 characters will take more than 40 milliseconds to complete. This will frequently be an unacceptably long duration.

The most obvious way of solving this problem is to increase the baud rate; however, this is not always possible and it does not solve the underlying problem.

A better solution is to write all data to a buffer in the microcontroller. The contents of this buffer will then be sent – usually one byte at a time – to the PC, using a regular, scheduled task.

This solution is used in the code libraries presented in the following sections and included on the accompanying CD. The architecture is a good example of a MULTISTAGE TASK [page 317], and is frequently used in user-interface libraries: LCD CHARACTER PANEL [page 467], for example, uses a very similar architecture.

Using the on-chip U(S)ART for RS-232 communications

Having decided on the basic architecture for the RS-232 library, we need to consider in more detail how the on-chip serial port is used.

This port is full duplex, meaning it can transmit and receive simultaneously. It is also receive buffered, meaning it can commence reception of a second byte before a previously received byte has been read from the receive register. (However, if the first byte still has not been read by the time reception of the second byte is complete, one of the bytes will be lost.)

The serial port can operate in four modes (one synchronous mode, three asynchronous modes). In this pattern, we are primarily interested in Mode 1. In this mode, 10 bits are transmitted (through TxD) or received (through RxD): a start bit (0), 8 data bits (lsb first), and a stop bit (1).

Note that the serial interface may also provide interrupt requests when transmission or reception of a byte has been completed. However, for reasons discussed in Chapter 1, none of the code used in this pattern will generate interrupts.

Serial port registers

The serial port control and status register is the special function register SCON. This register contains the mode selection bits (and the serial port interrupt bits, TI and RI).

SBUF is the receive and transmit buffer of serial interface. Writing to SBUF loads the transmit register and initiates transmission. Reading out SBUF accesses a physically separate receive register.

chapter 19

Switch interfaces

Introduction

Reading the status of one or more push-button switches is a very common requirement in embedded applications. The four patterns in this chapter describe different solutions to this problem.

The patterns are as follows:

- **SWITCH INTERFACE (SOFTWARE)** [page 399]
  Uses a minimum of external hardware. Reads a single switch, debounces it and reports its status
- **SWITCH INTERFACE (HARDWARE)** [page 410]
  Use external hardware to perform switch debouncing. Increased cost (compared with **SWITCH INTERFACE (SOFTWARE)**), but much higher level of protection against ESD, malicious damage etc.
- **ON-OFF SWITCH** [page 414]
  Building on **SWITCH INTERFACE (SOFTWARE)** or **SWITCH INTERFACE (HARDWARE)** this pattern provides, through software, the following behaviour:
  Assume that the switch is in the OFF state. It remains in this state until it is pressed. When pressed, the state changes to ON. It remains in this state until it is pressed. When pressed, the state changes to OFF.
  This type of behaviour can be used, for example, to allow a single (non-latching) switch to control a piece of machinery.
- **MULTI-STATE SWITCH** [page 423]
  Building on **SWITCH INTERFACE (SOFTWARE)** or **SWITCH INTERFACE (HARDWARE)** this pattern provides, through software, the following behaviour:
  Assume that the switch state is in the OFF state. The switch is pressed and briefly held. The switch state changes to 'ON STATE 1'. The user continues to depress the switch. The switch state becomes 'ON STATE 2', etc. Releasing the switch (at any time) puts the switch back in the OFF state.

That is, the switch state depends on the duration of the switch press. Any number of 'ON' states may be supported (although having more than around three is generally very confusing for the user).

This type of behaviour can be useful when, for example, setting time on a clock. Pressing the set switch will initially cause the displayed time to change slowly; sustained depressions will cause the displayed time to increase more rapidly.

Note: For reasons discussed in **SWITCH INTERFACE (SOFTWARE)**, in 'Safety and reliability', we are only concerned with push-button switches in these patterns: 'toggle' or 'latching' switches are not considered (and the use of such switches is not recommended).

SWITCH INTERFACE (SOFTWARE)

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you connect the port pin of an 8051 microcontroller to some form of mechanical switch (for example, a simple push-button switch or an electromechanical relay)?

Background

Consider the simple push-button switches illustrated in Figure 19.1.

Depressing this switch will, in either arrangement, cause a voltage change from approximately Vcc to 0V at the input port.

In an ideal world, this change in voltage would take the form illustrated in Figure 19.2 (top). In practice, all mechanical switch contacts *bounce* (that is, turn on and off, repeatedly, for a short period of time) after the switch is closed or opened. As a result, the actual input waveform looks more like that shown in Figure 19.2 (bottom). Usually, switches bounce for less than 20 ms: however large mechanical switches

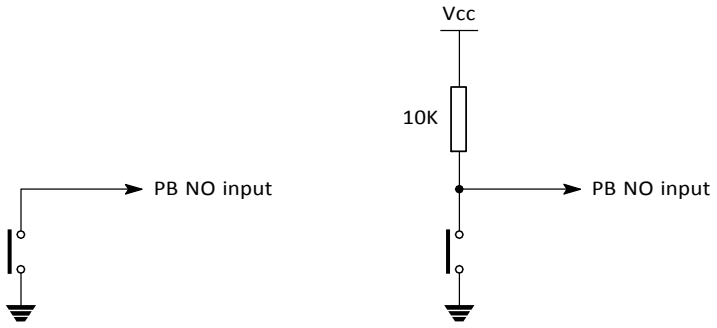


FIGURE 19.1 An example of a push-button ('normally open') switch input

[*Note*: The resistor-free arrangement [left] is appropriate where port pins have an internal pull resistor: this is usually the case on the 8051 family, with the exception of Port 0. Where there is no internal pull-up, the arrangement on the right must be used.]

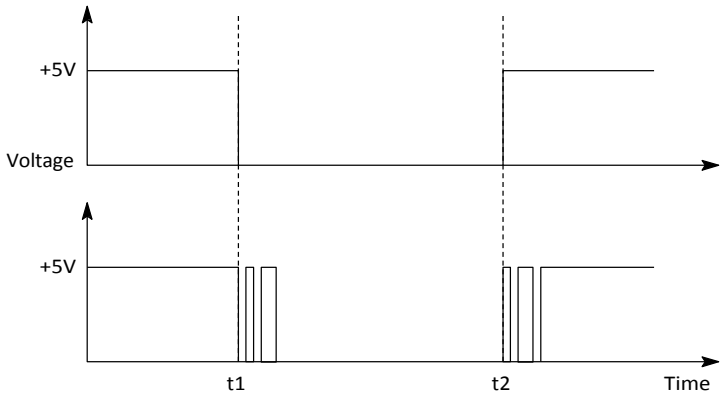


FIGURE 19.2 The voltage signal resulting from the switch shown in Figure 19.1

[*Note*: [Top] Idealized waveform resulting from a switch depressed at time t1 and released at time t2. [Bottom] Actual waveform showing leading edge bounce following switch depression and trailing edge bounce following switch release.]

exhibit bounce behaviour for 50 ms or more.

This bounce is equivalent to pressing an idealized switch multiple times. This causes various potential problems, not least:

- If we need to distinguish between single and multiple switch presses: for example, rather than reading 'A' from a keypad, we read 'AAAAA'.
- If we wish to count the number of switch presses.
- If we need to distinguish between a switch being depressed and being released: for example, if the switch is depressed once and then released some time later, the bounce will make it appear as if the switch has been pressed again.

Solution

Checking for a switch input is, in essence, straightforward:

1. We read the relevant port pin.
2. If we think we have detected a switch depression, we read the pin again (say) 20 ms later.
3. If the second reading confirms the first reading, then we assume the switch really has been depressed.

Note that the figure of '20 ms' will, of course, depend on the switch used: the data sheet of the switch will provide this information. If you have no data sheet, you can either experiment with different figures or measure directly using an oscilloscope.

We illustrate this basic procedure in the example.

SWITCH INTERFACE (HARDWARE)

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you create a very robust switch interface for use in a hostile (e.g. industrial, automotive) environment?

Background

Consider the following scenarios:

- Your embedded application is used in an industrial environment where high levels of electrostatic discharge are likely. How can you ensure that your device remains fully operational?
- Your automotive security application may be the subject of deliberate vandalism or damage. Specifically, reports on the WWW have revealed that thieves have found it possible to disable a similar security system by applying 12V from a car battery directly to one of the system switches. How can you ensure that your system is more robust?

In general, **SWITCH INTERFACE (SOFTWARE)** [page 399] describes techniques that are only suitable in 'safe' applications: in hostile environments, you need a more robust solution. This must be hardware based.

Solution

As noted in 'Background', creating a robust switch interface requires the use of off-chip hardware.

Traditionally, techniques involving J-K flip-flops, high-impedence CMOS gates or R-C integrators have all been used for switch debouncing: Huang (2000), for example, provides details of these techniques. In general, these approaches – while performing the debounce operation – provide only very limited protection, at best, again ESD and similar hazards. Because, as we saw in **SWITCH INTERFACE (SOFTWARE)** [page 399], the process of switch debouncing is almost trivial in a scheduled application, the cost of external hardware for switch debouncing alone cannot generally be justified.

More recently, several specialized ICs for protection and switch debouncing have appeared on the market. Of these, the Maxim 6816/6817/6818 family are a good example (Figure 19.7).

This is how Maxim describes these devices:

- The Max6816/Max6817/Max6818 are single, dual, and octal switch debouncers that provide clean interfacing of mechanical switches to digital systems. They accept one or more bouncing inputs from a mechanical switch and produce a clean digital output after a short, preset qualification delay. Both the switch opening bounce and the switch closing bounce are removed.
- Robust inputs can exceed power supplies by up to ±25V.
- ESD protection for input pins:

±15 kV	Human Body Model
±8 kV	IEC 1000-4-2, Contact Discharge
±15 kV	IEC 1000-4-2, Air-Gap Discharge

- Single-supply operation from +2.7V to +5.5V.
- Single (Max6816), dual (Max6817) and octal (Max6818) versions available.
- No external components required.
- 6 μA supply current.

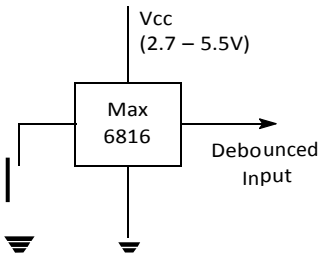


FIGURE 19.7 Typical application circuit for the Max6816

Hardware resource implications

Reading a debounced switch input imposes minimal loads on CPU and memory resources.

Reliability and safety issues

For the reasons discussed in 'Solution', this is a highly reliable method for creating a switch interface.

For additional reliability – particularly in the event of malicious damage – see **SWITCH INTERFACE (SOFTWARE)** [page 399] for discussions on the use of multi-pole switches.

Portability

These techniques are inherently portable.

Overall strengths and weaknesses

☺ **Greatly increased reliability (compared with software-only solutions) in hostile environments.**

☹ Increased costs and hardware complexity.

Related patterns and alternative solutions

See **SWITCH INTERFACE (SOFTWARE)** [page 399].

Example: Reading 8 switch inputs in a hostile environment

Figure 19.8 illustrates the use of a Max6818 to read the inputs from eight switches connected to Port 1 of an 8051 device. The results are reported on Port 2.

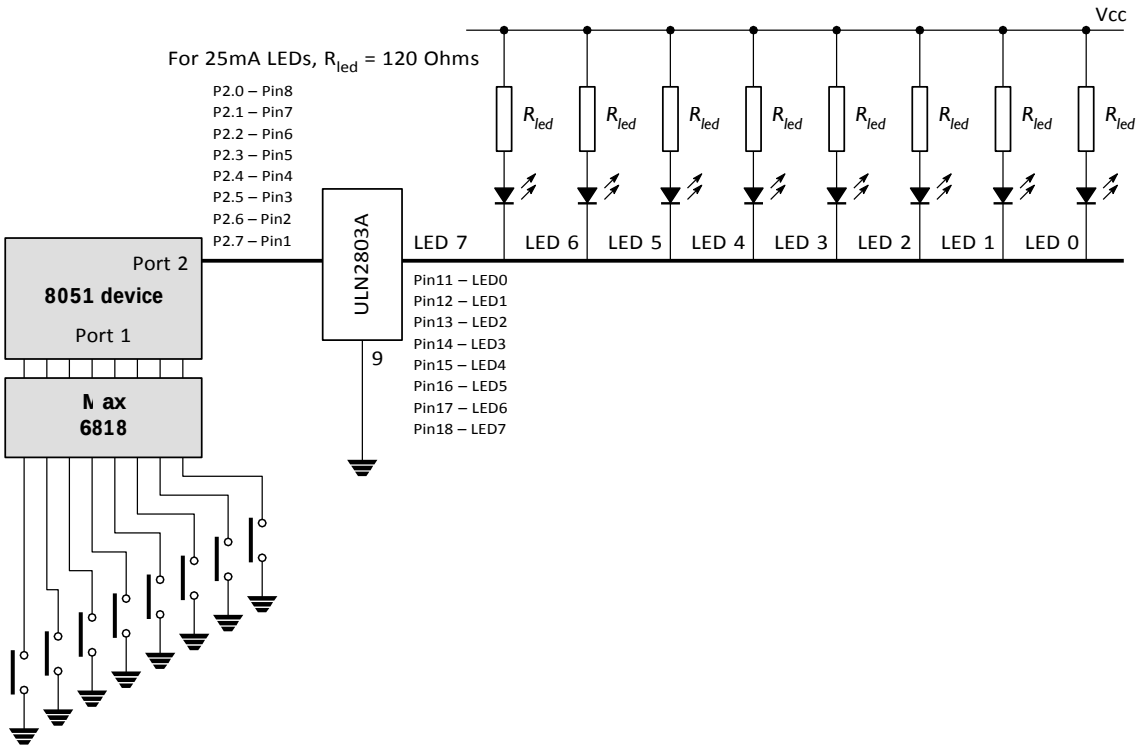


FIGURE 19.8 Using a Max6818 to debounce eight switches

The software requirements can be most easily met using a **SUPER LOOP** [page 162], as shown in Listing 19.5.

```
void main(void)
   {

   while(1)
      {
      P2 = P1;
      }

   }
```

Listing 19.5 A trivial Super Loop switch interface

ON - OFF SWITCH

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you obtain the behaviour illustrated in Figure 19.9 from a single, push-button switch connected to the port pin of a microcontroller?

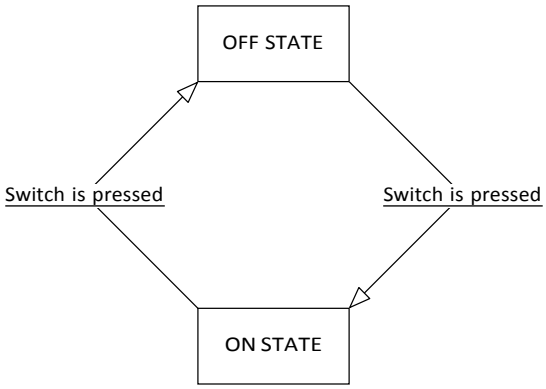


FIGURE 19.9 Creating an 'on-off' (latching) behaviour using a single push-button switch

Background

Consider a problem that can arise when we have a single switch used to turn on and off a piece of equipment (Figure 19.10).

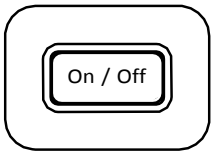


FIGURE 19.10 Illustrating the 'on-off' problem

The switch is intended to operate as follows:

- The user presses the switch to turn on a piece of equipment.
- The equipment operates as required.
- The user presses the switch again to turn off the equipment.

This seems very straightforward. However, suppose we are applying the basic approach to switch reading presented in **SWITCH INTERFACE (SOFTWARE)** [page 399], or **SWITCH INTERFACE (HARDWARE)** [page 410]. This can be summarized as follows:

1. We read the relevant port pin.
2. If we think we have detected a switch depression, we read the pin again 100 ms later.
3. If the second reading confirms the first reading, we assume the switch really has been depressed.

This is what can happen:

- The user presses the switch to turn on the piece of equipment.
- The switch is checked. It is depressed.
- The switch is checked (say) 100 ms later: the second check confirms the first. The equipment is turned on.
- The switch is checked 100 ms later. It is still depressed.
- The switch is checked 100 ms later: the second check confirms the first. The equipment is turned off again.
- And so on.

This behaviour arises because the user will generally wait until the equipment begins to work and will then remove their finger from the switch. In the best case scenario, a switch depression is likely to last about 500 ms. Unless we take action to prevent it, the equipment will 'flicker' on and off.

Solution

We can most simply create an on-off switch by adding a 'switch block' counter to the existing interface code. This works as follows:

1. Every time we find the switch has been pressed, we 'block' it for – say – 1 second.
2. While the switch is blocked, any changes in switch status are ignored: thus, for example, if the user keeps the switch depressed for half a second, this fact will be ignored.

The 'on-off' example that follows illustrates how this is achieved in practice.

Hardware resource implications

Reading a switch input imposes minimal loads on CPU and memory resources.

Reliability and safety issues

In hostile environments, this code should be based on **SWITCH INTERFACE (HARDWARE)** [page 410].

Note that, where safety is a primary concern, the blocking of switch inputs may not be appropriate. As an alternative, you can retain the basic switch-handling approach, but use two switches (Figure 19.11).

The use of two switches can make it easier to react quickly to changes in the inputs and may be safer under some circumstances.

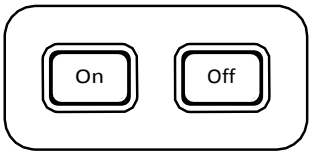


FIGURE 19.11 One approach to solving the 'on-off' problem

chapter 20

Keypad interfaces

Introduction

Keypads are a common component in embedded applications.

In this chapter, we consider how you can create reliable keypad-based user interfaces.

KEYPAD INTERFACE

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you connect a small keypad, similar to that illustrated in Figure 20.1, to your application?

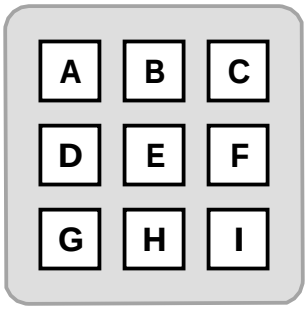


FIGURE 20.1 An example of a three-row x three-column keypad

Background

See Chapter 19 for background information on the reading of single switches.

Solution

Basics

We are concerned here with keypads made up of a matrix of switches, in an arrangement similar to that illustrated in Figure 20.2.

Some key points to note are as follows:

- The matrix arrangement is used to save port pins. If we have R rows and C columns of keys, we need R + C pins if we use a matrix arrangement and R × C pins if we use individual switches. If you need six or more keys, then the matrix arrangement requires fewer pins.

