CHAPTER 9

Interrupts, Timers, and Tasks

9.1 Time and Tasks in Embedded Systems

9.1.1 Timers and Interrupts

The very first diagram in this book, Fig. 1.1, shows the key features of an embedded system. Among these is *time*. Embedded systems have to respond in a timely manner to events as they happen. Usually, this means they have to be able to do the following:

- measure time durations;
- generate time-based events, which may be single or repetitive; and
- respond with appropriate speed to external events, which may occur at unpredictable times.

In doing all of these, the system may find that it has a conflict of interest, with two actions needing attention at the same time. For example, an external event may demand attention just when a periodic event needs to take place. Therefore, the system may need to distinguish between events which have a high level of urgency, and those which don't, and take action accordingly.

It follows that we need a set of tools and techniques to allow effective time-based activity to occur. Key features of this toolkit are interrupts and timers, the subject of this chapter. In brief, a timer is just what its name implies, a digital circuit which allows us to measure time, and hence makes things happen when a certain time has elapsed. An interrupt is a mechanism whereby a running program can be interrupted, with the central processing unit (CPU) then being required to jump to some other activity. In our study of interrupts and timers, we make major steps forward in our understanding of how programs can be structured, and how we can move toward more sophisticated program design. That's where the third topic in our chapter title comes in—the concept of program *tasks*.

9.1.2 Tasks

In almost all embedded programs, the program has to undertake a number of different activities. Entering briefly the world of small-scale farming, consider the requirements for a temperature controller for a mushroom-growing shed, working in a cold environment. These friendly little fungi grow best under tightly controlled conditions of temperature and

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00009-8

humidity. Their growth goes through separate phases, at which time different preferred temperatures may apply. The system will need to control, display, and log the temperature, keep track of time, respond to changes in setting by the user, and control the heater and fan. Each is a fairly distinct activity, and each will require a block of code. In programming terminology, we call these distinct activities *tasks*. Our ability to partition any program into tasks becomes an important skill in more advanced program design. Once a program has more than one task, we enter the domain of *multitasking*. As tasks become more, the challenge of responding to the needs of all tasks becomes greater, and many techniques are developed to do this.

9.1.3 Event-Triggered and Time-Triggered Tasks

Tasks performed by embedded systems tend to fall into two categories, *event-triggered* and *time-triggered*. Tasks which are event-triggered occur when a particular external event happens, at a time which is usually not predictable. Tasks which are time-triggered happen periodically, at a time determined by the microcontroller. To continue with our example of the mushroom shed, Table 9.1 lists some possible tasks, and suggests whether each is event- or time-triggered. For those which are time-triggered, it becomes necessary to indicate how frequently the tasks occur; suggestions for this are also shown.

9.1.4 Working in "Real Time"

The techniques which this chapter introduces make us aware of a range of timing challenges, and lead to an ability to develop systems which operate in *real time*. A simple but completely effective definition of real time, already adopted in Ref. [1] of Chapter 1, is as follows:

A system operating in real time must be able to provide the correct results at the required time deadlines.

Notice that this definition carries no implication that working in real time implies high speed, although this can often help. It simply states that what is needed must be ready at the time when it is needed.

Table 9.1: Example tasks—temperature controller for a mushroom shed.

Task	Event- or Time-Triggered
Measure temperature	Time (every minute)
Compute and implement heater and fan settings	Time (every minute)
Respond to user control	Event
Record and display temperature	Time (every minute)
Orderly switch to battery backup in case of power loss	Event

9.2 Responding to External Events

9.2.1 Polling

A simple example of an event-triggered activity is when a user pushes a button. This can happen at any time, without warning, but when it does the user expects a response. One way of programming for this is to continuously test that external input. This is illustrated in Fig. 9.1, where a program is structured as a continuous loop. Within this, it tests the state of two input buttons, and responds to them if activated. This way of checking external events is called *polling*, the program ensures that it periodically checks input states, and responds if there is a need. This is the sort of approach we have used so far in this book, whenever needed. It works well for simple systems; we will find that it is not adequate for more complex programs.

Suppose the program of Fig. 9.1 is extended, so that a microcontroller has 20 input signals to test in each loop. On most loop iterations, the input data may not even change, so we are running the polling for no apparent benefit. Worse, the program might spend time checking the value of unimportant inputs, while not recognizing very quickly when a major fault condition has arisen.

There are two main problems with polling:

1. The processor can't perform any other operations during a polling routine;
2. All inputs are treated as equal; the urgent change has to wait its turn before it's recognized by the computer.

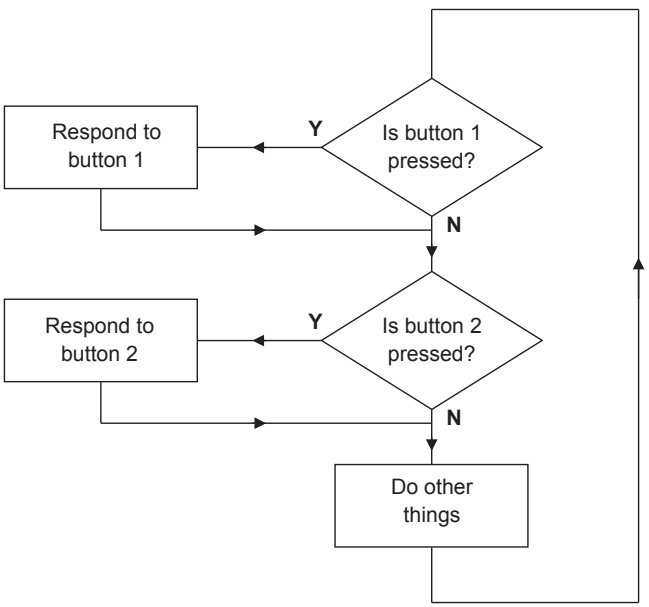

Figure 9.1
A simple program using polling.

A better solution is for input changes to announce themselves; time is not wasted finding out that there is no change. The difficulty lies in knowing when the input value has changed. This is the purpose of the interrupt system.

9.2.2 Introducing Interrupts

The interrupt represents a radical alternative to the polling approach just described. With an interrupt, the hardware is designed so that the external variable can stop the CPU in its tracks and demand attention. Suppose you lived in a house and were worried that a thief might come in during the night. You *could* arrange an alarm clock to wake you up every half hour to check there was no thief, but you wouldn't get much sleep. In this case, you would be *polling* the possible thief "event." Alternatively, you could fit a burglar alarm. You would then sleep peacefully, *unless* the alarm went off, interrupted your sleep, and you would jump up and chase the burglar. In very simple terms, this is the basis of the computer interrupt.

Interrupts have become a hugely important part of the structure of any microprocessor or microcontroller, allowing external events and devices to force a change in CPU activity. In early processors, interrupts were mainly used to respond to really major external events; designs allowed for just one, or a small number of interrupt sources. The interrupt concept was, however, found to be so useful that more and more possible interrupt sources were introduced, sometimes for dealing with rather routine matters.