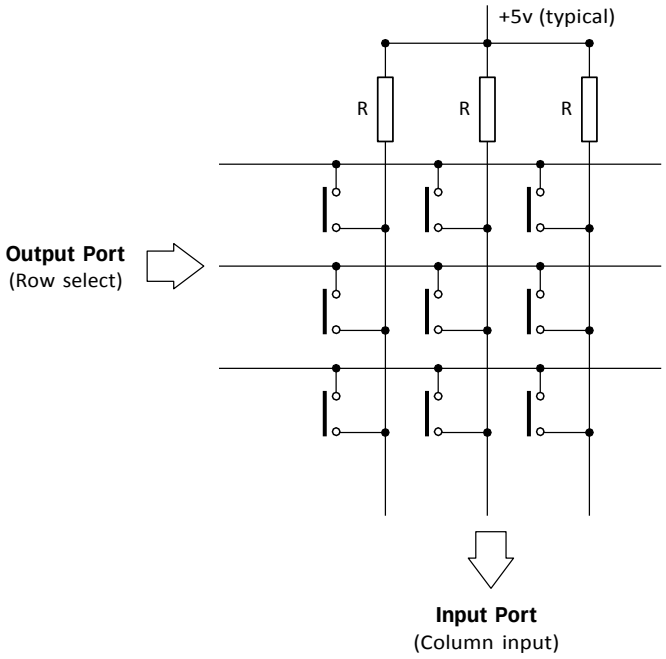


FIGURE 20.2 A schematic of a keypad interface

[*Note:* The pull-up resistors may be omitted, as usual, if the port has internal pull-ups.]

Many keypads have 12 keys ('0'–'9' plus two function keys – typically '#' and '*'). Using a matrix arrangement, this requires seven port pins.

- The keys may bounce, when pressed and released.
- The duration of the key press will generally be at least 500 ms.
- The keys will not generally be allowed to 'auto repeat': this can be very confusing for users.
- We may wish the user to be able to press one or more 'function keys' in combination with other keys.

Keypad scanning

At the heart of any keypad code is a scanning function: this will typically go through each column in turn and identify if any key in that column has been pressed.

Consider, for example, the keypad shown in Figure 20.3.

In Figure 20.3, the numbers adjacent to the rows and column indicate the port pins to which the keypad should be connected. Pins 0, 1 and 2 (the columns) will be referred to here as the output pins: these are written to during the scanning process. Pins 3, 4, 5 and 6 will be referred to as the input pins: these are read during the scanning process.

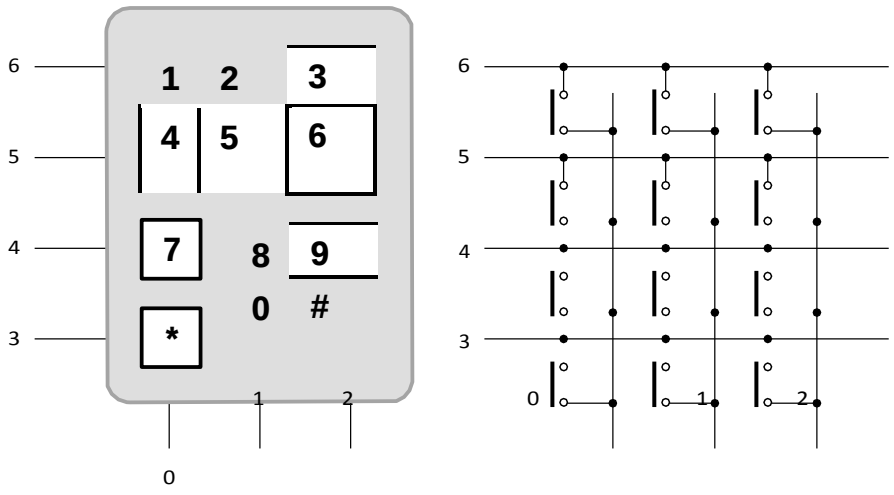


FIGURE 20.3 Typical keypad connections

[*Note:* The numbers adjacent to the keypad rows and columns represent the pin numbers on the port to which the keypad is connected. Note that no pull-up resistors will generally be needed: refer to Figure 20.2 for details.]

Suppose we wish to see if the key '1' is pressed. We can proceed as follows:

- We set the corresponding output pin (in this case, Pin 6) to a Logic 0 value and the remaining output pins to a Logic 1 value.
- We read the required input pin (in this case, Pin 0).

  If this pin is a Logic 1, then the key '1' is not pressed: the Logic 1 value is, instead, obtained via the pull-up resistor on the port pin.

  If this pin is at Logic 0, then the key is being pressed (subject to debounce considerations): the Logic 0 voltage reading results from the Logic 0 voltage output on Pin 6.
- We must repeat the reading (say) 200 ms later, to allow for switch bounce.

We need to repeat this process for every key. Listing 20.1 illustrates one way of performing this scanning for the whole keypad.

```
#define KEYPAD_PORT P2

sbit C1 = KEYPAD_PORT^0;
sbit C2 = KEYPAD_PORT^1;
sbit C3 = KEYPAD_PORT^2;

sbit R1 = KEYPAD_PORT^6;
sbit R2 = KEYPAD_PORT^5;
sbit R3 = KEYPAD_PORT^4;
sbit R4 = KEYPAD_PORT^3;
```

```
...
bit KEYPAD_Scan(char* const pKey, char* const pFuncKey)
   {
   static data char Old_Key;

   char Key = KEYPAD_NO_NEW_DATA;
   char Fn_key = (char) 0x00;

   C1 = 0; // Scanning column 1
      if (R1 == 0) Key = '1';
      if (R2 == 0) Key = '4';
      if (R3 == 0) Key = '7';
      if (R4 == 0) Fn_key = '*';
   C1 = 1;

   C2 = 0; // Scanning column 2
      if (R1 == 0) Key = '2';
      if (R2 == 0) Key = '5';
      if (R3 == 0) Key = '8';
      if (R4 == 0) Key = '0';
   C2 = 1;

   C3 = 0; // Scanning column 3
      if (R1 == 0) Key = '3';
      if (R2 == 0) Key = '6';
      if (R3 == 0) Key = '9';
      if (R4 == 0) Fn_key = '#';
   C3 = 1;

   if (Key == KEYPAD_NO_NEW_DATA)
      {
      // No key pressed (or just a function key)
      Old_Key = KEYPAD_NO_NEW_DATA;
      Last_valid_key_G = KEYPAD_NO_NEW_DATA;

      return 0;  // No new data
      }

   // A key has been pressed: debounce by checking twice
   if (Key == Old_Key)
      {
      // A valid (debounced) key press has been detected

      // Must be a new key to be valid - no 'auto repeat'
      if (Key != Last_valid_key_G)
         {
```

```
            // New key!
            *pKey = Key;
            Last_valid_key_G = Key;

            // Is the function key pressed too?
            if (Fn_key)
               {
               // Function key *is* pressed with another key
               *pFuncKey = Fn_key;
               }
            else
               {
               *pFuncKey = (char) 0x00;
               }

            return 1;
            }
         }

      // No new data
      Old_Key = Key;
      return 0;
      }
```

Listing 20.1 An example of code for keypad scanning

Function keys

Note that more than one key may be pressed at the same time. The 'function keys' ('#' and '*') in Listing 20.1 illustrate how we can make use of multiple key depressions.

The main code example presented with this pattern illustrate the use of function keys.

Buffer arrangements

In most scheduled keypad routines, it is useful to have a small buffer, so that key presses are not lost if the system is unable to process them immediately.

Numerous buffer arrangements are possible: the main code example presented with this pattern illustrates one possibility.

chapter 21

Multiplexed LED displays

Introduction

Many embedded applications contain user interfaces assembled from multi-segment LED displays.

In this chapter, we consider how such displays may be interfaced to the 8051 family of microcontrollers.

MX LED DISPLAY

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user-interface for your application.

Problem

How do you display information on one or more multi-segment LED displays?

Background

We consider some essential background material in this section.

What is a multi-segment LED?

Multiple LEDs are often arranged as multi-segment displays: combinations of eight segments (see Figure 21.1) and similar seven-segment displays (without a decimal point) are particularly common.

Such displays are arranged either as 'common cathode' or 'common anode' packages: the connection of the LEDs within each package is illustrated in Figure 21.2.

In either case, in addition to the common pin, we must provide appropriate signals to each of the LEDs in order to generate the digits we require (see 'Solution').

The required current per segment varies from about 2 mA (very small displays) to about 60 mA (very large displays, 100mm or more).

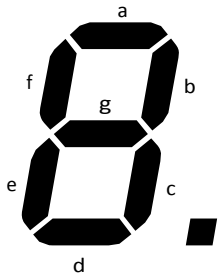


FIGURE 21.1 The segments of what is often referred to as a seven-segment display

[*Note*: that, as here, a decimal point will frequently also be included, so that this 'seven-segment' display actually has eight segments.]

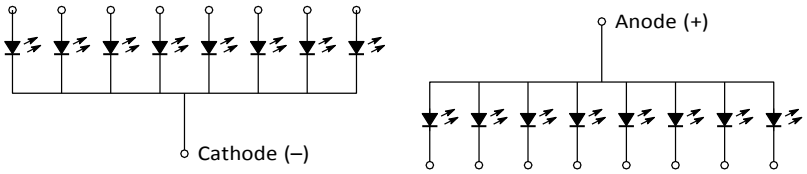


FIGURE 21.2 The internal arrangement of common cathode and common anode LED displays

[*Note*: that there are eight segments because we assume the presence of a decimal point.]

Basic hardware requirements (driving a single digit)

For reasons that we discussed in **IC BUFFER** [page 118], we cannot generally connect a multi-segment LED directly to a microcontroller port as illustrated in Figure 21.3. As you will recall, this is because the port cannot safely handle the required current (typically around 80 mA per digit).

In most cases, we require some form of buffer or driver IC between the port and the MS LED.

For small displays (around 9 mA per segment), the simple 240 and 241 buffers (see **IC BUFFER** [page 118]) are a cost-effective solution. Figure 21.4 shows a 74x241 (non-inverting buffer) used to drive a single, eight-segment LED display. Note that a 74x240 (inverting) buffer can be used as an alternative.

In most circumstances (such as the multiplexed displays we will focus on here), the basic 240 / 241 buffers have insufficient current capacity and an IC driver chip will generally be used. For example, Figure 21.5 shows a single common cathode LED digit connected to a single port via another octal buffer: the UDN2585A. As we discussed

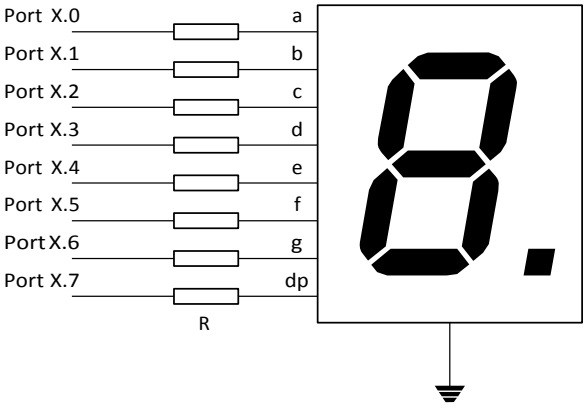
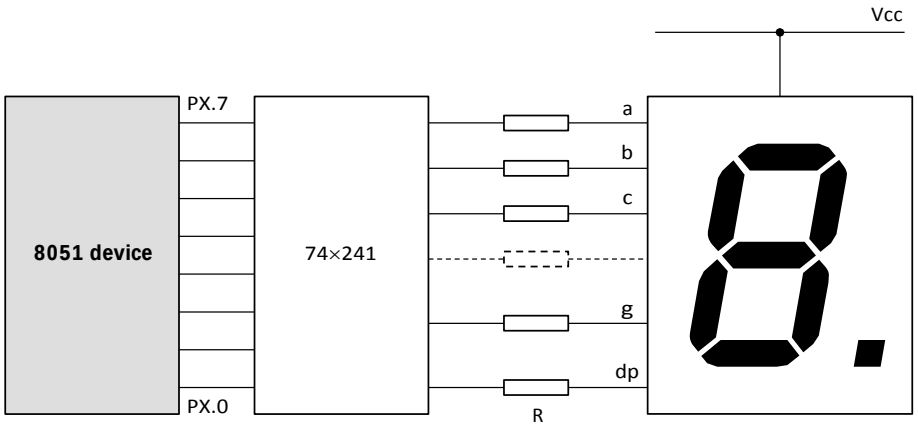


FIGURE 21.3 Attempting to drive a common anode display directly from a microcontroller port

[*Note*: in most cases, this will not work.]



$$R = \frac{V_{cc} - V_{diode}}{I_{diode}}$$

Note: All resistor values may not be the same
(the decimal point may require a different value)

FIGURE 21.4 Using a 74x241 to drive a multi-segment LED display

[*Note*: that, in some displays, a different resistor value may be required to drive the decimal point: check your data sheet. Note also that, as we discussed in **IC BUFFER** [page 118], we do not require pull-up resistors at the buffer output if CMOS logic is used.]

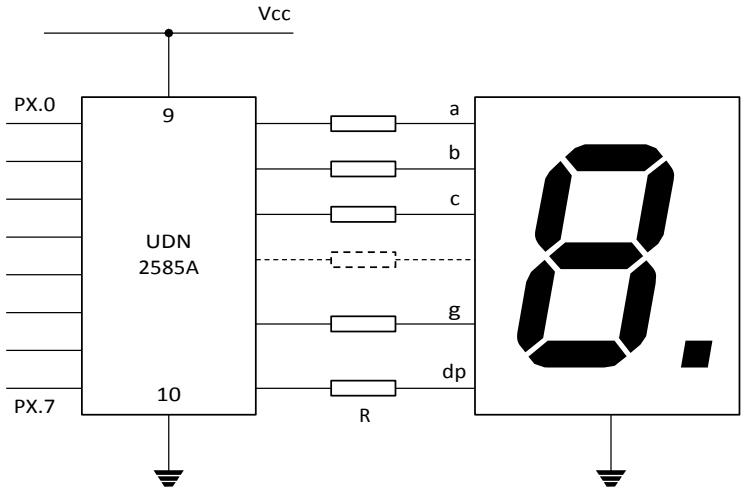


FIGURE 21.5 Using a UDN 2585A to drive an LED display

[*Note*: that this is an inverting (current source) buffer. Logic 0 on the input line will light the corresponding LED segment.]

in **IC DRIVER** [page 134], each of the (eight) channels in this device can simultaneously source up to 120 mA of current (at up to 25V): this is enough, for example, for even the largest of LED displays.

Here, the output of the 2585 in response to a Logic 0 input will be approximately Vcc. As a result, the resistor values are again calculated as shown in Figure 21.4.

Hardware for multiple digits

As discussed in 'Background', driving a single multi-segment display is straightforward. Suppose, however, that we need to drive four digits, perhaps for a simple digital clock. To do so using the approach discussed in 'Background' would require four spare ports. In many applications, you will not have four ports available. In addition, to use four ports often requires a separate driver (buffer) circuit and resistor pack for each digit: this can significantly increase the system cost.

The most common solution, discussed here, is to multiplex the displays. This means that (in this example) we drive each display for only a quarter of the time: as long as we cycle around all of the displays at between 20 and 50 Hz, the user will not generally notice that the displays are not being simultaneously driven. Even with a simple implementation, this basic approach allows us to drive up to eight LEDs (each with decimal points) using only two ports. More commonly, this allows us to drive four LED digits using 12 port pins. As in many 8051 applications (even those using external memory) Port 1 and most of Port 3 are available, this approach can be used in many applications.

A simple circuit suitable for creating a cost-effective multiplexed display is shown in Figure 21.6. Here, the information required to drive each digit is supplied on Port X, while the selection of the digits is controlled by Port Y.

Note that Port Y must control (sink) the current coming from eight individual LEDs: this may be up to around 140 mA, even for a collection of small displays and

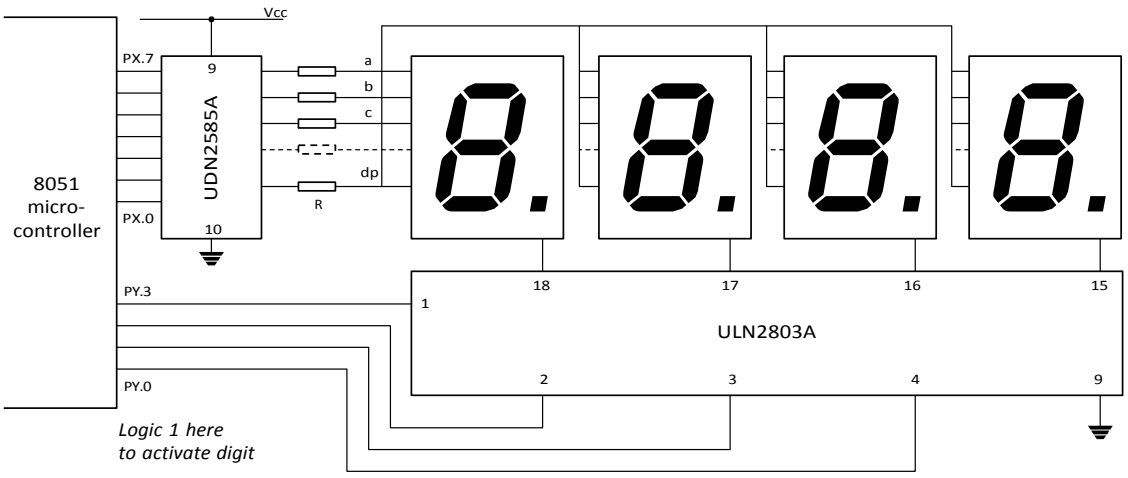


FIGURE 21.6 Multiplexing four (common cathode) LED modules

can approach 500 mA for large (100 mm) displays. As a result, we cannot directly drive the displays from a port, but must use, for example, a ULN 2003 or 2803 as a (sink) buffer. Note that you can replace the 2003 / 2803 (or equivalent) with four NPN transistors (e.g. 2N2222), but this is seldom cost-effective.

You will also need a buffer on Port X to source sufficient current: the UDN 2585A is, again, a good choice here. Use of such a source buffer is often essential, since – to achieve a bright display – we usually supply up to four times the normal display current, for a quarter of the time (thus keeping the average current at the required value). Your microcontroller will have a greatly reduced life if you try to provide this current from the naked chip.

Again, the values of R in Figure 21.6 are calculated as shown in Figure 21.4. In this case, it can be helpful to use a 'resistor pack' to implement the circuit: such packs contain seven or eight resistors, in a standard (dual in line: DIL or similar) package.

Solution

We consider here both the software codes required to display digits on the display and the refresh rates needed for flicker-free operation.

Basic software

The software to control the LED digits in arrangements like those shown in Figures 21.4 and 21.5 is easy to write. If, for example, we connect the display to Port 3, we can control it by simply writing:

```
P3 = code;
```

Here, 'code' is a numerical representation of the required segment activations. The data required to display the digits 0–9, with or without a following decimal point, are given in Listing 21.1.

```
/*-------------------------------------------------------------------*-

   Connections

   DP   G    F    E    D    C    B    A    =   LED display pins
   |    |    |    |    |    |    |    |
  x.7  x.6  x.5  x.4  x.3  x.2  x.1  x.0    =   Port pins

  LED codes (NB - positive logic assumed here)

  0  = abcdef  => 00111111   = 0x3F
  1  = bc      => 00000110   = 0x06
  2  = abdeg   => 01011011   = 0x5B
  3  = abcdg   => 01001111   = 0x4F
  4  = bcfg    => 01100110   = 0x66
  5  = acdfg   => 01101101   = 0x6D
  6  = acdefg  => 01111101   = 0x7D
```

```
  7  = abc     => 00000111   = 0x07
  8  = abcdefg => 01111111   = 0x7F
  9  = abcdfg  => 01101111   = 0x6F

 To display decimal point, add 10 (decimal) to the above values

-*--------------------------------------------------------------------*/

// Lookup table - stored in code area (ROM)

tByte code LED_Table_G[20] =
// 0     1     2     3     4     5     6     7     8     9
{0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F,
// 0     1     2     3     4     5     6     7     8     9
 0xBF, 0x86, 0xDB, 0xCF, 0xE6, 0xED, 0xFD, 0x87, 0xFF, 0xEF};

// ---------------------------------------------------------------
```

Listing 21.1 Codes that may be used to display data on multi-segment LED displays

Controlling more than one digit per port (Software issues)

We need to update all the displays at a rate of approximately 50 Hz, if we can: if necessary, rates as low as 20 Hz will do. If we have N LED modules, each will be active for approximately 1/N of the time. Note that, if you choose to drive the displays at high current values, the timing needs to be reasonably accurate, otherwise you may reduce the life of the display and driver components by exposing them to excessively high currents for sustained periods.

Overall, for a typical four-module display, we aim to update one module at least every 5 milliseconds. This is easy to achieve using a suitable scheduler, as we demonstrate in the examples that follow.

Hardware resource implications

All the examples here require the use of a scheduler. The main resource implication is that, to update (say) four digits, you need a scheduler with a tick interval of around 5 ms. This is not usually a limiting factor. Overall, the LED update code will consume only around 1% of the CPU time in a typical application, since – although frequent – it is a simple and fast operation.

The memory requirements are minimal.

A substantial number of port pins are required.

Reliability and safety implications

LEDs are not visible in bright light and, as such, must be used with care if they are displaying safety-related information.

For multiplexed displays, you must ensure that the application (or scheduler) cannot become locked: if it does, the display will very quickly be destroyed by high current values. Thus, if a poorly designed task blocks the scheduler, even for a few seconds, and delays a display update (usually scheduled around 20 to 50 Hz), you will probably destroy the display. Whatever happens, you need to ensure that such an event will not impact on the general operation of the microcontroller and, therefore, completely destroy the application.

chapter 22

Controlling LCD panels

Introduction

We considered the creation of LED-based user interfaces in Chapter 21. Here we are concerned with the use of liquid crystal displays (LCDs) in such interfaces.

Unlike LEDs, LCDs are based on passive display technology: this means that LCDs control the passage of light rather than emitting light. This fact directly contributes to the low power consumption of these devices: large (5V) panels require up to 5 mA: a total power consumption of up to 25 mW, excluding any backlight. Small panels consume around half this power. Since a *single* LED has a very similar power consumption, use of LCD displays in battery-powered embedded systems is particularly popular.

While various types of LCD display are available, they can be divided into two basic groups: graphics displays and text displays. Notebook (and increasingly desktop) PCs use sophisticated graphics displays, made up of many thousands of individual 'picture elements' (pixels). Such displays are expensive in their own right and generally require large amounts of memory (typically several megabytes) and powerful processors for efficient operation. As we have seen, the type of embedded devices we are concerned with in this book do not usually justify this level of expense. Instead, we will be concerned here with small displays intended primarily to display text.

Various types of LCD-based character displays are available. These are typically arranged as one, two or four lines of between 16 and 40 characters. Inevitably, the larger displays are more expensive and consume more power. Each LCD character is usually a 5x8 matrix of dots: less commonly, a 5x11 matrix is also used: note that, in each case, the characters themselves are 5x7 and 5x10 pixels in size, with the bottom line being reserved for the cursor. To generate characters on such displays would tend to consume a large percentage of the available CPU time (and most of the ports or address space) on most embedded processors. As a result, most LCD panels include an on-board controller to deal with this: this is generally a variant on the Hitachi

HD44780. Displays based on this popular controller all have very similar hardware interfaces and will display the same mixture of English (or Japanese) characters.

We focus on LCD panels based on this 'standard' controller in the pattern **LCD CHARACTER PANEL** presented in this chapter.

Please note that much of the background material presented here is adapted from the Hitachi HD44780 data sheet.

LCD CHARACTER PANEL

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.
- You are creating the user interface for your application.

Problem

How do you connect an LCD-based 'character panel' display to your embedded processor and control it efficiently using a high-level programming language?

Background

As outlined at the start of this chapter, we are concerned here with LCD character panels based on the HD44780 microcontroller.

Key HD44780 components

The HD44780 contains its own reset circuitry, memory and so forth: it contains several important components (Figure 22.1), which may be accessed and controlled from an attached microcontroller or microprocessor. We will discuss some of the key components in the sections that follow.

As shown in Figure 22.1, the interface between the microcontroller and the HD44780 consists of five sets of signals. A summary of each of these signals is given in Table 22.1.

DD RAM

At the heart of the HD44780 is the DD RAM: the 'Display Data' RAM. This stores display data represented in 8-bit character codes. Its capacity is 80 characters; as a result the maximum display sizes are 20 charactersx4 lines or 40 charactersx2 lines.

Use of the HD44780 involves transferring data (via the 4-bit or 8-bit data bus) into the DD RAM. The HD44780 will then refresh the display as required using these data.

Characters that can be stored and displayed via DD RAM include most of the core (displayable) characters from the ASCII table, with only minor changes (Table 22.2): this means that, in general, you can simply send character data to the display and obtain the expected outputs.

Note that *none* of these characters uses the bottom line of the display (the cursor line): as a result, low-case characters with 'tails' (g, j, p, q, y) will appear higher than normal on the display.

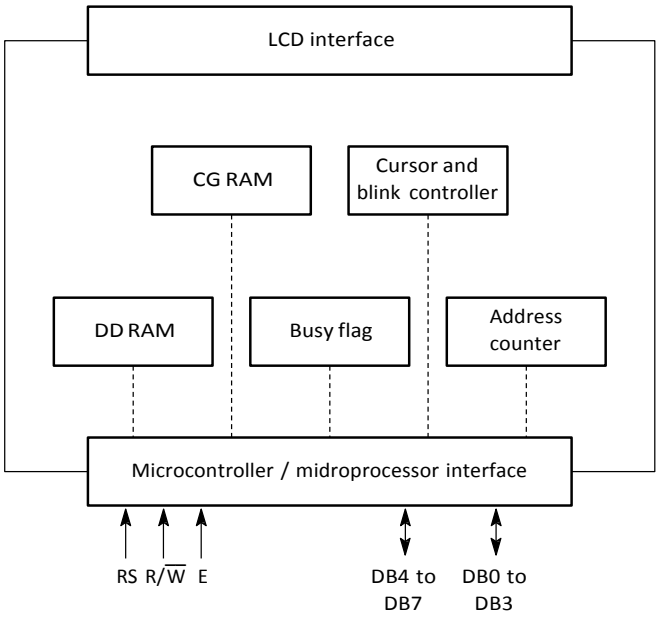


FIGURE 22.1 Key components in the Hitachi HD44780.

TABLE 22.1 The HD44780 interface

Signal	Number of lines	Input and / or output?	Function
RS	1	I	Selects instruction register (0) or data register (1)
R /W	1	I	Selects read (1) or write (0)
E	1	I	Starts data read/write
DB4 to DB7	4	I/O	Used for data transfer between the 8051 and the HD44780. DB7 can be used as a busy flag (not used in the code samples here)
DB0 to DB3	4	I/O	Used for data transfer and receive between the 8051 and the HD44780. These pins are not used during 4-bit operation (not used in the code samples here)

TABLE 22.2 The basic character set in the HD44780

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2X		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3X	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4X	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5X	P	Q	R	S	T	U	V	W	X	Y	Z	[	¥	]	^	_
6X	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7X	p	q	r	s	t	u	v	w	x	y	z	{	|	}	→	←

[Most of this follows the ASCII code: however, note the Yen symbol (Character 0x5C) in place of '\', and that 0x7E and 0x7F are left and right arrows, respectively. UK users should also note that there is no '£' symbol (character 0x23 is '#').]

While the character codes are straightforward, knowing which address to write the data to is slightly less straightforward. There are a number of different configurations of display available (including 16 characters x1 line, 16x2, 20x2, 20x4, 24x1, 24x2, 40x1, 40x2). The memory locations for each of the segments on these displays are shown in Table 22.3.

TABLE 22.3 Memory locations for the start of each line in various possible HD44780 LCD displays

#characters	#lines	Top line	Second line	Third line	Fourth line
16 (8)	1 (2)	0×00 – 0×07	0×40 – 0×47	-	-
16	2	0×00 – 0×0F	0×40 – 0×4F	-	-
16	4	0×00 – 0×0F	0×40 – 0×4F	0×10 – 0×1F	0×50 – 0×5F
20	2	0×00 – 0×13	0×40 – 0×53	-	-
20	4	0×00 – 0×13	0×40 – 0×53	0×14 – 0×27	0×54 – ×x67
24	1	0×00 – 0×17	-	-	-
24	2	0×00 – 0×17	0×40 – 0×58	-	-
40 (20)	1 (2)	0×00 – 0×13	0×40 – 0×53	-	-
40	2	0×00 – 0×27	0×40 – 0×67	-	-

CG RAM

The CG RAM is the 'Character Generator' RAM: this can be used to allow the creation and display of user-defined characters. This facility is helpful if you wish to display characters outside the 'standard' range, such as a '£' sign.

We demonstrate how to use CG RAM in an example that follows.

Registers

The HD44780 has two 8-bit registers, an instruction register (IR) and a data register (DR):

- The IR stores instruction codes, such as display clear and cursor shift and address information for DDRAM and character generator CGRAM.
- The DR temporarily stores data to be written into DDRAM or CGRAM. Data written into the DR from the 8051 are automatically written into DDRAM or CGRAM by an internal operation.

Busy flag (BF)

When the busy flag is 1, the HD44780 is performing an internal operation and further instructions or data will not be accepted. When RS = 0 and R/W = 1 (Table 22.4), the busy flag is output to DB7.

TABLE 22.4 Register selection

RS	R/W	Operation
0	0	IR write as an internal operation (display clear etc.)
0	1	Read busy flag (DB7) and address counter (DB0 to DB6)
1	0	DR write as an internal operation (DR to DDRAM or CGRAM)
1	1	DR read as an internal operation (DDRAM or CGRAM to DR)

Cursor/blink control circuit

As the name suggests, the cursor/blink control circuit generates the cursor or character blinking. The cursor or the blinking will appear with the digit located at the display data RAM (DDRAM) address set in the address counter (AC). For example, when the address counter is 0x08, the cursor position is displayed at DDRAM address 0x08.

Address counter (AC)

The address counter (AC) assigns addresses to both DDRAM and CGRAM. When an address of an instruction is written into the IR, the address information is sent from the IR to the AC.

Selection of either DDRAM or CGRAM is also determined concurrently by the instruction.

After writing to DDRAM or CGRAM, the AC is automatically incremented by 1. The AC contents are then output to DB0 to DB6 when RS = 0 and R/W = 1 (Table 22.4).

Solution

As discussed in 'Background', we will consider only LCD devices based on the Hitachi HD44780 controller.

Hardware

The HD44780 can send data in either two 4-bit operations or one 8-bit operation. This feature was probably originally intended to allow connection to both 4-bit and 8-bit microcontrollers. However, even with 8-bit microcontrollers (such as the 8051) the 4-bit interface is the most commonly applied, since:

1 The 4-bit interface is sufficiently fast for most applications (a single character is transferred in under 0.1 ms).

2 It saves four port pins.

We will use a 4-bit interface in our examples.

Despite its name the '4-bit' interface actually requires six port pins: four for the data bus and (usually) two additional port pins for the control lines (Figure 22.2).

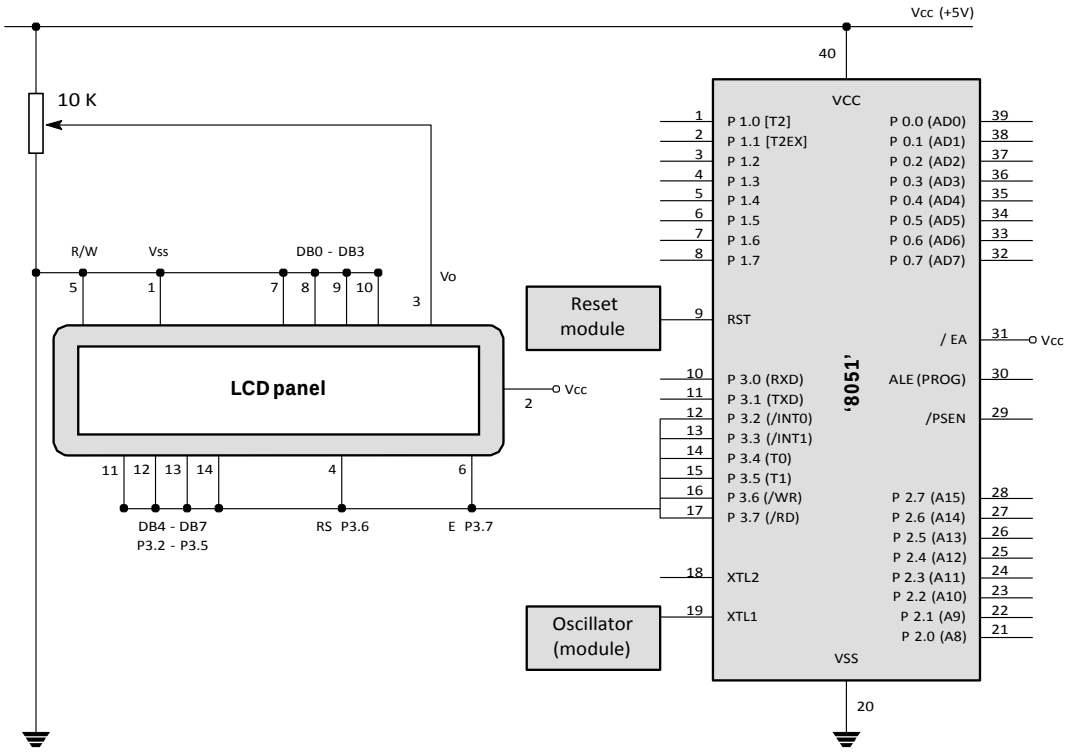


FIGURE 22.2 A 4-bit interface between an HD44780-based LCD panel and an 8051 microcontroller

In addition to the ground and power connections, you also need to connect the contrast adjustment pin (Vo). We have successfully used many LCD display by pulling this pin to ground. However, the recommended contrast adjustment takes the form of a potentiometer, connected as shown in Figure 22.3.

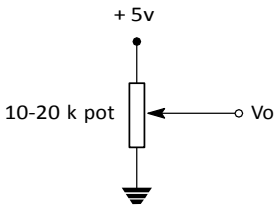


FIGURE 22.3 The recommended contrast adjustment connection

[*Note*: that, in many cases, a pulling Vo to ground will also work.]

Back lighting

If the module has a backlight, this will greatly increase the power consumption: check your data sheet for details. Before assuming that you require a backlight, try some of the modern displays now available: recent devices have greatly improved contrast and visibility.

The data bus

The data bus (4-bit) needs to be connected to four port pins with internal pull-up resistors. If using a port (e.g. Port 0) without internal pull-ups, add external 10K pull-ups (to Vcc).

Software

Complete software libraries are given in the examples that follow.

Note that in these examples – as in the RS-232 library – we have implemented the library as a **MULTI-STAGE TASK**: each character is written to a buffer and updates of the LCD display itself are carried out with a scheduled 'update' function.

Note also that these examples illustrate the use of user-defined characters.

part e

Using serial peripherals

In Part E we consider how two powerful and influential serial communication protocols ('I^2C' and 'SPI') may be utilized in applications with a time-triggered architecture.

Use of these protocols has two main advantages:

- They are designed to allow microcontrollers to be linked to a wide range of different peripherals – memory, displays, ADCs and similar devices – without requiring the use of large numbers of port pins.
- A common set of software code may be used with all SPI peripherals (for example), reducing the development effort required.

In Chapter 23, we consider the I^2C bus, developed by Philips. I^2C is a simple protocol, and may be easily generated in software. This allows the full range of 8051 devices (including Small 8051s, with very few spare port pins) to communicate with a wide range of peripheral devices.

In Chapter 24, we consider the SPI bus, developed by Motorola. Increasing numbers of 'Standard' and 'Extended' 8051 devices have hardware support for SPI and we will make use of these facilities in Chapter 24.

chapter 23

Using 'I²C' peripherals

Introduction

The I²C bus is a two-wire serial communication bus, originally introduced by Philips in 1992. I²C is now supported by a wide range of semiconductor manufacturers, with the result that many different peripheral devices (LCDs, LED displays, EEPROMs, ADC and DAC devices and so on) can be connected to a microcontroller without using large numbers of port pins.

The pattern in this chapter addresses the following problem:

- Should you use the I²C protocol to link your microcontroller to peripheral devices and, if so, how do you do so?

I²C PERIPHERAL

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, based on a scheduler.
- The microcontroller in your application will be interfaced to one or more peripherals, such as a keypad, EEPROM, digital-to-analog converter, or similar device.

Problem

Should you use the I²C protocol to link your microcontroller to peripheral devices and, if so, how do you do so?

Background

There are five key features of I²C as far as the developer of embedded applications is concerned:

- I²C is a protocol designed to allow microcontrollers to be linked to a wide range of different peripherals – memory, displays, ADCs and similar devices – and requires only two port pins to connect to (typically) up to 20 peripherals.
- There are many I²C peripherals available for purchase 'off the shelf'.
- I²C is a simple protocol and may be easily generated in software. This allows *all* 8051 devices to communicate with a wide range of peripheral devices.
- A common set of software code may be used with all I²C peripherals.
- I²C is fast enough (even when generated in software) to be compatible with time-triggered architectures. Typical data transfer rates will be up to 1,000 bytes / second (with a 1 ms scheduler tick).

We provide some background necessary for understanding and using the I²C bus in this section. Please note that much of the material here is adapted from the Philips (1998) I²C specification. For further details, the reader is referred to this document, a copy of which can be obtained from the Philips WWW site.[1]

Hardware

We begin by considering essential I²C hardware features.

1. www.philips.com

The bus

In the two-wire I²C bus the serial data (SDA) and serial clock (SCL) lines carry the information between the various devices (Figure 23.1). Due to the variety of different technologies (CMOS, NMOS, bipolar) used to create devices which can be connected to the I²C bus, the levels of the logical '0' (LOW) and '1' (HIGH) are not fixed and depend on the system supply voltage.

When the bus is free, both SCL and SDA lines are HIGH.

Both SDA and SCL are bidirectional lines. The output stages of devices connected to the bus should be open drain (open collector) in order to match the requirements of this protocol. This will often mean using pins on Port 0 of an 8051 device (but see following box).

The number of devices that can be connected to the I²C bus is limited only by the maximum bus load capacitance of 400 pF: in the absence of suitable information, assume that each peripheral device and its wiring contributes a total of 20 pF to the total capacitance.