In responding to interrupts, most microprocessors follow the general pattern of the flow diagram shown in Fig. 9.2. The CPU completes the current instruction it is executing. It's about to go off and find a completely different piece of code to execute, so it must save key information about what it has just been doing; this is called the *context*. The context includes at least the value of the *program counter* (this tells the CPU where it should come back to when the interrupt has completed), and generally a set of key registers, for example, those holding current data values. All this is saved on a small block of memory local to the CPU, called the *Stack*. The CPU then runs a section of code called an *Interrupt service routine* (ISR); this has been specifically written to respond to the interrupt which has occurred. The address of the ISR is found through a memory location called the *Interrupt vector*. On completing the ISR, the CPU returns to the point in the main code immediately after the interrupt occurred, finding this by retrieving the program counter from the stack where it left it. It then continues program execution as if nothing happened. Down in Section 9.4, we add to this explanation, and set it more in the context of the mbed.

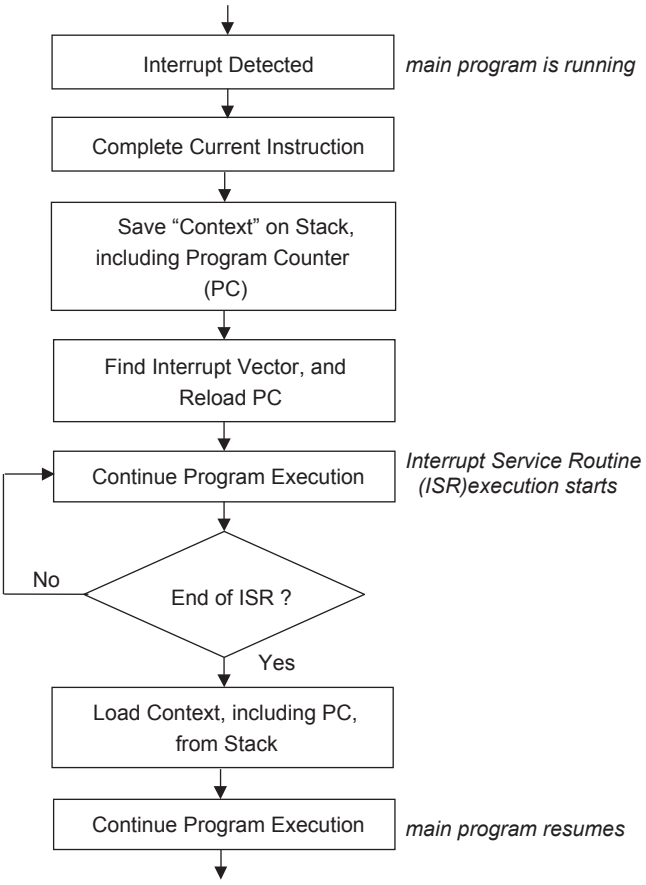

Figure 9.2
A typical microprocessor interrupt response.

9.3 Simple Interrupts on the mbed

The mbed application programming interface (API) exploits only a small subset of the interrupt capability of the LPC1768 microcontroller, mainly focusing on the external interrupts. Their use is very flexible, as any of pins 5 to 30 can be used as an interrupt input, excepting only pins 19 and 20. The available API functions are shown in Table 9.2. Using these, we can create an interrupt input, write the corresponding ISR, and link the ISR with the interrupt input.

Program Example 9.1 is a very simple interrupt program, adapted from the mbed website. It is made up of a continuous loop, which switches on and off LED4 (labeled **flash**). The interrupt input is specified as pin 5, and labeled **button**. A tiny ISR is written for it, called

Table 9.2: Application programming interface interrupts summary.

Function	Usage
`InterruptIn`	Create an InterruptIn connected to the specified pin
`rise`	Attach a function to call when a rising edge occurs on the input
`rise`	Attach a member function to call when a rising edge occurs on the input
`fall`	Attach a function to call when a falling edge occurs on the input
`fall`	Attach a member function to call when a falling edge occurs on the input
`mode`	Set the input pin mode

ISR1, which is structured exactly as a function. The address of this function is attached to the rising edge of the interrupt input, in the line

```
button.rise(&ISR1);
```

When the interrupt is activated, by this rising edge, the ISR executes, and LED1 is toggled. This can occur at any time in program execution. The program has effectively one time-triggered task, the switching of LED4, and one event-triggered task, the switching of LED1.

```
/* Program Example 9.1: Simple interrupt example. External input causes interrupt,
while led flashes
                                                                                */
#include "mbed.h"
InterruptIn button(p5);    //define and name the interrupt input
DigitalOut led(LED1);
DigitalOut flash(LED4);

void ISR1() {              //this is the response to interrupt, i.e. the ISR
  led = !led;
}

int main() {
  button.rise(&ISR1);      // attach the address of the ISR function to the
                                        // interrupt rising edge
  while(1) {               // continuous loop, ready to be interrupted
    flash = !flash;
    wait(0.25);
  }
}
```

Program Example 9.1: Introductory use of an interrupt

Compile and run Program Example 9.1, applying the very simple build shown in Fig. 9.3. Notice when you push the button, the interrupt is taken high and LED1 changes

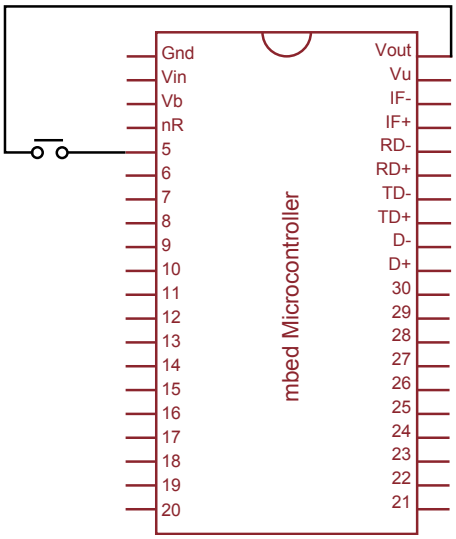

Figure 9.3
Circuit build for Program Example 9.1.

state; LED4 meanwhile continues its flashing, almost unperturbed. The program depends on the internal pull-down resistor, which is enabled by default during setup. *If* you experience erratic behavior with this program, you may be experiencing switch bounce. In this case, fast forward to Section 9.10, for an introduction to this important topic.

Exercise 9.1

Change Program Example 9.1 so that:

1. The interrupt is triggered by a falling edge on the input.
2. There are two ISRs, from the same push-button input. One toggles LED1 on a rising interrupt edge, and the other toggles LED2 on a falling edge.