Each of the lines is connected to the Vcc supply via a common pull-up resistor. If we assume a capacitance of 20 pF per device, the required value of resistor is determined by the maximum rise time in the I²C specification (1,000 ns in Standard I²C). This can be shown to translate into the following:

$$R = \frac{50}{d}$$

where: R is the required resistance (KΩ)
d is the number devices on the bus

Note that, in various applications, we have successfully used 8051 devices with 'conventional' (not open-drain) pins for I²C communications, without the use of external pull-up resistors.

If you do the same, we recommend that you do so only where a small number of peripherals are used and that you test the resulting application thoroughly.

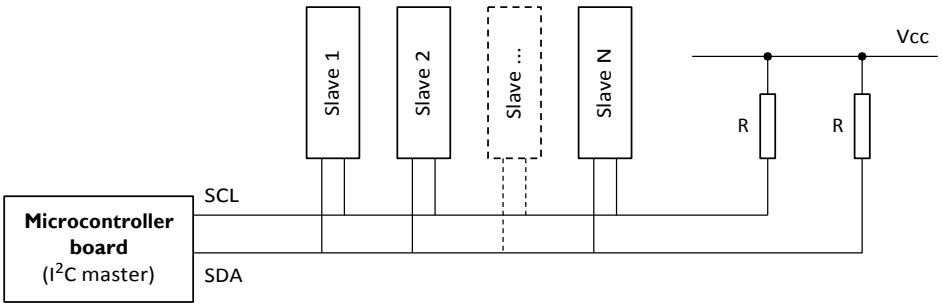


FIGURE 23.1 Using the I²C bus

Device addresses

In I²C, the communication is address based: that is, each device on the bus has a unique address and messages can be directed from anywhere on the bus to the device with a particular address.

The device addresses are ten-bits long in the latest version of this standard (seven bits in the original), allowing more than 1,000 different addresses to be used. Note that devices with 7-bit and 10-bit addresses may be used on the same bus.

While the facility for 7-bit or 10-bit addresses are included in the I²C specification, in most practical cases, the address of a device is partially hard wired into the device.

Consider, for example, a useful I²C peripheral, the Atmel 24C64 serial EEPROM.[2] This device provides 8192 × 8 bits of non-volatile data storage, with data retention for 100 years. As such it is ideal, for example, for storing small amounts of data in monitoring applications or for storing system passwords in other devices.

The pinout of the 24C64 is given in Figure 23.2. Note from the figure that there are only three address lines, which will usually be pulled to Vcc or ground to set the required device address. Assuming each device is given a unique address, up to eight 24C64s may be connected to one I²C bus, rather than 128 (2^7) or 1024 (2^{10}) as is suggested by the I²C standard.

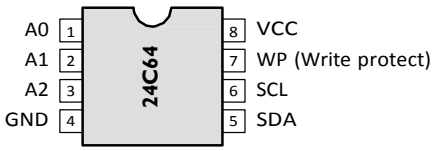


FIGURE 23.2 Pin configuration for the Atmel 24C64 serial EEPROM

As far as the bus is concerned, the 24C64 has, in fact, a 7-bit address. However, the most significant four bits are 'hard wired' into the device. These bits are always '1010' (0xA0), which is a code common to many serial EEPROMs. The full device address is therefore '1 0 1 0 A2 A1 A0'.

The hard-wiring of part of the device address makes practical sense. Few applications require more than eight serial EEPROMs on a single bus, particularly since quite large capacity memory devices are now available with an identical interface. If the full device address needed to be coded by the user for each device, the size of each chip would have to be increased to allow for (at least) four more pins (Figure 23.3).

Very similar schemes are used in other I²C devices. For example, the Dallas 1621 temperature sensor[3] is a useful and inexpensive component that provides a temperature reading, in Celsius, over an I²C bus (Figure 23.4).

---

2. A complete example of an I²C link from an 8051 to a 24C64 device is given on page 510.
3. A complete example of an I²C link from an 8051 to a DS1621 temperature sensor is given on page 515.

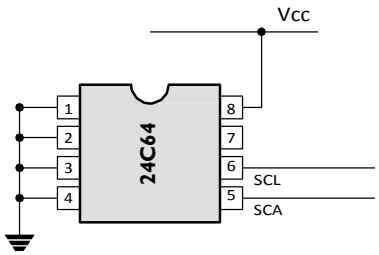


FIGURE 23.3 A 24C64 with device address '1010000'

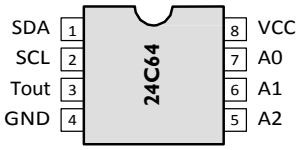


FIGURE 23.4 Pin configuration for the Dallas DS1621 (I²C) temperature sensor

Here, again, only three least significant bits of the address are user adjustable. In this case, the full address of the device is '1 0 0 1 A2 A1 A0'. The hard-wired part of this address (0x90) is shared by many DAC devices.

The I²C protocol: Masters and Slaves

Each device on an I²C bus can operate as either a transmitter or receiver of data. For example, an LCD driver will generally act as a receiver, whereas a memory device can act as both a receiver and a transmitter. In addition to transmitters and receivers, devices can also be considered as Masters or Slaves. A Master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered to be a Slave.

The I²C bus has a multi-Master capability: this means that more than one device capable of controlling the bus can be connected to it. This might be the useful, for example, if I²C were used to transfer data between two microcontrollers.

> Please note that in this pattern we are concerned only with I²C applications involving a single microcontroller (the device Master) and one or more Slave peripherals.

Generation of clock signals on the I²C bus is always the responsibility of a Master device; the Master generates its own clock signals when transferring data on the bus. In a single-Master system, bus clock signals from a Master can only be altered when they are stretched by a slow Slave device holding down the clock line.

Performing data transfers

We now consider how data are transferred between devices. Consider, for example, that we wish to read a temperature value from a DS1621 temperature sensor discussed earlier: we would do so as follows:

1. Generate a START condition
2. Send DS1621 device address (with WRITE access request)
3. Make sure that the Slave (DS1621) generates an ACKNOWLEDGE bit
4. Send the command 'Read Temperature' (0xAA)
5. Make sure that the Slave (DS1621) generates an ACKNOWLEDGE bit
6. Generate another START condition
7. Send DS1621 device address (with READ access request this time)
8. Make sure that the Slave (DS1621) generates an ACKNOWLEDGE bit.
9. Receive the first (most significant) byte of temperature data from the I^2C bus
10. Perform a MASTER – ACKNOWLEDGE
11. Receive the second byte of temperature data from the I^2C bus
12. Perform a MASTER – NOT ACKNOWLEDGE
13. Generate a STOP condition

Although this process appears at first glance rather complex, it contains all the core I^2C routines that are required for communication with any I^2C peripheral. As a result, when we have considered this process, you will be in a position to develop I^2C interfaces for a very wide range of devices.

To understand the discussions that follow, it will be helpful to refer to Figure 23.5.

We begin by considering the START and STOP conditions.

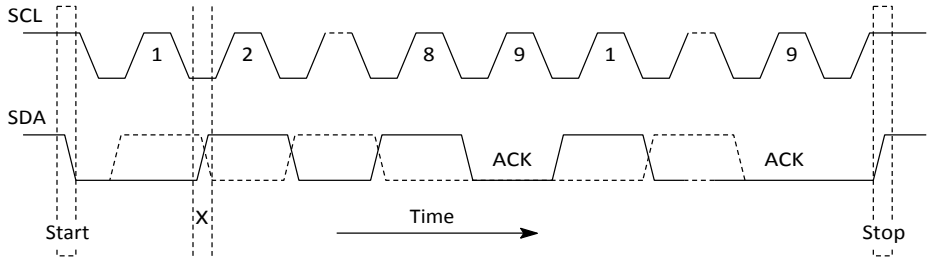


FIGURE 23.5 The basic format of I^2C communications

[*Note*: that one clock pulse is generated for each data bit transferred. Note also that the data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW.]

START and STOP conditions

I²C communications begin with a START condition. Further START conditions may be generated within the message, which will end with a STOP condition.

A START condition is represented as follows (see Figure 23.5): a HIGH to LOW transition on the SDA line while SCL is HIGH.

A STOP condition is represented as follows (see Figure 23.5): a LOW to HIGH transition on the SDA line while SCL is HIGH.

START and STOP conditions are always generated by the Master.

Byte format

Every byte put on the SDA line must be eight bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte has to be followed by an ACKNOWLEDGE bit (see following sections).

Data are transferred with the most significant bit first. If a Slave cannot receive or transmit another complete byte of data until it has performed some (internal) operation, for example updating memory, it can hold the SCL line LOW to force the Master into a wait state. Data transfer then continues when the Slave is ready for another byte of data and releases SCL.

Slave 'ACKNOWLEDGE' and 'NOT ACKNOWLEDGE'

In many situations the Master (the microcontroller in our case) will be sending data to the Slave (for example, an EEPROM). In these cases, the 'Slave-receiver' must generate an 'Slave-receiver ACKNOWLEDGE' signal when it has received a byte of data.

The ACKNOWLEDGE-related clock pulse is always generated by the Master. The transmitter releases the SDA line (HIGH) during the ACKNOWLEDGE clock pulse. The Slave-receiver must pull down the SDA line during the ACKNOWLEDGE clock pulse so that it remains stable LOW during the HIGH period of this clock pulse.

Usually, an active Slave-receiver is obliged to generate an ACKNOWLEDGE after each byte has been received.

When a Slave does not acknowledge the Slave address (for example, it is unable to receive or transmit because it is performing some internal operation), the data line must be left HIGH by the Slave. The Master can then generate either a STOP condition to abort the transfer or a repeated START condition to start a new transfer.

If a Slave-receiver does acknowledge the Slave address but, some time later in the transfer cannot receive any more data bytes, the Master must again abort the transfer. This is indicated by the Slave generating a 'NOT ACKNOWLEDGE' response on the first byte to follow: this means that the Slave leaves the data line HIGH during the ACKNOWLEDGE clock pulse generated by the Master. The Master will then generate a STOP (or a repeated START) condition and – probably after a delay – will attempt the transmission again.

Master-receiver 'ACKNOWLEDGE' and 'NOT ACKNOWLEDGE'

In some situations the Master (the microcontroller in our case) will be receiving data from the Slave (for example, the temperature sensor). In some cases, the 'Master-receiver' must generate a 'Master-receiver ACKNOWLEDGE' signal when it has received a byte of data. However, at the end of the data transfer (when the Master-receiver has received the last byte of data it requires), it must signal the end of data to the Slave-transmitter by not generating an ACKNOWLEDGE on the last byte that was clocked out of the Slave. The Slave-transmitter must release the data line to allow the Master to generate a STOP or repeated START condition.

Synchronization

All Masters generate their own clock on the SCL line to transfer messages on the I^2C bus. Data are only valid during the HIGH period of the clock. A defined clock is therefore needed for the bit-by-bit arbitration procedure to take place.

Clock synchronization is performed using the wired AND connection of I^2C interfaces to the SCL line. This means that a HIGH to LOW transition on the SCL line will cause the devices concerned to start counting off their LOW period and, once a device clock has gone LOW, it will hold the SCL line in that state until the clock HIGH state is reached. However, the LOW to HIGH transition of this clock may not change the state of the SCL line if another clock is still within its LOW period. The SCL line will therefore be held LOW by the device with the longest LOW period. Devices with shorter LOW periods enter a HIGH wait-state during this time.

When all devices concerned have counted off their LOW period, the clock line will be released and go HIGH. There will then be no difference between the device clocks and the state of the SCL line and all the devices will start counting their HIGH periods. The first device to complete its HIGH period will again pull the SCL line LOW. In this way, a synchronized SCL clock is generated with its LOW period determined by the device with the longest clock LOW period and its HIGH period determined by the one with the shortest clock HIGH period.

Further details

A complete technical specification for the I^2C bus will be found on the Philips WWW site.

Solution

Should you use I^2C?

In order to determine whether use of an I^2C bus is appropriate in your time-triggered application, we consider some key questions that should be asked when considering the use of any communications protocol or related technique.

Main application areas

The I²C bus was designed mainly to allow the interconnection of components within a single application. Although the bus may be used, for example, to connect processors (usually microcontrollers) to one another or to other computer systems, its main application area is in the connection of standard peripheral devices, such as LCD panels or EEPROMs, or 'smart' components in consumer applications (e.g. TV tuners, PLL synthesizers, video processors) to microcontrollers.

Ease of development

I²C can be used to communicate with a large number and range of peripherals. By using the same protocol to talk to a range of devices, development efforts may be reduced.

Scalability

The maximum size of an I²C bus is limited by the maximum capacitance of 400 pF. To estimate the maximum network size, we can add up the input capacitance of all the devices we wish to connect to the bus.

For example, the SGS-Thomson ST24C16 16K I²C-compatible EEPROM has an input capacitance of around 8 pF, and we could attach some 50 of these devices together on the same bus. Note that this rough calculation ignores any effects of stray capacitance due to the cabling itself. If we assume a 'standard' capacitance per device of 20 pF (including stray capacitance), we can still hope to connect around 20 peripherals to the microcontroller over a single bus.

Overall, the bus supports large enough collections of peripherals to meet the needs of its main application area.

Flexibility

I²C is flexible. Networks of one Master and multiple Slaves and networks with multiple Masters can both be implemented. Note: we consider only single-Master applications in this pattern.

Speed of execution and size of code

As we have mentioned, I²C can operate over a wide speed range: from 100 kbits/s in the 'standard' version (1992), to 400 kbits/s in the 'fast' version, and now 3.4 Mbit/s in the latest 'high-speed' version introduced in 1998.

However, in this pattern we are concerned with I²C interfaces created entirely in software. Such interfaces are only practical at the low end of the speed range.

Generation of the I²C protocol inevitably adds to the code size: see the core I²C library (in the following example) for details.

Cost

The cost of licence fees for use of the bus is included in the cost of the peripheral components which you purchase: in most circumstances, there are no additional fees to pay.

Note that this may not be the case if, for example, you are implementing a I^2C peripheral (to be sold for connection to an I^2C bus). If in doubt, please contact Philips for further details.

Choice of implementations and vendors

The I^2C library presented here may be used with any 8051 device.

Suitability for use in time-triggered applications

As we saw in Chapter 18, the RS-232 communication protocol is appropriate for use in time-triggered applications. This suitability arises because the task duration associated with transmission (and reception) of data on an RS-232 network is very short. Note that this transmission time is not directly linked to the baud rate of the network, largely because almost all of the 8051 family have on-chip hardware support for RS-232, with the result that messages are transmitted and received 'in the background'.

The situation with I^2C is rather different. Specifically, in this pattern, we are concerned with software-based I^2C protocols; this undoubtedly adds to the software load. For example, if we consider the process of sending one byte of data to an I^2C-based ROM chip (an example of this is presented in full later), then the total task duration is approximately 0.5 ms.

This task duration can be supported on a time-triggered application, even with 1 ms timer ticks, if the maximum data rate (1000 bytes / second) matches the needs of the application.

How do you use I^2C in a time-triggered application?

The examples that follow include a complete I^2C library which may be used with any member of the 8051 family.

Hardware resource implications

I^2C requires the use of two port pins. This is considerably fewer than would be required to create a parallel interface to most peripheral devices.

There is, however, a significant CPU load: see 'Solution' for details.

Reliability and safety implications

The I^2C protocol incorporates only minimal error-checking mechanisms: detection of data corruption (for example) during the transfer of information to or from a periph-

eral device must be carried out in software, if required. However, in most cases, damage to cabling will be quickly detected.

Portability

I²C is a simple protocol and may be easily generated in software. This allows the techniques presented here to be used with *all* 8051 devices.

Overall strengths and weaknesses

- ☺ **I²C is supported by a wide range of peripheral devices.**
- ☺ **I²C requires only two port pins to connect to (typically) up to 20 peripherals.**
- ☺ **I²C is a simple protocol and may be easily generated in software. This allows *all* 8051 devices to communicate with a wide range of peripheral devices.**
- ☺ **A common set of software code may be used with all I²C peripherals.**
- ☹ Although I²C is fast enough (even when generated in software) to be compatible with time-triggered architectures, typical data transfer rates will be comparatively slow (up to 1000 bytes / second with a 1 ms scheduler tick).

Related patterns and alternative solutions

See **SPI PERIPHERAL** [page 521].

Remember also that, if possible, the best approach is to avoid peripherals altogether and use a microcontroller with on-chip facilities if possible. For example, rather than using a basic 8051 plus serial EEPROM, consider using the Atmel AT89LS8252, which has two kbytes of EEPROM on the chip.

chapter 24

Using 'SPI' peripherals

Introduction

When using asynchronous serial protocols, such as 'RS-232', the two devices that are communicating must agree on a communication frequency (the baud rate) and each device then uses (say) an independent crystal-based clock to ensure that it operates at the required rate. One consequence of this approach is that, if the clocks in the transmitter and receiver devices vary by more than a few per cent, the receiving device will be unable to decode the incoming data correctly.

By contrast, the serial peripheral interface (SPI) uses a *synchronous* communication protocol: this means that both the transmitter and receiver devices share a common clock. The transitions of this common clock determine when to send and receive the various bits. For example, in a simple synchronous protocol, the transmitter device may write a bit on the rising edge of the clock and the receiving device will then read this bit when it detects the falling clock edge. Note that the clock frequency does not need to be held constant.

Such synchronous interfaces are primarily intended for use over distances measured in centimetres rather than in metres and we will restrict our discussions here to the use of SPI to link the 8051 with compatible peripherals.

SPI PERIPHERAL

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, based on a scheduler.
- The microcontroller in your application will be interfaced to one or more peripherals, such as a keypad, EEPROM, digital-to-analogue converter or similar device.
- Your microcontroller has hardware support for the SPI protocol.

Problem

Should you use the SPI bus to link your microcontroller to peripheral devices and, if so, how do you do so?

Background

There are five key features of SPI as far as the developer of embedded applications is concerned:

- SPI is a protocol designed to allow microcontrollers to be linked to a wide range of different peripherals – memory, displays, ADCs and similar devices – and requires (typically) three port pins for the bus, plus one chip-select pin per peripheral.
- There are many SPI-compatible peripherals available for purchase 'off the shelf'.
- Increasing numbers of 'Standard' and 'Extended' 8051 devices have hardware support for SPI and we will make use of such facilities in this pattern.
- A common set of software code may be used with all SPI peripherals.
- SPI is compatible with time-triggered architectures and, as implemented in this book, is faster than I^2C (largely due to the use of on-chip hardware support). Typical data transfer rates will be up to 5,000–10,000 bytes / second (with a 1 ms scheduler tick).

We provide some background to SPI in this section.

History

Serial peripheral interface (SPI) was developed by Motorola and included on the 68HC11 and other microcontrollers. Recently, this interface standard has been adopted by manufacturers of other microcontrollers. Increasing numbers of 'Standard' and 'Extended' 8051 devices (see Chapter 3) have hardware support for SPI and we will make use of such facilities in this pattern.

Basic SPI operation

SPI is often referred to as a three-wire interface. In fact, almost all implementations require two data lines, a clock line, a chip select line (usually one per peripheral device) and a common ground: this is at least four lines, plus ground.

The data lines are referred to as MOSI ('Master out Slave in') and MISO ('Master in Slave out').

The overall operation of SPI is easy to understand if you remember that the protocol is based on the use of two 8-bit shift registers, one in the Master, one in the Slave (Figure 24.1).

The key operation in SPI involves transferring a byte of data between the Master and the currently selected Slave device; simultaneously, a byte of data will be transferred back from the Slave to the Master.

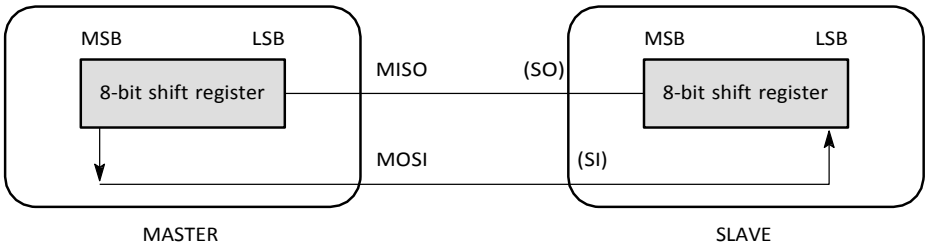


FIGURE 24.1 The two 8-bit shift registers at the heart of the SPI protocol

[*Note*: that some Slave devices (such as EEPROMs) label the data lines SI ('Slave in') and SO ('Slave out'). The SI line on the Slave is connected to the MOSI line on the Master and the SO line on the Slave is connected to the MISO line on the Master.]

Single-Master, multi-Slave

SPI is a single-Master, multi-Slave interface. The Master generates the clock signal. As far as we are concerned here, the microcontroller will form the Master device and one or more peripheral devices will act as Slaves.

Choice of clock polarities

SPI supports two clock polarities. With polarity 0, the clock line is low in the quiescent state: when active, the data to be sent are written on the rising clock edge and the data are read on the falling clock edge. With polarity 1, the clock line is high in the quiescent state: when active, the data to be sent are written on the falling clock edge and the data are read on the rising clock edge.

Polarity 0 is more widely used.

Maximum clock rate

The maximum clock rate for SPI is currently 2.1 MHz. Allowing for the fact that it takes eight clock cycles to transfer a byte of data and the fact that there are other

overheads too (instructions and addresses, for example), the maximum **data** transfer rates will be around 130,000 bytes per second.

Microwire

Note that the Microwire interface standard (developed by National Semiconductor) is similar to SPI, although the connection names, polarities and other details vary: Microwire is not discussed further in this book.

Solution

Should you use SPI?

In order to determine whether use of an SPI bus is appropriate in your time-triggered application, we consider some key questions that should be asked when considering the use of any communications protocol or related technique.

Main application areas

Although the SPI bus may be used, for example, to connect processors (usually microcontrollers) to one another or to other computer systems, its main application area is – like I^2C – in the connection of standard peripheral devices, such as LCD panels or EEPROMs to microcontrollers.

Ease of development

SPI can be used to communicate with a large number and range of peripherals. By using the same protocol to talk to a range of devices, development efforts may be reduced.

Scalability

Each SPI Slave device requires a separate /CS (chip select) line from the Master node. This increases the number of pins required on the microcontroller if large numbers of peripherals are used.

Flexibility

Individual SPI-compatible microcontrollers may act as Master or Slave nodes. We consider only the use of the microcontroller as the Master node in this pattern.

Speed of execution and size of code

The maximum clock rate for SPI is currently 2.1 MHz.

As we will be using hardware-based SPI in this pattern, the code overhead will be small.

Cost

The cost of licence fees for use of the bus is included in the cost of the peripheral components which you purchase: in most circumstances, there are no additional fees to pay.

Note that this may not be the case if, for example, you are implementing an SPI peripheral (to be sold for connection to an SPI bus). If in doubt, contact Motorola for further details.

Choice of implementations and vendors

The SPI library presented here may be used only with 8051 devices that have hardware support for SPI.

Suitability for use in time-triggered applications

As we saw in Chapter 18, the RS-232 communication protocol is appropriate for use in time-triggered applications. This suitability arises because the task duration associated with transmission (and reception) of data on an RS-232 network is very short. Note that this transmission time is not directly linked to the baud rate of the network, largely because almost all members of the 8051 family have on-chip hardware support for RS-232, with the result that messages are transmitted and received 'in the background'.

The situation with SPI is similar. Specifically, in this pattern, we are concerned with hardware-based SPI protocols. These typically impose a low software load and allow a short task duration. For example, if we consider the process of sending one byte of data to an SPI-based ROM chip (an example of this is presented in full later), then the total task duration is approximately 0.1 ms; note that this is considerably shorter than the equivalent operation using the I^2C library presented in this book.

This task duration can be easily supported in a time-triggered application, even with 1 ms timer ticks.

How do you use SPI in a time-triggered application?

The discussions will centre around the Atmel AT89S53, a Standard 8051 device with on-chip SPI support. Note that hardware support provided by other manufacturers is very similar.

The AT89S53 SPI features include the following:

- Full-duplex, three-wire synchronous data transfer
- Master or Slave operation
- 1.5 MHz bit frequency (max.)
- LSB first or MSB first data transfer
- Four programmable bit rates
- End of transmission interrupt flag

- Write collision flag protection
- Wakeup from idle mode (Slave mode only)

The interconnection between Master and Slave CPUs with SPI is as shown in Figure 24.1.

The SCK pin is the clock output in the Master mode (the only mode we will use here). Writing to the SPI data register of the Master CPU starts the SPI clock generator and the data written shifts out of the MOSI pin and into the MOSI pin of the Slave CPU. After shifting one byte, the SPI clock generator stops, setting the end of transmission flag (SPIF). If both the SPI interrupt enable bit (SPIE) and the serial port interrupt enable bit (ES) are set, an interrupt is requested. We will not use these interrupt facilities.

The Slave Select input, SS/P1.4, is set low to select an individual SPI device as a Slave. When SS/P1.4 is set high, the SPI port is deactivated and the MOSI/P1.5 pin can be used as an input.

There are four combinations of SCK phase and polarity with respect to serial data, which are determined by control bits CPHA and CPOL (see Table 24.1).

Most of the features of the SPI interface in the AT89S53 are illustrated in the example that follows.

Hardware resource implications

With on-chip hardware support, **SPI PERIPHERAL** imposes a minimal software load.

Reliability and safety issues

The SPI protocol incorporates only minimal error-checking mechanisms: detection of data corruption (for example) during the transfer of information to or from a peripheral device must be carried out in software, if required.

Portability

This pattern requires hardware support for SPI: it cannot be used with microcontrollers without such support.

The discussions here are based on the Atmel 89S53. Use with other 8051 microcontrollers – including many Infineon 8051s – is straightforward.

Overall strengths and weaknesses

- ☺ **SPI is supported by a wide range of peripheral devices.**
- ☺ **SPI requires (typically) three port pins for the bus, plus one chip-select pin per peripheral.**
- ☺ **Use of hardware-based SPI (as discussed here) facilitates the design of tasks with short durations; as a consequence the protocol is well matched to the**

TABLE 24.1 The SPI control register (SPCR) in the Atmel 89S53

Bit	Name	Overview
7	SPIE	SPI interrupt enable. **Not used in this text** **Usually set SPIE = 0**
6	SPE	SPE = 1 enables the SPI channel. **Note that this means that the upper nybble of Port 1 is not available for general-purpose I/O** Usually set SPE = 1
5	DORD	The data order. DORD = 0 selects 'most significant first' data transmission; DORD = 1 selects 'least significant first' data transmission **Usually set DORD = 0**
4	MSTR	Master / Slave select. MSTR = 1 selects Master SPI mode **We will only use Master mode in this text: set MSTR = 1**
3	CPOL	The clock polarity. When CPOL = 0, the SCK of the Master device is low when no data are being transmitted; when CPOL = 1, the SCK is high under these circumstances **Usually set CPOL = 0**
2	CPHA	The clock phase. Refer to the Atmel documents for details **Usually set CPHA = 0**
1	SPR1	The clock-rate select. These two bits control the SCK rate of the device, when it is configured as a Master
0	SPR0	The relationship between SCK and the oscillator / resonator frequency ('Osc') is as follows: SPR1 SPR0 SCK 0 0 Osc / 4 0 1 Osc / 16 1 0 Osc / 64 1 1 Osc / 128 Note that the AT89S53 has a maximum (SPI) bit rate of 1.5 MHz

needs of time-triggered applications. Typical data transfer rates will be up to 5,000–10,000 bytes / second (with a 1 ms scheduler tick).

☺ **A common set of software code may be used with all SPI peripherals.**

☹ The use of this pattern is restricted to microcontrollers with hardware support for SPI.

Related patterns and alternative solutions

The use of this pattern is restricted to microcontrollers with hardware support for SPI: see **I²C PERIPHERAL** [page 494] for an alternative solution that provides very similar facilities without the need for hardware support.

Monitoring and control components

The penultimate group of patterns presented in this book are intended for use in data acquisition, condition monitoring and control systems. As in previous parts, we are concerned with techniques which are appropriate for use in systems employing a time-triggered software architecture.

We begin in Chapter 30 by considering the counting of pulses. These techniques are in widespread use in many industrial applications and throughout the automotive industry. **HARDWARE PULSE COUNT** [page 728] and **SOFTWARE PULSE COUNT** [page 736] present, respectively, hardware-based and software-only techniques for pulse counting.

In Chapter 31, we address pulse-rate modulation: that is, the generation of square-wave signals of a specified frequency. Again, two patterns are presented: **HARDWARE PRM** [page 742] and **SOFTWARE PRM** [page 748].

In Chapter 32 we consider how we can use an 8051 microcontroller to measure analogue voltage or current signals. These may, for example, represent the battery voltage in a charger application: more generally, such signals can come from an enormous range of sensors, such as potentiometers or temperature probes. The patterns **ONE-SHOT ADC** [page 757] and **SEQUENTIAL ADC** [page 782s] discuss how to make effective use of on-chip and off-chip analogue-to-digital converters (ADCs). Also in this chapter, the patterns **ADC PRE-AMP** [page 777] and **A-A FILTER** [page 794] consider important pre-processing stages that may be required before an analogue signal is measured. Finally, the pattern **CURRENT SENSOR** [page 802] considers how we can use analogue measurements for current-sensing applications, such as detecting blown bulbs or stalled DC motors.

In Chapter 33 we explore how analogue outputs from the microcontroller – generated in the form of pulse-width modulated signals – can be used in applications such as setting the speed of motors or controlling the brightness of bulbs. Again, both hardware-

and software-based techniques are considered, in the patterns **HARDWARE PWM** [page 808] and **SOFTWARE PWM** [page 831]. We also consider the post-processing that may be required to filter PWM-based signals, through the pattern **PWM SMOOTHER** [page 818] and the creation of a high-frequency PWM output without using specialized hardware in the pattern **3 - LEVEL PWM** [page 822].

In Chapter 34 we consider how analogue outputs from the microcontroller may be generated using digital-to-analogue converter (DAC) hardware (see **DAC OUTPUT** [page 841]). We consider the post-processing that may be required to filter and / or amplify DAC-derived signals, through the patterns **DAC SMOOTHER** [page 853] and **DAC DRIVER** [page 857].

Finally, in Chapter 35, we turn our attention to proportional-integral-differential (PID) control. PID is both simple and effective: as a consequence it is the most widely used control algorithm. The focus in this chapter is on techniques for designing and implementing PID controllers for use in embedded, time-triggered applications.

chapter 30

Pulse-rate sensing

Introduction

Suppose we wish to measure the speed of a rotating shaft. This could be part, for example, of an automotive or industrial application.

An effective way of measuring the speed is to attach an optically or magnetically based rotary encoder to the shaft (Figure 30.1), and to count the number of pulses that occur over a fixed period of time (say 100 ms or 1 second). From the count, and having details of the rotary encoder, we can calculate the average speed of rotation.

In this chapter we consider how pulses may be counted. The technique we discuss may be used not just for the measurement of rotational speed but also, for example, in liquid flow-rate measurement and in the sensing of vibration.

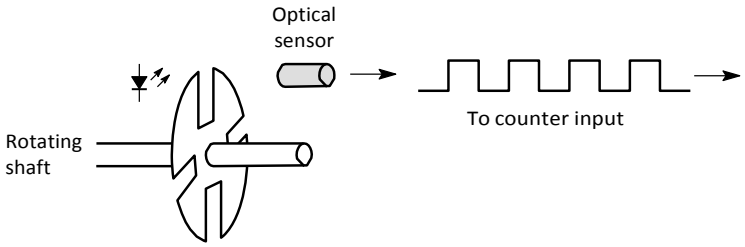


FIGURE 30.1 Counting pulses from an optical encoder in order to measure the speed of rotation of a shaft

HARDWARE PULSE COUNT

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you count rectangular pulses arriving from an external peripheral?

Background

–

Solution

Consider the train of pulses obtained from the rotary encoder discussed in the introduction to this chapter.

Where we have sufficient hardware resources available, we can often determine the average pulse rate from this stream largely without software intervention. Timer 0 or Timer 1[1] will count pulses (more specifically, the falling edges of pulses) on external pins, without generating interrupts and without interfering with any other processing.

The key to **HARDWARE PULSE COUNT** is the setting of the TMOD SFR (Table 30.1).

TABLE 30.1 The TMOD SFR, used to control Timer 0 and Timer 1

Bit	7 (MSB)	6	5	4	3	2	1	0 (LSB)
Name	Gate	$C/\overline{T}$	M1	M0	Gate	$C/\overline{T}$	M1	M0
	Timer 1				*Timer 0*			

In Table 30.1, most of the features of TMOD have been discussed previously (see Chapter 11). Here we are particularly concerned with the 'counter' bits (6 and 2). If either of these bits is set to 0, then the corresponding timer is set to counter mode.

1. In many cases, as we saw in Chapter 3, modern '8051' family devices are based on the slightly later 8052 architecture: such devices include an extra, more powerful timer (Timer 2). Timer 2 can also be used to count pulses; however, as we discussed in Chapters 13 and 14, we prefer to use this timer – where available – as a source of scheduler ticks. We will therefore not discuss the use of Timer 2 for pulse counting here.

For example:

```
// Timer 0 used as 16-bit timer, counting pulses
// (falling edges) on Pin 3.4 (T0 pin)
TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
TMOD |= 0x05; // Set required T0 bits (T1 left unchanged)
```

In this case, the Timer 0 registers (TL0 and TH0) are incremented in response to a transition (1-to-0) at its corresponding external input pin (Figure 30.2). The pin is sampled every machine cycle. When the samples show a high in one cycle and a low in the next cycle, the count is incremented. The new count value then appears in the register in the cycle following the one in which the transition was detected.

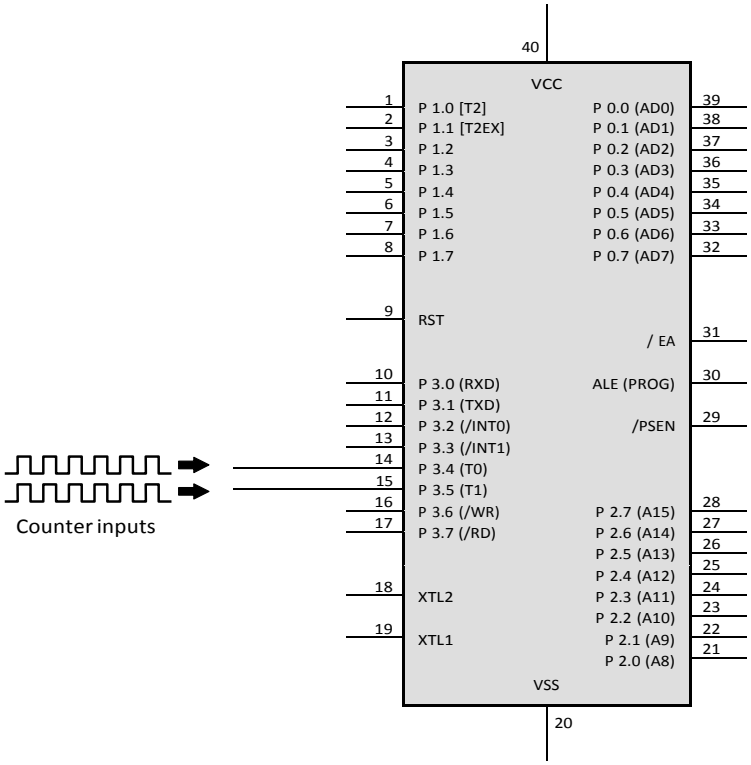


FIGURE 30.2 Reading pulse counts

There are no restrictions on the duty cycle of the waveform that we can monitor using this approach, but – to ensure accurate measurement – the 'high' and 'low' levels should be held for at least one full machine cycle (12 oscillator cycles in an original 8051; 6, 4 or 1 oscillator cycle(s) in some more recent variants). As a result, it takes at ▶

least two machine cycles (24, 12, 8 or 2 oscillator cycles) to detect a 1-to-0 transition. This determines the maximum pulse rate that you can measure: for example, on a basic 12 Mhz / 12 oscillator cycle 8051, the maximum pulse rate (square wave) that we can measure has a period of 24 oscillator cycles (2 µs), and a frequency of 500 kHz.

We can use this information efficiently to measure the rotational speed in the previous example by creating a task that operates as follows:

1 Read the current hardware pulse count.
2 Store the result in a global variable.
3 Reset the count to 0.

We need to schedule this task for repeated execution (say every 100 ms).
We give a code example that illustrates this approach in a following example.

Hardware resource implications

HARDWARE PULSE COUNT requires exclusive access to a timer. In many cases, Timer 2 will be used for the scheduler and Timer 1 may be used for baud-rate generation. This will only leave Timer 0 for this purpose. Where this is not available (or you need to measure multiple pulse trains) you will need to consider software-based techniques: see **SOFTWARE PULSE COUNT** [page 736] or the use of an additional microcontroller (see Part F).

Reliability and safety implications

There are no specific reliability and safety implications, provided you do not attempt to count pulses beyond the limits of your hardware (see 'Solution' for details).

Portability

This pattern uses only core 8051 features: there are no specific portability implications.

Overall strengths and weaknesses

☺ **HARDWARE PULSE COUNT imposes a minimal software overhead.**

☹ **HARDWARE PULSE COUNT** requires exclusive access to a timer for each pulse train you wish to count.

Related patterns and alternative solutions

There are two main alternatives to **HARDWARE PULSE COUNT**.

The closest match is **SOFTWARE PULSE COUNT** [page 736]: this performs in a similar way, but does not require the use of timer hardware. Inevitably, this imposes a larger software load than the present pattern.

Alternatively, you can use external hardware to convert the pulse train to an analog voltage, which may then be measured using an on-chip or external analogue-to-digital converter (ADC). Various 'frequency-to-voltage' conversion chips are available to assist with this process, such as the National Semiconductor[2] LM2907 and LM2917. The use of an ADC is discussed in **ONE-SHOT ADC** [page 757].