■

9.4 Getting Deeper Into Interrupts

We can take our first generalized understanding of interrupts further, and try to understand a little more what goes on inside the mbed. Let's point out straight away that this is not essential as far as using the mbed API is concerned. In fact, although the LPC1768 microcontroller has a very sophisticated interrupt structure, the mbed uses only a small part of this, and that in only a modest way. Therefore, go on straight away to the next Section if you don't want to get into any deeper interrupt detail.

Okay, so you're still there! Let's extend our ideas of interrupts. While we said earlier that an interrupt was possibly like a thief coming in the night, imagine now a different scenario. Suppose you are a teacher of a big class made up of enthusiastic but poorly behaved kids; you've set them a task to do but they need your help. Tom calls you over, but while you're helping him, Jane starts clamoring for attention.

Do you:

tell Jane to be quiet and wait until you've finished with Tom?
OR
tell Tom you'll come back to him, and go over to sort Jane out?

To make matters worse, your school principal has asked you to let the members of the school band out of class half an hour early, but you really want them to finish their work before they go. This influences the above decision. Suppose Jane's in the band, but Tom isn't. Therefore in this situation, you decide you must leave Tom to help Jane. This school classroom situation is reflected in almost any embedded system. There could be a number of interrupt sources, all possibly needing attention. Some will be of great importance, others much less. Therefore most processors contain four important mechanisms:

- Interrupts can be *prioritized*, in other words some are defined as more important than others. If two occur at the same time, then the higher priority one executes first.
- Interrupts can be *masked*, i.e., switched off, if they are not needed, or are likely to get in the way of more important activity. This masking could be just for a short period, for example, while a critical program section completes.
- Interrupts can be *nested*. This means that a higher priority interrupt can interrupt one of lower priority, just like the teacher leaving Tom to help Jane. Working with nested interrupts increases the demands on the programmer and is strictly for advanced players only. Not all processors permit nested interrupts, and some allow you to switch nesting on or off.
- The location of the ISR in memory can be selected, to suit the memory map and programmer wishes.

Let's take on just a couple more important interrupt concepts, these ones from the point of view of the interrupt source. Go to the moment in the above scenario when Jane suddenly realizes she needs help, and puts her hand up. Some short time later, the teacher comes over. The delay between her putting up her hand, and the teacher actually arriving, is called the interrupt *latency*. Latency may be due to a number of things, in this case, the teacher has to notice Jane's hand in the air, may need to finish with another pupil, and then actually has to walk over. Once the teacher arrives, Jane puts her hand

down. While Jane is waiting with her hand in the air, patiently we hope, her interrupt is said to be *pending*.

These concepts and capabilities hint at some of the deep magic that can be achieved with advanced interrupt structures.

We can put all of this into more technical terms, and hence refine our understanding of interrupt action. This was first illustrated in the flow diagram of Fig. 9.2. Further detail on part of this figure is now shown in Fig. 9.4. The interrupt being asserted is like Jane putting up her hand. In a microprocessor, the interrupt input will be a logic signal; depending on its input configuration, it may be active high or low, or triggered by a rising or falling edge. This input will cause an internal *flag* to be set. This is normally just a single bit in a register, which records the fact that an interrupt has occurred. This doesn't necessarily mean that the interrupt automatically gets the attention it seeks. If it's not enabled (i.e., it is masked), then there will be no response. The flag is left high, however, as the program might later enable that interrupt, or the program may just poll the interrupt flag. Back to the flow diagram—if another ISR is already running, then again the incoming interrupt may not get a response, at least not immediately. If it's higher priority and nested interrupts are allowed, then it will be allowed to run. If it's lower priority, it will have to wait for the other ISR to complete. The subsequent actions in the flow diagram, as already seen in Fig. 9.2, then follow. Note that the figure is potentially misleading, as it implies these actions happen in turn. To get low latency, they should happen as fast as possible; a

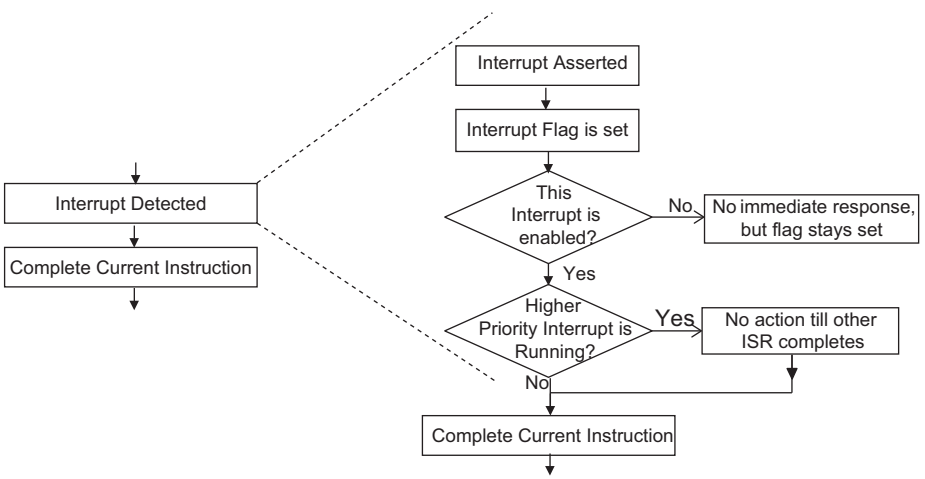

Figure 9.4
A typical microprocessor interrupt response—some greater detail.

good interrupt management system will allow some of the actions to take place in parallel. For example, the interrupt vector could be accessed while the current instruction is completing.

9.4.1 Interrupts on the LPC1768

Now let's get back to microprocessor hardware. Recall that the mbed contains the LPC1768 microprocessor, and that the LPC1768 contains the ARM Cortex core. It is to the Cortex core that we must turn first, as it provides the interrupt structure used by the microcontroller. Back in Fig. 2.3, we saw the Cortex core, set within the LPC1768 microprocessor. Management of all interrupts in the Cortex is undertaken by the formidable-sounding *nested vectored interrupt controller* (NVIC). You could think of this as managing the processes overviewed in Figs. 9.2 and 9.4. The NVIC is also a bit like a digital electronic control box with a lot of unconnected wires hanging out. When the Cortex is embedded into a microcontroller, such as the LPC1768, the chip designer assigns and configures those features (through the "loose wires") of the NVIC which are needed for that application. For example, the Cortex allows for 240 possible interrupts, both external and from the peripherals, and 256 possible priority levels. The LPC1768, however, has "only" 33 interrupt sources, with 32 possible programmable priority levels.