Note that the use of an ADC (and associated hardware) will not allow you to count, precisely, the number of pulses that occur: this approach would probably not be appropriate, for example, for counting visitors passing through a turnstile. However, if you require a measure of the average pulse count – as in some forms of speed measurement – then this solution may be adequate. Note, however, that there may be significant hardware costs involved.

SOFTWARE PULSE COUNT

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you count pulses arriving from an external peripheral?

Background

See **HARDWARE PULSE COUNT** [page 582] for background details.

Solution

HARDWARE PULSE COUNT requires exclusive access to a timer.

Where this timer is not available and comparatively low pulse rates are to be measured, software-only pulse counting is possible.

Software-based pulse counting involves a periodic task that keeps track of the state of one or more input pins. When a pin changes state in the interval between task invocations, we increment the corresponding count by 1. The main drawback is the maximum pulse rate: with a 1 ms scheduler, the maximum pulse rate we can measure is 500 Hz: this is around 1,000 times less than can be measured using **HARDWARE PULSE COUNT** [page 730].

chapter 32

Using analogue-to-digital converters (ADCs)

Introduction

The recording of analog signals is an important part of many condition monitoring, data acquisition and control applications.

We will consider how to read analog values using an 8051 microcontroller in this chapter. In doing so, we will present the following patterns:

- **ONE-SHOT ADC** [page 757] addresses the problem of measuring an analogue voltage signal at irregular (or infrequent) intervals, using a microcontroller
- **ADC PRE-AMP** [page 777] addresses the problem of amplifying an analogue signal, in order to convert it to a range suitable for subsequent analogue-to-digital conversion.
- **SEQUENTIAL ADC** [page 782] addresses the problem of recording a sequence of analogue samples using a microcontroller.
- **A-A FILTER** [page 794] addresses the problem of filtering an analogue signal to remove high-frequency components.
- **CURRENT SENSOR** [page 802] addresses the problem of monitoring the current flowing through a DC load.

ONE-SHOT ADC

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you measure an analogue voltage or current signal at irregular (or infrequent) intervals?

Background

We consider some basic background material in this section.

Measuring voltages

Consider the analogue input from the potentiometer shown in Figure 32.1.

The resulting analogue voltage (in the range, here, of 0–5V) can be used as part of a user interface, if we have an on- or off-chip analogue-to-digital converter (ADC) available; as we will see, such devices are common.[4]

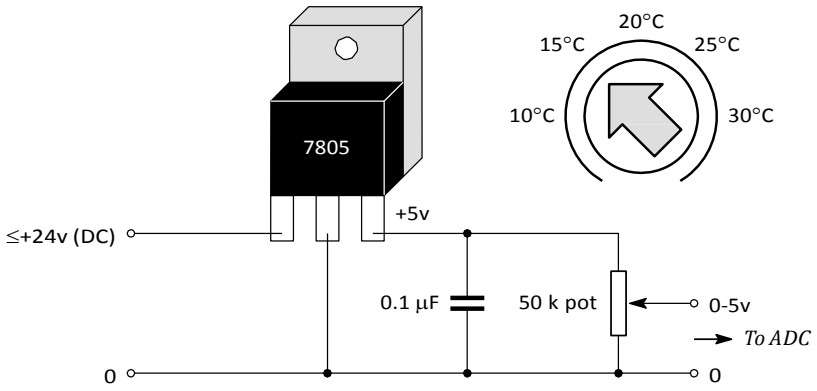


FIGURE 32.1 Using a potentiometer as part of a user interface

4. Please note that, although in widespread use, there may be better ways of creating such a user interface: refer to 'Related patterns and alternative solutions' (page 762).

This general approach is more widely applied. For example, consider Figure 32.2. Here we use three potentiometers (and an appropriate three-channel ADC) to measure angles in a mechanical excavator. By measuring the angles at points A, B and C, we can determine the position (specifically, the depth) of the shovel (X).

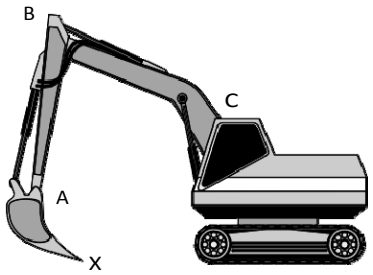


FIGURE 32.2 Using potentiometers to measure angles in a mechanical excavator

[*Note*: By measuring the angles at points A, B and C, we can determine the position (specifically, the depth) of the shovel (X).]

Measuring currents

In these simple earlier examples, we illustrated how analogue voltage signals might be a useful source of information in embedded applications. However, particularly in industrial applications, it can be helpful to be able to measure analogue *current* signals.

To see why current signals can be useful, consider Figure 32.3.

Figure 32.3 shows a sensor which, we assume, is connected to a long stretch of wire (with resistance R_{wire}) and, thereby, to the analogue (voltage) input of a microcontroller. The sensor, we will assume, measures temperature and generates an output of 5V to represent 100°C and 1 V to represent 0°C. We will further assume that only positive temperatures are possible and that, if the sensor or the wiring is broken, the voltage measured will be 0V, indicating an error.

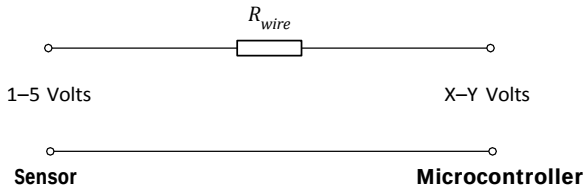


FIGURE 32.3 The problems with voltage-output sensors

There are two problems with this arrangement:

1 The voltage at the microcontroller will be less – possibly much less – than the voltage at the sensor. We will therefore probably need to fine-tune the sensor code to take into account the voltage drop in the wiring.
2 The wire resistance (and, hence, the voltage drop) will be temperature dependent, making it very difficult to obtain accurate readings even after fine-tuning.

One good solution to this problem is to digitize the analogue readings in the sensor and to transmit a digital representation of the voltage signal to the main microcontroller; the techniques discussed in Part F can be used to implement this solution.

Another solution, which is very common in the process industry, is to use a varying current (e.g. 4–20 mA) rather than a varying voltage to encode the sensor information (Figure 32.4).

The current-source sensor works well, even over long distances, even where the wiring resistance varies with temperature, since the sensor can adjust its output voltage, as required, to keep the current at the specified level. Note also that, at the microcontroller, we may simply convert the current signal back to a voltage signal by using a fixed resistor.

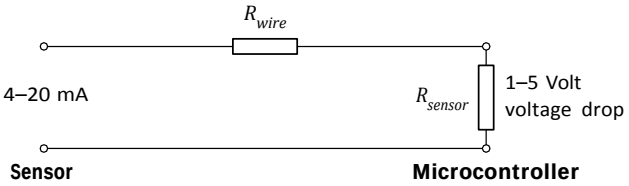


FIGURE 32.4 A schematic representation of a current-mode sensor

Solution

We will briefly consider here some of the hardware options that are available to allow the measurement of analogue voltage or current signals using a microcontroller.

Specifically, we will consider four options:

- Using a microcontroller with on-chip (voltage-mode) ADC
- Using an external serial (voltage-mode) ADC
- Using an external parallel (voltage-mode) ADC
- Using a current-mode ADC

We present a number of complete software libraries in the examples that follow.

Using a microcontroller with on-chip (voltage-mode) ADC

Many members of the 8051 family contain on-board ADCs. In general, use of an internal device will result in increased reliability, since both hardware and software

complexity will often be lower. In addition, the 'internal' solution will usually be physically smaller and have a lower system cost.

Here are some examples of the available ADC components provided on two 8051 devices.

Infineon c515c

From the Infineon[5] c515c data sheet:

The c515c includes a high performance / high speed 10-bit A/D-converter (ADC) with 8 channels. It operates with a successive approximation technique and uses self calibration mechanisms for reduction and compensation of offset and linearity errors.

The A/D converter provides the following features:

- 8 multiplexed input channels (port 6), which can also be used as digital inputs
- 10-bit resolution
- Single or continuous conversion mode
- Internal or external start-of-conversion trigger capability
- Uses successive approximation conversion technique via a capacitor array
- Built-in hidden calibration of offset and linearity errors

Analog Devices ADμC812

From the Analog Devices[6] ADμC812 data sheet:

The ADC conversion block incorporates a fast, multi-channel, 12-bit, single supply A/D converter. This block provides the user with multi-channel mux, track/hold, on-chip reference, calibration features and A/D converter. All components in this block are easily configured via the SFR interface from the core MCU.

The A/D converter section in this block consists of a conventional successive-approximation converter based around a capacitor DAC. The converter accepts an analogue input range of 0 to +VREF. A high precision, low drift 2.5V reference is provided on-chip. The internal reference may be overdriven via the external VREF pin. This external reference can be in the range 2.3V to AVDD.

Single step or continuous conversion modes can be initiated in software or alternatively by applying a convert signal to the an external pin. Timer 2 can also be configured to generate a repetitive trigger for ADC conversions. The ADC may also be configured to operate in a DMA Mode whereby the ADC block continuously converts and captures samples without any interaction from the MCU core.

The ADC core contains self-calibration and system calibration options to ensure accurate operation over time and temperature. A voltage output from an On-Chip bandgap reference proportional to absolute temperature can also be routed through the front-end ADC multiplexor facilitating a temperature sensor implementation.

---

5. www.infineon.com
6. www.analog.com

Using an external parallel (voltage-mode) ADC

The 'traditional' alternative to an on-chip ADC is a parallel ADC. In general, parallel ADCs have the following strengths and weaknesses:

- ☺ **They can provide fast data transfers.**
- ☺ **They tend to be inexpensive.**
- ☺ **They require a very simple software framework.**
- ☹ They tend to require a large number of port pins. In the case of a 16-bit conversion, the external ADC will require 16 pins for the data transfer, plus between one and three pins to control the data transfers.
- ☹ The wiring complexity can be a source of reliability problems in some environments.

Examples of the use of a parallel ADC follow.

Using an external serial (voltage-mode) ADC

Many more recent ADCs have a serial interface. In general, serial ADCs have the following strengths and weaknesses:

- ☺ **They require a small number of port pins (between two and four), regardless of the ADC resolution.**
- ☹ They require on-chip support for the serial protocol or the use of a suitable software library.
- ☹ The data transfer may be slower than a parallel alternative.
- ☹ They can be comparatively expensive.

Two examples of the use of serial ADCs follow.

Using a current-mode ADC

As we discussed in 'Background', use of current-based data transmission can be useful in some circumstances.

A number of current-mode sensor components (e.g. the Burr-Brown[7] XTR105) and ADCs (e.g. the Burr-Brown RCV420) are now available.

In addition, **CURRENT SENSOR** [page 802] discusses current sensing using voltage-mode ADCs.

Hardware resource implications

Use of the internal ADC will generally mean that at least one input pin is unavailable for other purposes; use of an external ADC will require the use of larger numbers of port pins.

---

7. www.burr-brown.com

Use of the on-chip ADC may also have an impact on the power consumption of your microcontroller.

Reliability and safety implications

Use of ADCs has no particular safety implications. However, all things being equal, an application using an internal ADC is likely to prove more reliable than the same application created using an external ADC, largely because the hardware complexity of the 'internal' solution will be much less than that of the 'external' solution.

Portability

All ADCs vary in their properties: for example, code for one serial ADC will often not work without alteration on another serial device. However, the required changes are generally minor.

Overall strengths and weaknesses

☺ **ADC inputs are essential in many applications.**

☹ Microcontrollers with ADCs are more expensive than those without such facilities.

Related patterns and alternative solutions

Many microcontrollers now have multiple A/D converter channels available, making it possible to implement low-cost user inputs, for example for setting temperatures, operating points and so on.

The main advantage of this type of analogue input is that it directly provides user feedback: for example, we simply add a scale around the potentiometer control to indicate the required temperature. The cost of such an input sensor is frequently much lower than the corresponding digital (two switch plus display) equivalent (Figure 32.5).

However, in almost all circumstances the digital solution will provide more precise control and will not alter with age (as the performance of the potentiometer is likely to do). In addition, if we wish to alter the temperature of the system under software control we can easily, in the case of the digital solution, update the temperature display, while we cannot generally rotate the potentiometer under software control to match the new temperature.

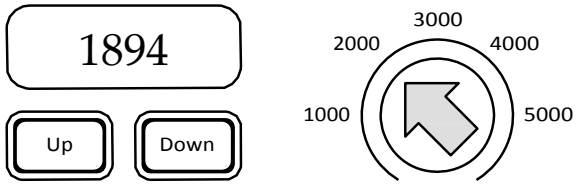


FIGURE 32.5 Two possible user interfaces, one based on an LCD/LED display with switches, the other based on an analog potentiometer

Nonetheless, the analog solution is often adequate, where high precision is not a requirement. For example, for a room thermostat control, a potentiometer may well be appropriate.

CURRENT SENSOR

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you monitor the current flowing through a DC load?

Background

Despite the name, **CURRENT SENSOR** involves the measurement of analogue voltage: see **ONE-SHOT ADC** [page 757] and **SEQUENTIAL ADC** [page 782] for relevant background information.

Solution

We consider solutions to the problem of current sensing in this section.

Using current-sense resistors

The theoretical basis of traditional current-sensing techniques is very straightforward. Suppose, for example, we wish to monitor the current flowing through the load shown in Figure 32.33.

To measure this current, we can place a resistor in series with the load and measure the voltage drop across the resistor (Figure 32.34).

The current through the load can then simply be determined, from Ohm's law, as follows:

$$I_{load} = \frac{V_{load}}{R_{load}}$$

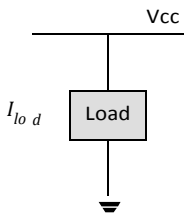


FIGURE 32.33 The problem: we wish to measure the current flowing through this load

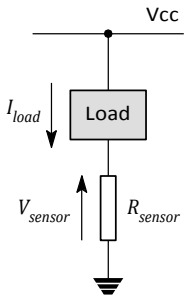


FIGURE 32.34 Using a resistor as a 'current to voltage converter' in a current-sense application

Any resistor may be used, but in most cases a component with a very small resistance is best: such devices are specially produced for current sensing. Typical values used will be less than 1 Ohm.

To understand why small resistors are required, suppose that we have a 12V load (connected to a 12V supply), with a current requirement of 1A: this is a typical requirement for many small bulbs or DC motors. If we add a 10 Ohm resistor in series with this load, the voltage drop across the resistor (again determined from Ohm's law, V = IR), will be 10V. If we use a resistor of 0.2 Ohms, the voltage drop will be reduced to 0.2V: this is acceptable in most situations.

Note that we need to have a sufficiently large voltage across the sensor resistor to enable us to monitor the voltage using an ADC connected to the microcontroller. To allow for reliable monitoring, you will generally need to have a voltage drop across the resistor of around 0.1V. Any less than this and you run the risk of supply fluctuations or EMI interfering with your measurements.

Note also that you must ensure that the sensor resistor is of an appropriate power rating. The required power rating, in Watts, can be determined as follows:

$$P_{resistor} = R_{sensor}(I_{load})^2$$

Thus, with a 0.5 Ohm resistor and a 3A load, the required power rating will be 4.5W.

Current-sensing resistors are available in ratings of 50W and above. However, the stated power ratings often assume the use of a heat sink connected to the resistor. If you do not use such a heat sink, then choose a resistor with at least double the calculated power rating.

A resistor-free alternative

In microcontroller-based systems using MOSFETs or BJTs for switching loads, we can often carry out current sensing without the use of a separate current-sense resistor.

For example, consider Figure 32.35.

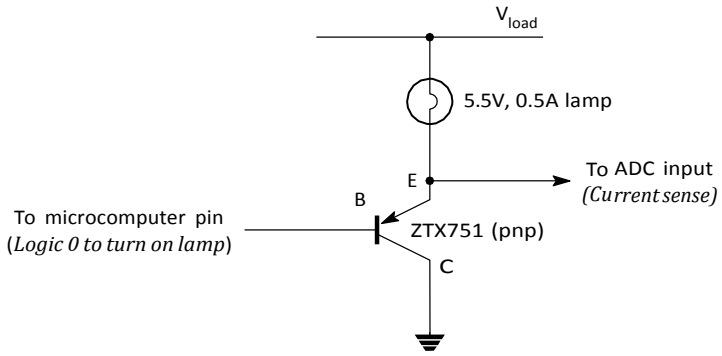


FIGURE 32.35 Current sensing without a resistor

With the BJT in saturation, the voltage drop across the emitter-collector terminals will be about 1V. We can therefore determine whether the bulb has blown by measuring the voltage at the emitter (relative to ground): if the voltage is ~1V, the bulb is lit. Note that we cannot reliably determine the level of the load current using this approach: we can only tell whether the load is being driven (or not).

MOSFETs provide a similar solution: see, for example, Figure 32.36.

In this circuit, if the MOSFET has an 'on' resistance of R_{on}, then the voltage at the MOSFET drain pin, when the load is driven, is given (again using Ohm's law) by:

$$V_{Drain} = I_{load}R_{on}$$

For the IRF540N (for example), R_{on} is 0.055Ω, so that – at a typical load current of around 2A – we have a voltage of ~0.1V at the drain pin, when the load is being driven. This may be readily measured even with an 8-bit ADC.

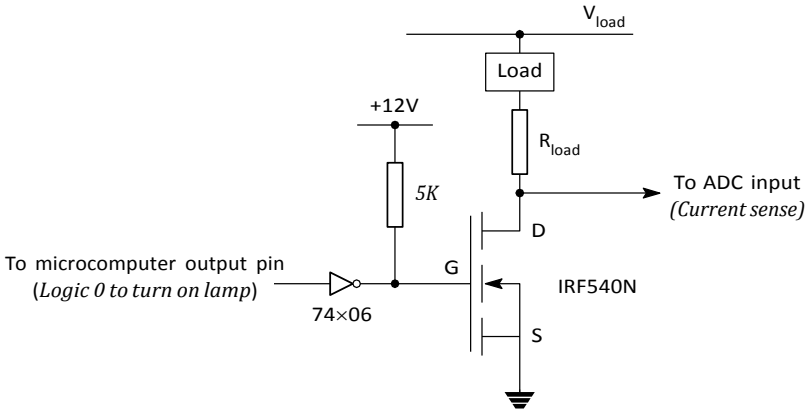


FIGURE 32.36 Current sensing without a resistor

Hardware resource implications

Use of this pattern generally requires the use of an on-chip or external ADC or comparator circuit.

Reliability and safety implications

Careful use of this technique can improve reliability and safety by allowing changes in the condition of a load (e.g., blown bulb, stalled DC motor) to be detected.

Portability

This hardware-only pattern is highly portable.

Overall strengths and weaknesses

☺ **Can improve reliability and safety.**

☹ Requires use of an ADC or similar hardware.

Related patterns and alternative solutions

See **ONE-SHOT ADC** [page 757] for alternative current-sense solutions.

Example: Detecting a blown bulb

As part of a security system, a 12V, 20W bulb (in fact, a car headlight bulb) is to be controlled by a microcontroller-based system. The basic arrangement is illustrated in Figure 32.37.