9.4.2 Testing Interrupt Latency

Now that we've met the concept of interrupt latency, let's test it in the mbed. Program Example 9.2 adapts Program Example 9.1, but the interrupt is now generated by an external square wave, instead of an external button push. This makes it easier to see on an oscilloscope. When an interrupt occurs, the external LED is pulsed high for a fixed duration.

```
/* Program Example 9.2: Tests interrupt latency. External input causes interrupt,
which pulses external LED while LED4 flashes continuously.
                                                                              */
#include "mbed.h"
InterruptIn squarewave(p5);     //Connect input square wave here
DigitalOut led(p6);
DigitalOut flash(LED4);

void pulse() {                  //ISR sets external led high for fixed duration
  led = 1;
  wait(0.01);
  led = 0;
}
```

```
int main() {
  squarewave.rise(&pulse);      // attach the address of the pulse function to
                                                          // the rising edge
  while(1) {                    // interrupt will occur within this endless loop
    flash = !flash;
    wait(0.25);
  }
}
```

Program Example 9.2: Testing interrupt latency

Create a new project from Program Example 9.2. Connect an external LED between pin 6 and ground, ensuring correct polarity. Connect to pin 5 a signal generator set to logic-compatible square wave output (probably labeled "TTL compatible"), running initially at around 10 Hz. Once connected, with the program running, the external LED should flash at this rate, i.e., around 10 times a second.

Exercise 9.2

Connect two inputs of an oscilloscope to the interrupt input, and the LED output, triggering from the interrupt. Increase the input frequency to around 50 Hz. Set the oscilloscope time base to 5 μs per division. You should be able to see the rising edge of the interrupt input, and a few microseconds later the LED output rising. The time delay between the two is an indication of latency. The rise of the LED will be flickering a little, as the delay will depend on what the CPU is doing at the moment the interrupt occurs. It's important to note that the latency as measured here depends on both hardware and software factors.

■

9.4.3 Disabling Interrupts

Interrupts are an essential tool in embedded design. But because they can occur at any time, they can have unexpected or undesirable side effects. There are a number of situations where it is essential to disable (mask) the interrupt. This can include when you're undertaking a time-sensitive activity, or a complex calculation which must be completed in one go. As any incoming interrupt will have left its flag set, it can be responded to once interrupts are enabled again. Of course a delay in response has been introduced, and the latency much compromised.

A feature of the compiler allows interrupts to be disabled, as shown here.

```
___disable_irq();                          //disable interrupts
//activity which can't be interrupted
__enable_irq();                            //enable interrupts
```

Note that each line starts with *two* underscores.

Exercise 9.3

Using Program Example 9.2, disable the interrupt for the duration of the `wait (0.25)` delay. Connect the "scope" as in Exercise 9.2, and observe the output again. Comment on the outcome.

Experiment with different values for this wait function. Try then splitting it into two waits, running immediately after each other, with one having interrupts disabled, and the other enabled.

■

9.4.4 Interrupts From Analog Inputs

Aside from digital inputs, it is useful to generate interrupts when analog signals change, for example, if an analog temperature sensor exceeds a certain threshold. One way to do this is by applying a *comparator*. A comparator is just that, it compares two input voltages. If one input is higher than the other, then the output switches to a high state; if it is lower, the output switches low. A comparator can easily be configured from an operational amplifier (op amp), as shown in Fig. 9.5. Here, an input voltage, labeled V_{in}, is compared to a threshold voltage derived from a potential divider, made from the two resistors R_1 and R_2. These are connected to the supply voltage, labeled V_{sup}. For the right choice of op amp or comparator, and with suitable supply voltages, the output is a Logic 1 when the input is above the threshold value, and Logic 0 otherwise. The threshold voltage just mentioned, labeled $V_{\_}$ in the diagram, is calculated using Eq. (9.1).

$$V_{\_} = V_{\text{sup}}\, R_2/(R_1 + R_2) \tag{9.1}$$

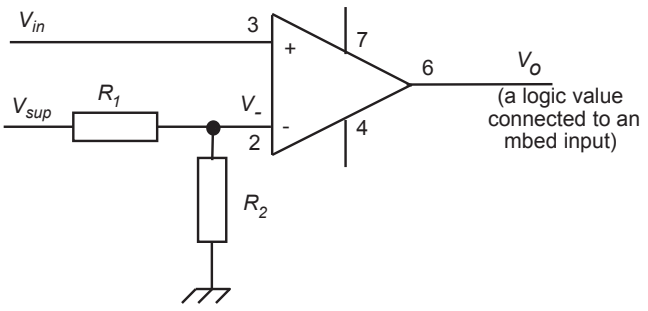

Figure 9.5
A comparator circuit.

As an example, we might want to generate an interrupt from a temperature input, using an LM35 temperature sensor (Fig. 5.8), with the interrupt triggered if the temperature exceeds 30°C. As we know, this sensor has an output of 10 mV/°C, which would lead to an output voltage of 300 mV at the trigger point proposed. To set V_- to 300 mV in Fig. 9.5, we apply Eq. (9.1), with a V_{sup} value of 3.3 V; we find that $R_1 = 0.1R_2$. Values of $R_1 = 10k$, and $R_2 = 1k$ could therefore be chosen.

Exercise 9.4

Unlike many op amps, the ICL7611 can be run from very low supply voltages, even the 3.3 V of the mbed. Using an LM35 IC temperature sensor and an ICL7611 op amp connected as a comparator, design and build a circuit which causes an interrupt when the temperature exceeds 30°C. Write a program which lights an LED when the interrupt occurs. The pin connections shown in Fig. 9.5 can be applied. Pin 7 is the positive supply, and can be connected to the mbed 3.3 V; pin 4 is the negative supply, and is connected to 0 V. For this op amp, also connect pin 8 to the positive supply rail. (This pin controls the output drive capability; check the data sheet for more information.)

■

9.4.5 Conclusion on Interrupts

We have had a good introduction to interrupts, and the main concepts associated with them. They become an absolutely essential part of the toolkit of any embedded designer. We have so far limited ourselves to single-interrupt examples. Where multiple interrupts are used, the design challenges become considerably greater—interrupts can have a very destructive effect if not used well. The design of advanced multiple interrupt programs is, however, beyond the scope of this book.

9.5 An Introduction to Timers

In a simple program, for example, our very first Program Example 2.1, we use **wait( )** functions to perform timing operations, for example, to introduce a delay of 200 ms. This is easy and very convenient, but during this delay loop, the microcontroller can't perform any other activity; the time spent waiting is just wasted time. As we try and write more demanding programs, this simple timing technique just becomes inadequate. We need a way of letting timing activity go on in the background, while the program continues to do useful things. We turn to the digital hardware for this.