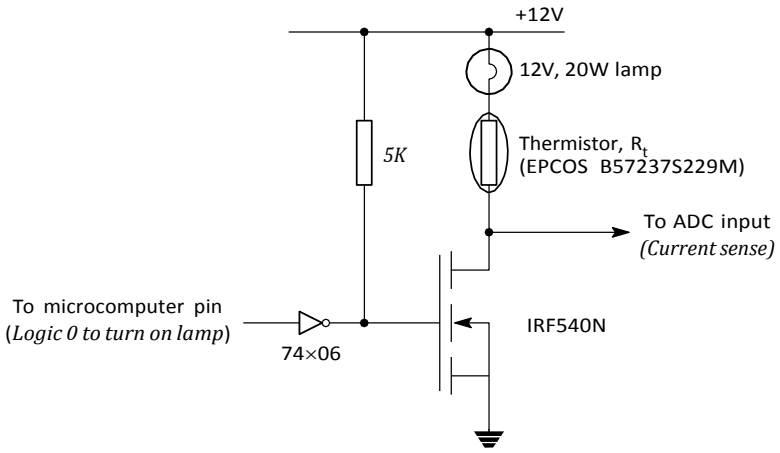


FIGURE 32.37 Detecting a blown bulb

With the bulb blown, the voltage is 0; with the bulb lit, a voltage of >= 0.1V can be measured.

Further reading

—

chapter 33

Pulse-width modulation

Introduction

Wait until it gets dark. Go to a room in your office or home where there is an ordinary (filament) light bulb. Switch the light on and off as rapidly as you can. The room will be dimly lit, with a flickering light. Pulse-width modulation (PWM) does exactly the same thing, at a much higher frequency. More specifically, PWM allows us to control (for example) the brightness of the light – without visible flickering – by switching the light on and off at a particular duty cycle.

PWM is an efficient basis for the control of high-power loads and is particularly widely used in applications such as DC (and AC) motor speed control.

There are two practical approaches to PWM signal generation in a time-triggered application:

- Several 8051-family microcontrollers provide hardware support for PWM on chip. This is generally easy to use. Where on-chip support is not available, specialist external hardware can provide cost-effective, high-frequency PWM switching.
- PWM outputs can be generated easily, at low frequencies, using software-only techniques.

These approaches are considered in the patterns **HARDWARE PWM** [page 808] and **SOFTWARE PWM** [page 831]. We also consider the post-processing that may be required to filter PWM-based signals, through the pattern **PWM SMOOTHER** [page 818] and creation of a high-frequency PWM output without using specialized hardware in the pattern **3-LEVEL PWM** [page 822].

HARDWARE PWM

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you create a high-frequency PWM output signal?

Background

A pulse-width modulated signal has many characteristics in common with the pulse-rate modulated signal discussed in **HARDWARE PRM** [page 742].

Like PRM, PWM is carried out by setting a port pin to Logic 1 for a period (X) and then to Logic 0 for another period (Y). We then repeat this process (Figure 33.1).

The (average) voltage at the port pin is determined by the duty cycle of the waveform. The duty cycle and some other common PWM features are defined as follows:

$$\text{Duty cycle (\%)} = \frac{x}{x + y} \times 100$$

Period = $x + y$, where x and y are in seconds.

$$\text{Frequency} = \frac{1}{x + y}\text{, where } x \text{ and } y \text{ are in seconds.}$$

The key point to note is that the average voltage seen by the load is given by the duty cycle multiplied by the load voltage.

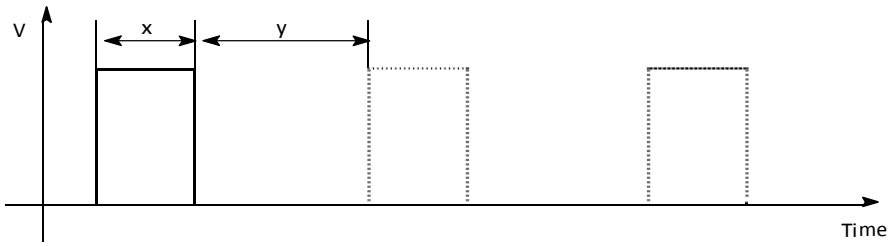


FIGURE 33.1 The underlying principles of PWM

Solution

We consider the following issues in this section:

- Creating PWM signals using on-chip hardware
- Creating PWM signals using external hardware
- Determining the required switching frequency
- Buffer and driver limitations
- Smoothing the PWM outputs

Creating PWM signals using on-chip hardware

We consider two specific examples of the generation of PWM signals using on-chip hardware in this section.

The Infineon c515c

In the Infineon c515c the PWM timer (which is based on Timer 2) is incremented with the machine clock, at one-sixth of the oscillator frequency. To generate a 16-bit PWM signal at the maximum (10 MHz) operational frequency, the timer is incremented every 300 ns (0.3 μs). The total period is 2^{16} x 0.3 μs, which is 19,660.8 μs or 19.7 ms. The frequency is therefore approximately 50 Hz.

The Dallas 87C550

The Dallas 87C550 incorporates a high-speed, 4-channel, PWM hardware interface. The maximum PWM frequency (8-bit PWM, 12 MHz oscillator) is approximately 46 kHz.

Creating PWM signals using external hardware

If your microcontroller does not have hardware PWM support, external PWM chips are available: see, for example, the low-cost Dallas DS1050 family,[16] which has a (5-bit) PWM output operating at up to 100 kHz.

Determining the required switching frequency: general guidelines

Determining, on paper, the PWM frequency you need to use for your application is generally not easy: it depends on a number of factors associated with the load and driver and it is generally necessary to perform some practical tests to determine the required frequencies: we discuss such tests later.

16. `www.dalsemi.com`

However, some basic guidelines can be given:

- The human eye can detect 'flicker' at rates of up to around 50 Hz. If your PWM hardware is controlling something like a bulb, the user may be able to see the flicker if the switching frequency is below this.
- The human auditory system has a range, at birth, of around 20 Hz to 20 kHz. If you switch high-power loads within this range, it is likely that the results will be audible to users, typically as a very annoying whine.

Determining the required switching frequency: practical studies

The only way of determining the required switching frequency in a practical application is to try a range of different frequencies.

One flexible way of doing this is to use a signal generator to drive the hardware and observe the results.

An effective alternative, that can often be carried out using your prototype system, is to apply the pattern **HARDWARE PRM** [page 742]. This allows the generation of square waves with a fixed (50%) duty cycle but variable period, at frequencies from less than 100 Hz to 3 MHz or more.

Buffer and driver limitations

The speed of the PWM hardware is, of course, not the only consideration in developing a PWM interface. For example, Figure 33.2 shows a circuit which attempts to perform PWM-based speed control of an AC motor.

This approach is doomed to failure, since the switch time of the EM relay will, typically, be of the order of 10 milliseconds; this restricts the PWM switching frequency to around 100 Hz. In many applications, this will not be adequate.

It is essential that you check the switching times of both switch hardware **and any associated buffer circuitry** when developing a PWM application.

Smoothing the PWM outputs

It is sometimes necessary to smooth PWM outputs, in order to remove high-frequency harmonics: see **PWM SMOOTHER** [page 818] for details.

Hardware resource implications

The main hardware resource implication is that, if you use the on-chip PWM, this is often based on Timer 2. As we have discussed, this timer is often used to drive the system scheduler (in a single-processor system).

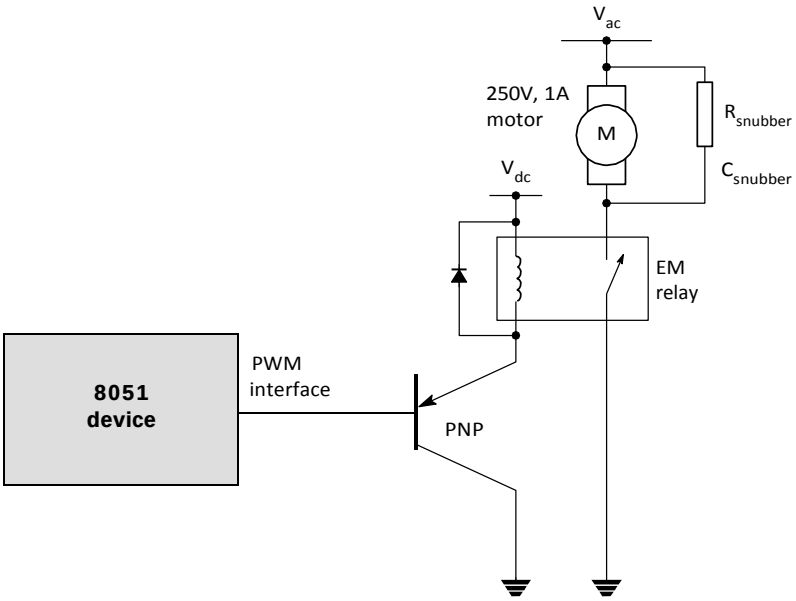


FIGURE 33.2 An attempt to control the speed of an AC motor using PWM control, using a driver based on an EM relay

[*Note*: This approach will fail, accept at very low switching frequencies.]

Reliability and safety implications

PWM routines are frequently used with high-power loads and, as usual with such loads, you need to ensure that your system starts in a safe state: for example, you need to ensure that, when the microcontroller is reset, the associated machinery begins at a slow speed, rather than a high speed. Refer to Chapters 7 and 8 for further discussion of this issue and some possible solutions.

More specifically, high-frequency PWM signals can be a source of EMI; this can be a particular problem for microcontroller-based embedded applications. Please see Ong *et al*. (2001) for further discussion of this issue.

Portability

A drawback with on-chip PWM hardware is that the features and implementation vary greatly over the 8051 family. In general, code written for external PWM hardware will be more easily ported across the 8051 family

Overall strengths and weaknesses

☺ **PWM has been used for many years as efficient means of providing a continuously varying output voltage to 'slow' external components, such as AC and**

DC motors and large heating elements. It remains a very effective way of controlling such applications, without the need for complex external hardware, like digital-to-analogue converters (see DAC OUTPUT [page 841]).

- ☹ PWM can be a source of EMI.
- ☹ Not all 8051 devices have on-chip support for PWM.
- ☹ Where on-chip PWM support is available, it is in no sense 'standard' and code written for one chip will not be particularly portable.

Related patterns and alternative solutions

See SOFTWARE PWM [page 831].
See DAC OUTPUT [page 841].

SOFTWARE PWM

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you create a low-frequency PWM output signal without using specialized PWM hardware?

Background

See **HARDWARE PWM** [page 808] for general background material on PWM.

See **SOFTWARE PRM** [page 748] for a similar solution.

Solution

The techniques discussed in **SOFTWARE PRM** [page 748] may be readily adapted to allow us to generate low-frequency pulse-width modulated signals. Listing 33.8 illustrates this.

```
void PWM_Soft_Update(void)
    {
    // Have we reached the end of the current PWM cycle?
   if (++PWM_position_G >= PWM_PERIOD)
      {
      // Reset the PWM position counter
      PWM_position_G = 0;

      // Update the PWM control value
      PWM_G = PWM_new_G;

      // Set the PWM output to OFF
      PWM_pin = PWM_OFF;

      return;
      }

   // We are in a PWM cycle
   if (PWM_position_G < PWM_G)
```

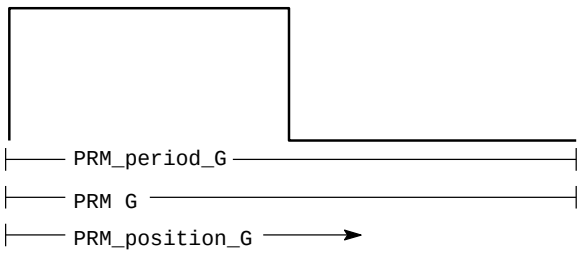
```
      {
      PWM_pin = PWM_ON;
      }
   else
      {
      PWM_pin = PWM_OFF;
      }
   }
```

Listing 33.8 Implementing software PWM

The key to Listing 33.8 is the use of the variables `PWM_position_G`, `PWM_period_G` and `PWM_period_new_G`:

- `PWM_period_G` is the current PRM period. Note that if the update function is scheduled every millisecond, this period is in milliseconds. `PWM_period_G` is fixed during the program execution.
- `PWM_G` represents the current PWM duty cycle (see Figure 33.9).
- `PWM_new_G` is the next PWM duty cycle. This period may be varied by the user, as required. Note that the 'new' value is only copied to `PWM_G` at the end of a PWM cycle, to avoid noise.
- `PWM_position_G` is the current position in the PWM cycle. This is incremented by the update function. Again, the units are milliseconds if these conditions apply.

The link between these various variables and constants is illustrated in Figure 33.9.




```
if (PRM_position_G < PWM_G)
   {
   PRM_pin = PWM_ON:
   }
else
   {
   PRM_pin = PWM_OFF:
   }
```

FIGURE 33.9 Implementing pulse-rate modulation in software

Using a 1 ms scheduler, the PWM frequency (Hz) and PWM resolution (%) we obtain are given by:

$$Frequency_{PWM} = \frac{1000}{2^N}$$

and:

$$Resolution_{PWM} = \frac{1}{2^N} \times 100\%$$

where N is the number of PWM bits you use.

For example, 5-bit PWM allows you to control the output to a resolution of approximately 3%, at a frequency of approximately 31 Hz.

Hardware resource implications

SOFTWARE PWM has no significant hardware resource implications.

Reliability and safety implications

When working with PWM, you may be switching high-power loads and / or mains voltages. You need to take care with start-up states, to ensure that any equipment you are controlling is not going to cause damage or injury when the application starts.

Portability

SOFTWARE PWM uses only core 8051 features: it may be easily used throughout the 8051 family.

Overall strengths and weaknesses

☺ **PWM has been used for many years as efficient means of providing a continuously varying output voltage to 'slow' external components, such as AC and DC motors and large heating elements. It remains a very effective way of controlling such applications, without the need for digital-to-analogue converters.**

☺ **SOFTWARE PWM does not impose a heavy CPU load.**

☹ **SOFTWARE PWM** can only operate at comparatively low frequencies. For example, at a 1 ms tick rate and 8-bit PWM, the PWM frequency is 1000 / 256: that is, less than 4 Hz. Typical hardware PWM units operate at up to 100 *kHz*. For many applications, high-speed PWM is an important consideration: please see **HARDWARE PWM** [page 808] for further information on this issue.

Related patterns and alternative solutions

See **HARDWARE PWM** [page 808] for one alternative.

An alternative implementation of the present pattern is also worth highlighting. Consider the code shown in Listing 33.9.

```
void Fast_Software_PWM_Update(void)
   {
   tWord PWM_position;

   for (PWM_position = 0; PWM_position < PWM_INCREMENTS; PWM_position++)
      {
      if (PWM_position < PWM_G)
         {
         PWM_pin = PWM_ON;
         }
      else
         {
         PWM_pin = PWM_OFF;
         }

      // Appropriate delay here
      PWM_Delay();
      }
   }
```

Listing 33.9 Attempting to increase the frequency of software-only PWM signals

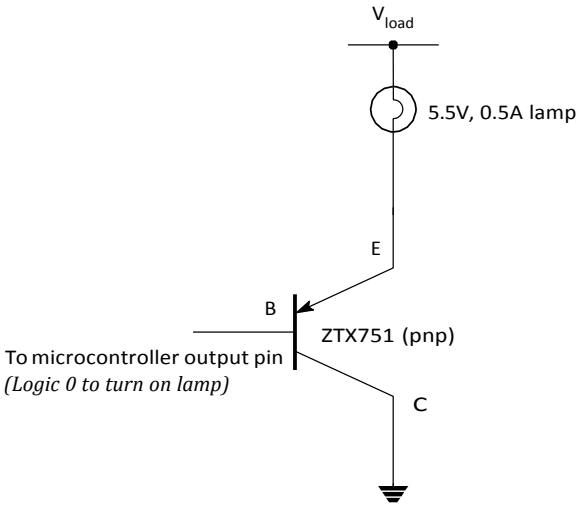
In this solution, we assume that we are generating an entire PWM cycle every time we call the task: this seems to offer the possibility of generating faster PWM signals. For example, at a 0.25 ms tick intervals, we can generate a 4 kHz PWM frequency, which will be adequate for many applications. Moreover, we can generate any number of 'PWM_INCREMENTS': that is, any required bit rate.

However, this solution has two problems. The first is that the CPU is tied up, completely, in the PWM signal generation. This is not insurmountable but might mean, for example, that the solution could only be implemented on a Slave node in a multiprocessor system.

The second problem is more serious and relates to the delays required within the loop. For example, at a 1 ms tick interval, the generation of 5-bit PWM outputs requires the generation of delays of precisely 3.125 μs. This is very difficult to achieve, as we discussed in Chapter 11. At higher bit rates and tick frequencies, the required delays are very much shorter and cannot be obtained in most implementations.

Example: Controlling the brightness of a bulb

We again wish to control the brightness of the small bulb, illustrated in the hardware schematic in Figure 33.10.


V_{load}
5.5V, 0.5A lamp
E
B
ZTX751 (pnp)
To microcontroller output pin
(Logic 0 to turn on lamp)
C

chapter 34

Using digital-to-analog converters (DACs)

Introduction

In the 1980s, if you wanted to create an analog signal from a microcontroller, you would probably have used a 'digital-to-analog converter' (DAC). More recently, as the operating frequencies for digital hardware have grown, pulse-width modulation (PWM) has become a more cost-effective means of creating an analog signal in many circumstances. As a result, use of DACs in areas such as motor control and robotics has become much less common.

However, there are two main sets of circumstances in which use of a DAC is still cost-effective: in high-frequency / high bit-rate applications (particularly audio applications) and in process control.

We consider how to use a DAC to generate analog outputs in this chapter. In doing so we present the following patterns:

- **DAC OUTPUT** [page 841]

DAC OUTPUT

Context

You are developing a 'hard' or 'soft' real-time application, for either a desktop or embedded environment (see Chapter 1 for definitions of these various terms).

Problem

How do you use a digital-to-analog converter (DAC) to generate analog signals?

Background

Suppose you are developing an intruder alarm system for a home environment. In the event of a break-in, we generally expect simply to sound an alarm (ring a bell) to notify the neighbours. This is adequate: the intruder alarm is only intended to detect one kind of problem.

However, suppose you are developing a safety monitoring system for an industrial plant. Various problems may arise: different forms of gas leak, nuclear materials leak, fire, intruder and so forth. In some situations, those in the plant need to take different action (for example, donning different kinds of protective clothing), depending on the type of threat they face.

How do we tell those in the plant what kind of problem has occurred?

One appropriate solution might be to use digitized messages that describe the problem and tell users what to do. For example: 'Warning, a leak of Contaminant 356 has been detected at a level of 345 parts per million. Please use your respirator now and leave the building via your designated escape route.'

To play back this type of message, we will generally require a digital-to-analog converter (DAC).

DACs have an important role, not only in the playback of digitized warnings, but also in other control applications. We consider how to use DACs in this pattern.