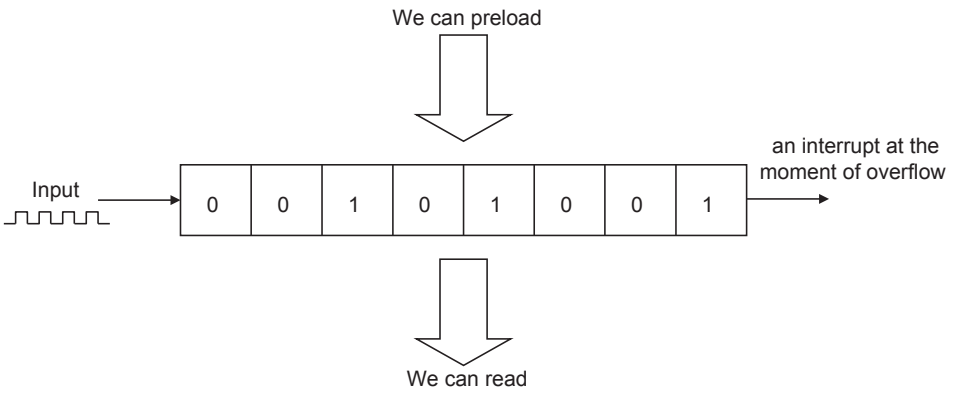

Figure 9.6
A simple 8-bit counter.

9.5.1 The Digital Counter

It is an easy task in digital electronics to make electronic counters; you simply connect together a series of bistables or flip-flops. Each one holds 1 bit of information, and all those bits together form a digital word. This is illustrated in very simple form in Fig. 9.6, with each little block representing one flip-flop, each holding 1 bit of the overall number. If the input of this arrangement is connected to a clock signal, then the counter will count, in binary, the number of clock pulses applied to it. It is easy to read the overall digital number held in the counter, and it is not difficult to arrange the necessary logic to preload it with a certain number, or to clear it to zero.

The number that a counter can count up to is determined by the number of bits in the counter. In general, an n-bit counter can count from 0 to $(2^n - 1)$. For example, an 8-bit counter can count from 0000 0000 to 1111 1111, or 0 to 255 in decimal. Similarly, a 16-bit counter can count from 0 to 65,535. If a counter reaches its maximum value, and the input clock pulses keep on coming, then it overflows back to zero, and starts counting up all over again. All is not lost if this happens—in fact, we have to be ready to deal with it. Many microcontroller counters cause an interrupt as the counter overflows; this interrupt can be used to record the overflow, and the count can continue in a useful way.

9.5.2 Using the Counter as a Timer

The input signal to a counter can be a series of pulses coming from an external source, for example, counting people going through a door. Alternatively, it can be a fixed frequency logic signal, such as the clock source within a microcontroller. Very importantly, if that clock source is a known and stable frequency, then the counter becomes a timer. As an example, if the clock frequency is 1.000 MHz (hence with period of 1 μs), then the count

will update every microsecond. If the counter is cleared to zero, and then starts counting, the value held in the counter will give the elapsed time since the counting started, with a resolution of 1 μs. This can be used to measure time, or trigger an event when a certain time has elapsed. It can also be used to control time-based activity, for example, serial data communication, or a PWM stream.

Alternatively, if the counter is just free-running with a continuous clock signal, then the "interrupt on overflow" occurs repeatedly. This becomes very useful where a periodic interrupt is needed. For example, if an 8-bit counter is clocked with a clock frequency of 1MHz, it will reach its maximum value and overflow back to zero in 256 μs (it's the 256th pulse which causes the overflow from 255 to 0). If it's left running continuously, then this train of interrupt pulses can be used to synchronize timed activity, for example, it could define the baud rate of a serial communication link.

Timers based on these principles are an incredibly important feature of any microcontroller. Indeed, most microcontrollers have many more than one timer, applied to a variety of different tasks. These include generating timing in PWM or serial links, measuring the duration of external events, and producing other timed activity.

9.5.3 Timers on the mbed

To find out what hardware timers the mbed has, we turn back to Fig. 2.3 and Ref. [5] of Chapter 2, the LPC1768 user manual. We find that the microcontroller has four general-purpose timers, a *repetitive interrupt timer*, and a *system tick timer.* All are based on the principles just described. The mbed makes use of these in three distinct applications, described in the sections which follow. These are the timers, used for simple timing applications, timeout, which calls a function after a predetermined delay, and ticker, which repeatedly calls a function, at a predetermined rate. It also applies a *real-time clock* (RTC) to keep track of time of day, and date.

9.6 Using the mbed Timer

The mbed timer allows basic timing activities to take place, for comparatively short time durations. A timer can be created, started, stopped, and read. There is no limit on the number of timers that can be set up. The API summary is shown in Table 9.3. The mbed site notes that the timer is based on the 32-bit counters, and can time up to a maximum of $(2^{31}-1)$ microseconds, i.e., something over 30 min. Based on the theory above, one might expect the timer to count up to $(2^{32}-1)$. However, 1 bit is reserved in the API object as a sign bit, so only 31 bits are available for counting.

Program Example 9.3 gives a simple but interesting timing example, and is taken from the mbed site. It measures the time taken to write a message to the screen, and displays that

Table 9.3: Application programming interface summary for timer.

Function	Usage
start	Start the timer
stop	Stop the timer
reset	Reset the timer to 0
read	Get the time passed in seconds
read_ms	Get the time passed in milliseconds
read_us	Get the time passed in microseconds

message. Compile and run the program, with CoolTerm or Tera Term activated (Appendix E). Then, do Exercise 9.5 to make some further measurements and calculations.

```
/* Program Example 9.3: A simple timer example, from mbed website.
Activate host terminal to test.
                                                                    */
#include "mbed.h"
Timer t;                              // define Timer with name “t”
Serial pc(USBTX, USBRX);

int main() {
  t.start();                          //start the timer
  pc.printf("Hello World!\n");
  t.stop();                           //stop the timer
  pc.printf("The time taken was %f seconds\n", t.read());  //print to pc
}
```

Program Example 9.3: A simple timer application

Exercise 9.5

Run Program Example 9.3, and note from the computer screen readout the time taken for the message to be written. Then, write some other messages, of differing lengths, and record in each case the number of characters, and the time taken. Can you relate the times taken to the baud rate used? Can you make any other deductions? If necessary, check Section 7.9, to recall some of the timing issues relating to the asynchronous serial data link used.

■

9.6.1 Using Multiple mbed Timers

We are now going to apply the timer in a different way, to run one function at one rate and another function at another rate. Two LEDs will be used to show this; you'll quickly realize that the principle is powerful, and can be extended to more tasks, and more

activities. Program Example 9.4 shows the program listing. The program creates two timers, named **timer_fast** and **timer_slow**. The main program starts these running, and tests when each exceeds a certain number. When the time value is exceeded, a function is called, which flips the associated led.

```
/*Program Example 9.4: Program which runs two time-based tasks
                                                                    */
#include "mbed.h"
Timer timer_fast;            // define Timer with name "timer_fast"
Timer timer_slow;            // define Timer with name "timer_slow"
DigitalOut ledA(LED1);
DigitalOut ledB(LED4);

void task_fast(void);        //function prototypes
void task_slow(void);

int main() {
  timer_fast.start();    //start the Timers
  timer_slow.start();
  while (1){
    if (timer_fast.read()>0.2){ //test Timer value
     task_fast();               //call the task if trigger time is reached
     timer_fast.reset();            //and reset the Timer
    }
    if (timer_slow.read()>1){     //test Timer value
     task_slow();
     timer_slow.reset();
    }
  }
}

void task_fast(void){          //"Fast" Task
  ledA = !ledA;
}

void task_slow(void){          //"Slow" Task
  ledB = !ledB;
}
```

Program Example 9.4: Running two timed tasks

Create a project around Program Example 9.4, and run it on the mbed alone. Check the timing with a stopwatch or oscilloscope.

Exercise 9.6

Experiment with different repetition rates in Program Example 9.4, including ones which aren't multiples of each other. Add a third and then fourth timer to it, flashing all mbed LEDs at different rates.

■

9.6.2 Testing the Timer Maximum Duration

We quoted above the maximum value that the timer can reach. As with many microcontroller features, it's of course very important to understand the operating limits, and to be certain that we remain within them. Program Example 9.5 measures the timer limit in a simple way. It clears the timer and then sets it running, displaying a time update to the CoolTerm or Tera Term screen every second. In doing this, it keeps its own record of seconds elapsed, and compares this with the elapsed time value given by the timer. From the program structure, we expect the timer value always to be just ahead of the recorded time value. At some point, however, the timer overflows back to zero, and this condition is no longer satisfied. The program detects this, and sends a message to the screen. The highest value reached by the timer is recorded on the screen.

```
/* Program Example 9.5: Tests Timer duration, displaying current time values to
terminal
                                                                                */
#include "mbed.h"

Timer t;
float s=0;                              //seconds cumulative count
float m=0;                              //minutes cumulative count
DigitalOut diag (LED1);
Serial pc(USBTX, USBRX);

int main() {
  pc.printf("\r\nTimer Duration Test\n\r");
  pc.printf("————————————————\n\n\r");
  t.reset();                                    //reset Timer
  t.start();                                   // start Timer
  while(1){
    if (t.read()>=(s+1)){ //has Timer passed next whole second?
      diag = 1;           //If yes, flash LED and print a message
      wait (0.05);
      diag = 0;
      s++ ;
      //print the number of seconds exceeding whole minutes
      pc.printf("%1.0f seconds\r\n",(s-60*(m-1)));
    }
    if (t.read()>=60*m){
      printf("%1.0f minutes \n\r",m);
      m++ ;
    }
    if (t.read()<s){     //test for overflow
      pc.printf("\r\nTimer has overflowed!\n\r");
      for(;;){}         //lock into an endless loop doing nothing
    }
  }            //end of while
}
```

Program Example 9.5: Testing timer duration

Exercise 9.7

Create a project around Program Example 9.5, and run it on an mbed, enabling also a connection to CoolTerm or Tera Term. No further hardware is needed. Calculate precisely the time duration you expect from the timer, i.e., $(2^{31}-1)$ microseconds. You can leave the program running while doing other things. Come back, however, just after half an hour, and watch intently to see if it overflows! Do your predicted and measured times agree?

■

C code feature

Insome embedded programs, there comes a moment when we just want the program to stop, there is simply nothing more for it to do. Program Example 9.3 is one such. Yet we have no instruction available to us which just says stop; the CPU is designed to go on and on running until it's switched off. Notice at the end of this program example how the stop is implemented, by trapping program execution in an endless loop which does nothing.

9.7 Using the mbed Timeout

Program Example 9.4 showed the mbed timer being used to trigger time-based events in an effective way. However, we needed to poll the timer value to know when the event should be triggered. The timeout allows an event to be triggered by an interrupt, with no polling needed. Timeout sets up an interrupt to call a function after a specified delay. There is no limit on the number of timeouts created. The API summary is shown in Table 9.4.

9.7.1 A Simple Timeout Application

A simple first example of Timeout is shown in Program Example 9.6. This causes an action to be triggered a fixed period after an external event. This simple program is made

Table 9.4: Application programming interface summary for timeout.

Function	Usage
`attach`	Attach a function to be called by the timeout, specifying the delay in seconds
`attach`	Attach a member function to be called by the timeout, specifying the delay in seconds
`attach_us`	Attach a function to be called by the timeout, specifying the delay in microseconds
`attach_us`	Attach a member function to be called by the timeout, specifying the delay in microseconds
`detach`	Detach the function

up of the **main( )** function and a **blink( )** function. A **Timeout** object is created, named **Response**, along with some familiar digital input and output. Looking in the **main( )** function, we see an **if** declaration, which tests if the button is pressed. If it is, the **blink( )** function gets attached to the **Response** Timeout. We can expect that two seconds after this attachment is made, the **blink( )** function will be called. To aid in our diagnostics, the button also switches on LED3. As a continuous task, the state of LED1 is reversed every 0.2 s. This program is thus a microcosm of many embedded systems programs. A time-triggered task needs to keep going, while an event-triggered task needs to take place at unpredictable times.

```
/*Program Example 9.6: Demonstrates Timeout, by triggering an event a fixed
duration after a button press.                                          */

#include "mbed.h"
Timeout Response;              //create a Timeout, and name it "Response"
DigitalIn button (p5);
DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);

void blink() {                 //this function is called at the end of the Timeout
  led2 = 1;
  wait(0.5);
  led2=0;
}

int main() {
  while(1) {
    if(button==1){
      Response.attach(&blink,2.0); // attach blink function to Response Timeout,
                                            //to occur after 2 seconds
      led3=1;                     //shows button has been pressed
    }
    else {
      led3=0;
    }
    led1=!led1;
    wait(0.2);
  }
}
```

Program Example 9.6: Simple timeout application

Compile Program Example 9.6, and download to an mbed, with the build of Fig. 9.3. Observe the response as the push-button is pressed.

Exercise 9.8

With Program Example 9.6 running, answer the following questions:

1. Is the 2-s timeout timed from when the button is pressed, or when it is released? Why is this?
2. When the event-triggered task occurs (i.e., the blinking of LED2), what impact does it have on the time-triggered task (i.e., the flashing of LED1)?
3. If you tap the button very quickly, you will see that it is possible for the program to miss it entirely (even though electrically we can prove that the button has been pressed). Why is this?

■

9.7.2 Further Use of Timeout

Program Example 9.6 has demonstrated use of the timeout nicely, but the questions in Exercise 9.8 throw up some of the classic problems of task timing, notably that execution of the event-triggered task can interfere with the timing of the time-triggered task.