Solution

There are several key design decisions that must be made when generating an analog output:

1. You need to determine the required sample rate.
2. You need to determine the required bit rate (DAC resolution).
3. You need to select an 8051 family member with an appropriate DAC or, if necessary, add an external DAC.

4 You may need to shape the frequency response of the output signal.

5 You need to use an appropriate software architecture.

We now deal with each of these points in turn.

Determining the required sample rate

Refer to **SEQUENTIAL ADC** [page 782] for discussions on sample rates.

Determining the required bit rate

Refer to **ONE-SHOT ADC** [page 757] for discussions on bit rates.

8051 microcontrollers with on-chip DACs

While, as we saw in **ONE-SHOT ADC** [page 757], many 8051 microcontrollers contain on-chip ADCs, the number of devices with on-chip DACs is very limited.

Some examples of the available DAC components provided on two recent 8051 devices follow.

Analog Devices ADμC812

Adapted from the Analog Devices[17] data sheet:

The ADμC812 incorporates two 12-bit voltage-mode DACs On-Chip.

DAC operation is controlled via three special function registers. In normal mode of operation, each DAC is updated when the low DAC byte (DACxL) SFR is written. Both DACs can be updated simultaneously using the SYNC bit in the DACCON SFR. The DAC's can operate in 12 or 8-bit modes and have programmable output ranges of 0 -> 2.5V or 0 ->VDD.

Cygnal C8051F000

Adapted from the Cygnal[18] C8051F000 data sheet:

The C8051F000 MCU family has two 12-bit voltage-mode DACs (DAC0, DAC1).

Each DAC has an output swing of 0V to VREF-1LSB for a corresponding input code range of 0x000 to 0xFFF. Using DAC0 as an example, the 12-bit data word is written to the low byte (DAC0L) and high byte (DAC0H) data registers. Data is latched into DAC0 after a write to the corresponding DAC0H register, so the write sequence should be DAC0L followed by DAC0H if the full 12-bit resolution is required. The DAC can be used in 8-bit mode by initializing DAC0L to the desired value (typically 0x00), and writing data to only DAC0H. DAC0 Control Register (DAC0CN) provides a means to enable/disable DAC0 and to modify its input data formatting.

The DAC0 enable/disable function is controlled by the DAC0EN bit (DAC0CN.7). Writing a 1 to DAC0EN enables DAC0 while writing a 0 to DAC0EN disables DAC0. While disabled, the

17. www.analog.com
18. www.cygnal.com

output of DAC0 is maintained in a high-impedance state, and the DAC0 supply current falls to 1uA or less.

In some instances, input data should be shifted prior to a DAC0 write operation to properly justify data within the DAC input registers. This action would typically require one or more load and shift operations, adding software overhead and slowing DAC throughput. To alleviate this problem, the data-formatting feature provides a means for the user to program the orientation of the DAC0 data word within data registers DAC0H and DAC0L. The three DAC0DF bits (DAC0CN.[2:0]) allow the user to specify one of five data word orientations.

DAC1 is functionally the same as DAC0 described above.

Using an external voltage - mode DAC

In general, while the use of an internal DAC will result in increased reliability, smaller system size and lower system cost, this is not always possible.

As with ADCs, both parallel and serial (voltage-mode) DACs are available. For example, in terms of parallel components, Analog Devices[19] produce the parallel AD7245a (12-bit), AD7248a (12-bit) and AD7801 (8-bit), while Maxim[20] produce the Max5480 (8-bit). In serial devices, Maxim (again) produce, for example, the Max517 with an I^2C interface and the Max541 with an SPI interface: refer to Part E for discussion of the use of these interfaces in a time-triggered environment.

Because we will generally be using DACs in circumstances where a high bit-rate and sample frequency is required, the parallel solution will often be the most practical solution.

Using an external current - mode DAC or a transconductance amplifier

As we discussed in **ONE - SHOT ADC** [page 757], there are circumstances, particularly in monitoring or process control, when the use of current-mode components (sensors or actuators) may be more appropriate than the use of voltage-mode devices.

To generate analog current signals from a microcontroller, we have two main options:

- Use a current-mode DAC
- Use a voltage-mode DAC *and* a voltage-to-current converter (often referred to as a *transconductance amplifier*)

The second option may be particularly attractive if your microcontroller has an on-chip DAC.

An example of a suitable DAC component is the Analog Devices[21] AD421, a DAC designed specifically for 4 to 20mA current loops and which is powered by the loop itself.

An example of a transconductance amplifier is the XTR 110 from Burr Brown.[22] This can be used, for example, to convert a 1–5V analog voltage (from, say, a microcontroller

19. www.analog.com
20. www.maxim-ic.com
21. www.analog.com
22. www.burr-brown.com

voltage-mode DAC output) into the 4–20mA range required in process-control applications. As such, it can be a useful component in some 'intelligent' process sensors.

Shaping the frequency response

Use of a DAC will introduce 'noise' (through aliasing effects) and frequency distortions. In most cases, the aliasing effects, at least, must be eliminated: see **DAC SMOOTHER** [page 853] for further details.

General implications for the software architecture

The use of a DAC at high frequencies (10 kHz or 16 kHz) will have a major impact on the overall architecture of your application. For example, even at 10 kHz, you may require a 0.1 ms tick interval. This imposes a substantial load on a basic 8051 device.

In general, only 8051 devices which operate fewer than 12 clock cycles per instruction can provide these levels of performance. Use of recent devices such as the Dallas 89C420 (a 'Standard 8051' with 1 clock cycle per instruction, operating at up to 50 MHz: see Chapter 3) can make it practical to operate at 16 kHz (0.0625 ms tick interval).

Hardware resource implications

Use of the internal ADC will generally mean that at least one input pin is unavailable for other purposes; use of an external ADC will require the use of larger numbers of port pins.

Use of the ADC may also have an impact on the power consumption of your microcontroller.

Reliability and safety implications

In general, use of on-chip DACs is likely to improve the system reliability (compared with an off-chip solution), since the hardware (and, possibly, software) complexity, number of soldered joints, etc., are all greatly reduced.

More specifically, use of a DAC with a long conversion time may introduce delays that will impact on the general performance of signal-processing applications and which may impact on the stability of control applications. Make sure the speed of the DAC is an appropriate match for your intended application.

Portability

All DAC components vary. The principles and basic techniques are portable but the details will always be heavily hardware dependent.

Overall strengths and weaknesses

☺ **DAC outputs are essential in many applications and are generally easy to use.**

☹ Microcontrollers with on-board DACs are comparatively rare and more expensive than those without such facilities.

Related patterns and alternative solutions

In many cases, PWM outputs (see Chapter 33) provide a low-cost and effective alternative to the use of DACs: see **HARDWARE PWM** [page 808].

See **DAC SMOOTHER** [page 853] for techniques suitable for removing 'noise' (introduced through aliasing effects) and frequency distortions introduced by the use of DACs.

See **DAC DRIVER** [page 857] for information about the hardware needed to drive high-power loads, such as loudspeakers or DC motors.

Example: Speech playback using a 12-bit parallel DAC

Here we consider how we can use a 12-bit parallel DAC to play back a speech sample at a 10 kHz sample rate.

Figure 34.1 shows the speech waveform.

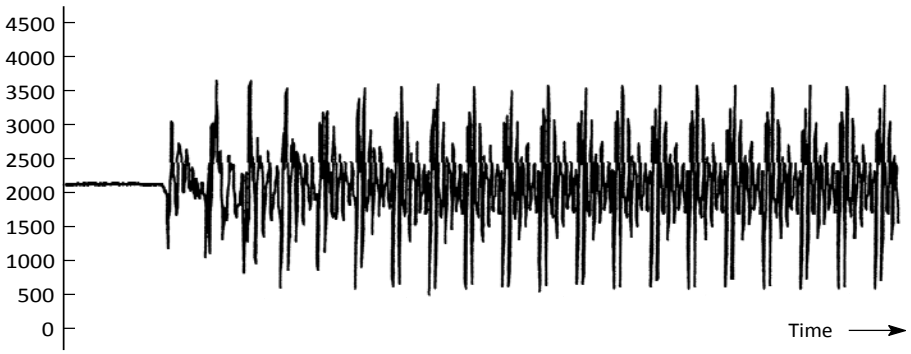


FIGURE 34.1 An example of a fragment of speech

Figure 34.2 shows the basic hardware used. Note that the necessary smoothing and amplification components will be discussed in **DAC SMOOTHER** [page 853] and **DAC DRIVER** [page 857].

chapter 35

Taking control

Introduction

The focus of the chapter is on proportional-integral-differential (PID) control. PID is both simple and effective: as a consequence it is the most widely used control algorithm. The focus here will be on techniques for designing and implementing PID controllers for use in embedded applications.

PID CONTROLLER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you design and implement a PID control algorithm?

Background

We consider in this section why we need 'closed-loop' control algorithms.

Open-loop control

Suppose we wish to control the speed of a DC motor, used as part of an air-traffic control application (Figure 35.1). To control this speed, we will assume that we have decided to change the applied motor voltage using a DAC.[25]

As described, this is an example of a much more general 'open-loop' control approach (Figure 35.2).

FIGURE 35.1 A radar used as part of an air-traffic control application

25. We could use PWM control here: this would, however, simply complicate the example and would not alter the general conculsions

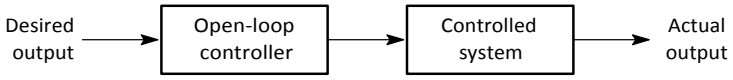


FIGURE 35.2 A schematic representation of an open-loop control system

Using open-loop control, we need to know what the system parameters are that will be required to create the desired system outputs. Thus, in the case of our air-traffic system, having knowledge of the motor, radar hardware and the motor drive circuit, we can set the speed of rotation that we require.

In an ideal world, this type of open-loop control system would be easy to design: we would simply have a lookup table linking the required motor speed to the required output parameters. For example, consider the DC motor to be used at the heart of this navigation system. We might start by assuming that this motor has a speed of operation directly proportional to the applied voltage (Figure 35.3).

As a result, to set the required speed of rotation, we should be able to use a lookup table (Table 35.1).

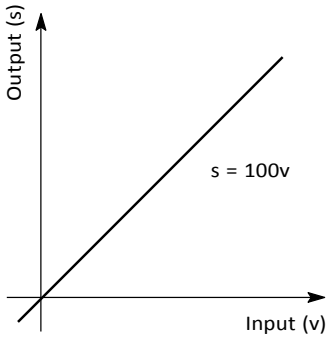


FIGURE 35.3 A simple model predicting the likely speed of rotation of a DC motor as the voltage is varied

TABLE 35.1 Translating the simple model in Figure 35.3 into a lookup table

Radar rotation speed (RPM)	DAC setting (8-bit)
0	0
2	51
4	102
6	153
8	204
10	255

This is an example of a general, linear, single-input single-output (SISO) system: this type of system can be represented graphically as shown in Figure 35.4.

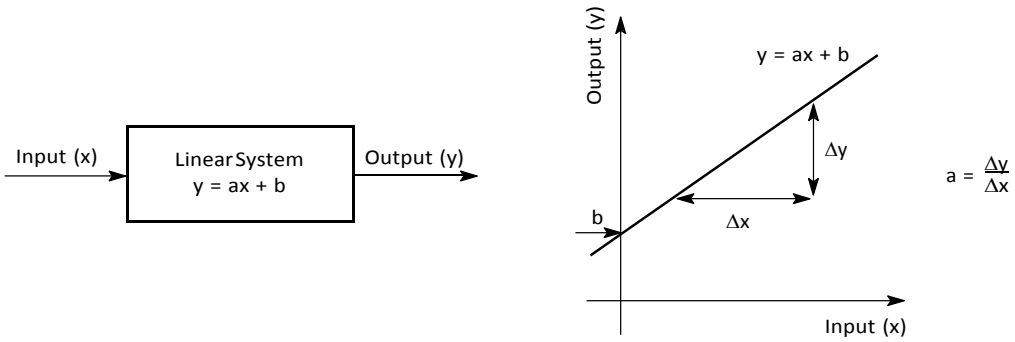


FIGURE 35.4 Modelling a linear system

Unfortunately, such linearity is very rare in practical systems. For example, our practical motor will have a maximum input voltage and a corresponding maximum speed of rotation (Figure 35.5). In addition, our practical motor will not begin rotating until a certain minimum voltage has been reached and will not abruptly stop at a maximum speed of rotation, but will have an I-O curve something like that shown in Figure 35.6.

Overall, our real motor is a non-linear system: it cannot be accurately represented by a simple linear ('straight line') model. Simply by inspecting Figure 35.6, we can see that to write down a description of this curve (that is, create a model of this motor) is going to be more complex than describing the simple linear model in Figure 35.3. Nonetheless, our (open-loop) lookup table solution can be adapted to deal with this non-linearity (Table 35.2).

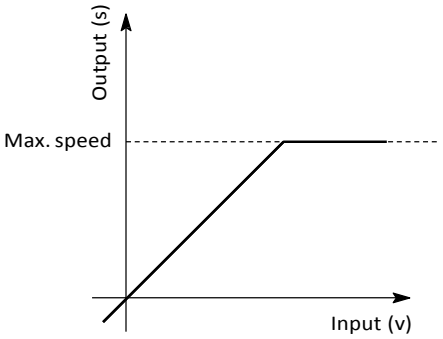


FIGURE 35.5 A slightly more realistic model predicting the likely speed of rotation of a DC motor as the voltage is varied

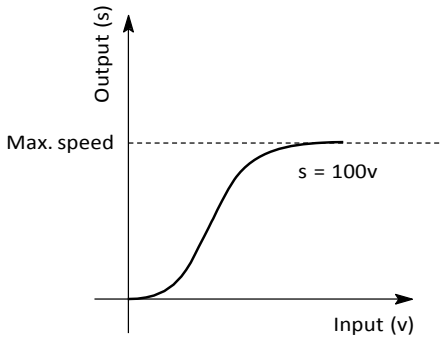


FIGURE 35.6 A further improvement on the DC motor model

TABLE 35.2 Translating the model in Figure 35.6 into a lookup table

Radar rotation speed (RPM)	DAC setting (8-bit)
0	0
2	61
4	102
6	150
8	215
10	255

However, this is not the only problem we have to deal with. Table 35.2 has, we assume, been created as a result of a series of tests on the real system. It therefore takes into account the non-linear nature of the motor and other system components. However, in addition to their non-linearity, most real systems also demonstrate characteristics which vary with time. In the case of the radar system, Table 35.2 does not take into account the wind speed or wind direction. As a result, this table of values is only valid (say) in still (wind-free) conditions.

If we are determined to use an open-loop approach, we will need to measure the wind speed and wind direction (Figure 35.7). Then we will need to create another table for 5 mph SE winds and another for 10 mph NW winds and so on.

Overall, this approach to control system design quickly becomes impractical.

Closed-loop control

The fundamental problem with the open-loop version of the radar control system is that the controller is 'blind': it receives no feedback about the system output (in this case the speed of rotation) which it is trying to control.

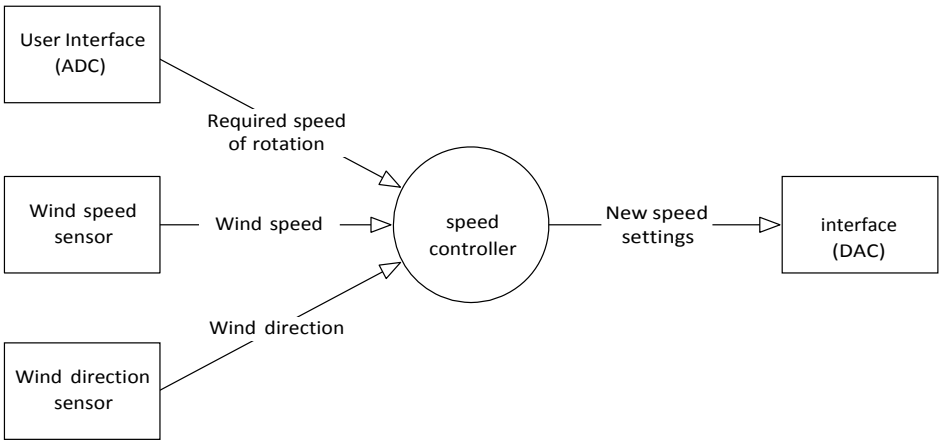


FIGURE 35.7 Taking the open-loop radar control system to its logical extreme

Consider the equivalent closed-loop version of the radar application (Figure 35.8). The key feature of this form of controller is the feedback loop (see Figure 35.9). This allows the system to respond effectively – for example – to external disturbances such as changes in wind speed or direction.

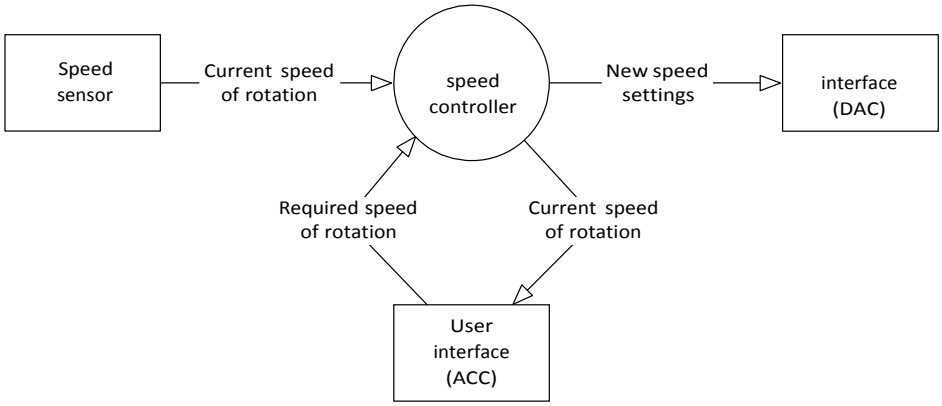


FIGURE 35.8 Controlling the speed of rotation of a radar in an air-traffic application (closed-loop version)

What closed - loop algorithm should you use?

There are numerous possible control algorithms that can be employed in the box marked 'closed-loop controller' in Figure 35.9 and the development and evaluation of new algorithms is an active area of research in many universities. A detailed discussion of some of the possible algorithms available is given by Nise (1995), Dutton *et al.* (1997) and Dorf and Bishop (1998).

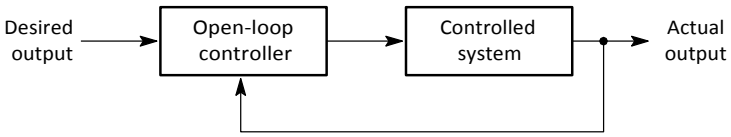


FIGURE 35.9 A schematic representation of a closed-loop control system

Despite the range of algorithms available, proportional-integral-differential (PID) control is found to be very effective in many cases and, as such, it is generally considered the 'standard' against which alternative algorithms are judged. Without doubt, it is the most widely used control algorithm in the world at the present time.

Solution

In this section we consider how to implement a PID-based control algorithm using a microcontroller.

What is PID control?

If you open a textbook on control theory, you will encounter a description of PID control containing an equation similar to that shown in Figure 35.9.

This may appear rather complex, but can – in fact – be implemented very simply, if you understand what the algorithm is trying to achieve.