Program Example 9.7 does the same thing as the previous example, but in a much better way. Glancing through it, we see two timeouts created, and an interrupt. The latter connects to the push-button, and replaces the digital input which previously took that role. There are now three functions, in addition to **main( )**. This has become extremely short and simple, and is concerned primarily with keeping the time-triggered task going. Now response to the push-button is by interrupt, and it is within the interrupt function that the first timeout is set up. When it is spent, the **blink( )** function is called. This sets the LED output, but then enables the second timeout, which will trigger the end of the LED blink.

```
/*Program Example 9.7: Demonstrates the use of Timeout and interrupts, to allow
response to an event-driven task while a time-driven task continues.
                                                                              */
#include "mbed.h"
void blink_end (void);
void blink (void);
void ISR1 (void);
DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);
Timeout Response;             //create a Timeout, and name it Response
Timeout Response_duration;    //create a Timeout, and name it Response_duration
InterruptIn button(p5);      //create an interrupt input, named button
```

```
void blink() {        //This function is called when Timeout is complete
  led2=1;
  // set the duration of the led blink, with another timeout, duration 0.1 s
  Response_duration.attach(&blink_end, 1);
}

void blink_end() {       //A function called at the end of Timeout
Response_duration
  led2=0;
}

void ISR1(){
  led3=1;   //shows button is pressed; diagnostic and not central to program
  //attach blink1 function to Response Timeout, to occur after 2 seconds
  Response.attach(&blink, 2.0);
}

int main() {
  button.rise(&ISR1);   //attach the address of ISR1 function to the rising edge
  while(1) {
    led3=0;                //clear LED3
    led1=!led1;
    wait(0.2);
  }
}
```

Program Example 9.7: Improved use of timeout

Compile and run the code of Program Example 9.7, again using the build of Fig. 9.3.

Exercise 9.9

Repeat all the questions of Exercise 9.8 for the most recent program example, noting and explaining the differences you find.

■

9.7.3 Timeout Used to Test Reaction Time

C code feature

Program Example 9.8 shows an interesting recreational application of Timeout, in which the timeout duration is itself a variable. It tests reaction time by blinking an LED, and timing how long it takes for the player to hit a switch in response. To add challenge, a "random" delay is generated before the LED is lit. This uses the C library function **rand( )**. The program should be understandable from the comments it contains.

The circuit build is the same as seen in Fig. 9.3. Now the push-button is the switch the player must hit to show a reaction. A Tera Term terminal should be enabled.

```
/*Program Example 9.8: Tests reaction time, and demos use of Timer and Timeout
functions
                                                                              */
#include"mbed.h"
#include <stdio.h>
#include <stdlib.h>            //contains rand() function
void measure ();
Serial pc(USBTX, USBRX);
DigitalOut led1(LED1);
DigitalOut led4(LED4);
DigitalIn responseinput(p5);  //the player hits the switch connected here to
respond
Timer t;                                    //used to measure the response time
Timeout action;               //the Timeout used to initiate the response speed
test

int main (){
  pc.printf("Reaction Time Test\n\r");
  pc.printf("———————————————————\n\r");
  while (1) {
    int r_delay;     //this will be the "random" delay before the led is blinked
    pc.printf("New Test\n\r");
    led4=1;                      //warn that test will start
    wait(0.2);
    led4=0;
    r_delay = rand() % 10 + 1;   // generates a pseudorandom number range 1-10
    pc.printf("random number is %i\n\r", r_delay);  // allows test randomness;
                                                   //removed for normal play
    action.attach(&measure,r_delay); // set up Timeout to call measure()
                                                        // after random time
    wait(10);      //test will start within this time, and we then return to it
   }
}

void measure (){      // called when the led blinks, and measures response time
  if (responseinput ==1){                          //detect cheating!
    pc.printf("Don't hold button down!");
  }
  else{
    t.start();             //start the timer
    led1=1;                //blink the led
    wait(0.05);
    led1=0;
    while (responseinput==0) {
     //wait here for response
    }
    t.stop();                       //stop the timer once response detected
    pc.printf("Your reaction time was %f seconds\n\r", t.read());
    t.reset();
  }
}
```

Program Example 9.8: Reaction time test: applying timer and timeout

Exercise 9.10

Run Program Example 9.8 for a period of time, and note the sequence of "random" numbers. Run it again. Do you recognize a pattern in the sequence? In fact, it is difficult for a computer to generate true random numbers, though a number of tricks and algorithms are used to create *pseudorandom* sequences of numbers.

■

9.8 Using the mbed Ticker

The mbed ticker feature sets up a recurring interrupt, which can be used to call a function periodically, at a rate decided by the programmer. There is no limit on the number of tickers created. The API summary is shown in Table 9.5.

We can demonstrate ticker by returning to our very first program example, number 2.1. This simply flashes an LED every 200 ms. Creating a periodic event is one of the most natural and common requirements in an embedded system, so it's not surprising that it appeared in our first program. We created the 200-ms period by using a delay function; we now recognize that these are useful only in a limited way, as when they're running they tie up the CPU, so that it can do nothing else productive.

Program Example 9.9 simply replaces the delay functions with the ticker.

```
/* Program Example 9.9: Simple demo of "Ticker". Replicates behaviour of first
led flashing program.
                                                                              */
#include"mbed.h"
void led_switch(void);
Ticker time_up;             //define a Ticker, with name "time_up"
DigitalOut myled(LED1);
```

Table 9.5: Application programming interface summary for ticker.

Function	Usage
`attach`	Attach a function to be called by the ticker, specifying the interval in seconds
`attach`	Attach a member function to be called by the ticker, specifying the interval in seconds
`attach_us`	Attach a function to be called by the ticker, specifying the interval in microseconds
`attach_us`	Attach a member function to be called by the ticker, specifying the interval in microseconds
`detach`	Detach the function

```
void led_switch(){          //the function that Ticker will call
    myled=!myled;
}

int main(){
    time_up.attach(&led_switch, 0.2);            //initialises the ticker
    while(1){       //sit in a loop doing nothing, waiting for Ticker interrupt
    }
}
```

Program Example 9.9: Applying ticker to our very first program

It should be easy to follow what is going on in this program. The major step forward is that the CPU is now freed to do anything that's needed, while the task of measuring the time between LED changes is handed over to the timer hardware, running in the background.

We've already called functions periodically with the timer feature, so at first, ticker doesn't seem to add anything really new. Remember, however, that we have to poll the timer value in Program Example 9.4 to test its value, and instigate the related function. Using ticker, it is an interrupt that calls the associated function when time is up. As already discussed, this is a more effective approach to programming.

9.8.1 Using Ticker for a Metronome

Program Example 9.10 uses the mbed to create a metronome, using the ticker facility. If you've not met one before, a metronome is an aid to musicians, setting a steady beat, against which they can play their music. The musician generally selects a beat rate, in a range usually between 40 to 208 beats per second. Old metronomes were based on elegant clockwork mechanisms, with a swinging pendulum arm. Most these days are electronic. Normally the indication given to the musician is a loud audible "tick" sometimes these days accompanied by an LED flash. Here, we just restrict ourselves to the LED.

The main program **while** loop checks the up and down buttons, adjusts the beat rate accordingly, and displays the current rate to the host terminal screen. This loops continuously, but lurking in the background is the ticker. This has been initialized before the **while** loop, in the line

```
beat_rate.attach(&beat, period);  //initialises the beat rate
```

Once the time indicated by **period** has elapsed, the **beat** function is called. At this moment, the ticker is updated, possibly with a new value of **period**, and the LED is

flashed to indicate a beat. Program execution then returns to the main **while** loop, until the next ticker occurrence.

```
/*Program Example 9.10: Metronome. Uses Ticker to set beat rate
                                                                        */
#include "mbed.h"
#include <stdio.h>
Serial pc(USBTX, USBRX);
DigitalIn up_button(p5);
DigitalIn down_button(p6);
DigitalOut redled(p19);       //displays the metronome beat
Ticker beat_rate;             //define a Ticker, with name "beat_rate"
void beat(void);
float period (0.5);           //metronome period in seconds, inital value 0.5
int rate (120);               //metronome rate, initial value 120

int main() {
  pc.printf("\r\n");
  pc.printf("mbed metronome!\r\n");
  pc.printf("________________\r\n");
  period = 1;
  redled = 1;        //diagnostic
  wait(.1);
  redled = 0;
  beat_rate.attach(&beat, period);  //initialises the beat rate
  //main loop checks buttons, updates rates and displays
  while(1){
    if (up_button ==1)     //increase rate by 4
      rate = rate + 4;
    if (down_button ==1)   //decrease rate by 4
      rate = rate - 4;
    if (rate > 208)        //limit the maximum beat rate to 208
      rate = 208;
    if (rate < 40)                      //limit the minimum beat rate to 40
      rate = 40;
    period = 60/rate;      //calculate the beat period
    pc.printf("metronome rate is %i\r", rate);
    //pc.printf("metronome period is %f\r\n", period);        //optional check
    wait (0.5);
  }
}

  void beat() {                        //this is the metronome beat
    beat_rate.attach(&beat, period);    //update beat rate at this moment
    redled = 1;
    wait(.1);
    redled = 0;
  }
```

Program Example 9.10: Metronome, applying ticker

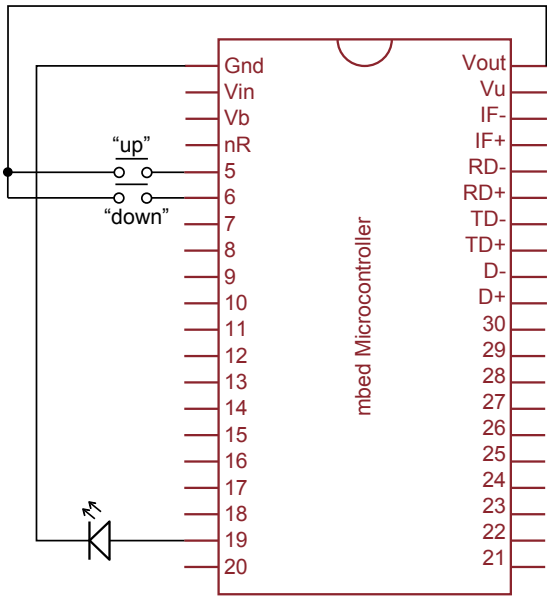

Figure 9.7
Metronome build.

The breadboard build for the metronome is simple, and is shown in Fig. 9.7. CoolTerm or Tera Term should be enabled. Build the hardware, and compile and download the program. With a stopwatch or other timepiece, check that the beat rates are accurate. If you are using the app board, you can configure the Joystick up and down switch positions and an onboard LED instead of the connections shown in the Figure.

Exercise 9.11

We've left a bit of polling in Program Example 9.10. Rewrite it so that response to the external pins is done with interrupts.

■

9.8.2 Reflecting on Multitasking in the Metronome Program

Having picked up the idea of program tasks at the beginning of this chapter, we've seen a sequence of program examples which could be described as multitasking. This started with Program Example 9.1, which runs one time-triggered and one event-triggered task. Many

of the subsequent programs have clear multitasking features. In the metronome example, the program has to keep a regular beat going. While doing this, it must also respond to inputs from the user, calculate beat rates, and write to the display. There is therefore one time-triggered task, the beat, and at least one event-triggered task, the user input. Sometimes, it's difficult to decide what program activities belong together in one task. In the metronome example, the user response seems to contain several activities, but they all relate closely to each other, so it is sensible to view them as the same task.

We have developed here a useful program structure, whereby time-triggered tasks can be linked to a timer or ticker function, and event-triggered tasks to an external interrupt. This will be useful as long as there are not too many tasks, and as long as they are not too demanding of CPU time.

9.9 The Real-Time Clock

The RTC is an ultra-low-power peripheral on the LPC1768, which is implemented by the mbed. The RTC is a timing/counting system which maintains a calendar and time-of-day clock, with registers for seconds, minutes, hours, day, month, year, day of month, and day of year. It can also generate an alarm for a specific date and time. It runs from its own 32-kHz crystal oscillator, and can have its own independent battery power supply. It can thus be powered, and continue in operation, even if the rest of the microcontroller is powered down. The mbed API doesn't create any C++ objects, but just implements standard functions from the standard C library, as shown in Table 9.6. Simple examples for use can be found on the mbed website [1].

9.10 Switch Debouncing

With the introduction of interrupts, we now have some choices to make when writing a program to a particular design specification. For example, Program Example 3.3 uses a

Table 9.6: Application programming interface summary for real-time clock.

Function	Usage
`time`	Get the current time
`set_time`	Set the current time
`mktime`	Converts a tm structure (a format for a time record) to a timestamp
`localtime`	Converts a timestamp to a tm structure
`ctime`	Converts a timestamp to a human-readable string
`strftime`	Converts a tm structure to a custom format human-readable string

digital input to determine which of two LEDs to flash. The digital input value is continuously polled within an infinite loop. However, we could equally have designed this program with an event-driven approach, to flip a control variable every time the digital input changes. Importantly, there are some inherent timing constraints within Program Example 3.3 which have not previously been discussed. One is that the frequency of polling is actually quite slow, because once the switch input has been tested, a 0.4-s flash sequence is activated. This means that the system has a response time of at worst 0.4 s, because it only tests the switch input once for every program loop. When the switch changes position, it could take up to 0.4 s for the LED to change, which is very slow in terms of embedded systems.

With interrupt-driven systems, we can have much quicker response rates to switch presses, because response to the digital input can take place while other tasks are running. However, when a system can respond very rapidly to a switch change, we see a new issue which needs addressing, called *switch bounce*. This is due to the fact that the mechanical contacts of a switch do literally bounce together, as the switch closes. This can cause a digital input to swing wildly between Logic 0 and Logic 1 for a short time after a switch closes, as illustrated in Fig. 9.8. The solution to switch bounce is a technique called *switch debouncing*.

First, we can identify the problem with switch bounce by evaluating a simple event-driven program. Program Example 9.11 attaches a function to a digital interrupt on pin 5, so the circuit build shown in Fig. 9.3 can again be used. The function simply toggles (flips) the state of the mbed's onboard LED1 for every raising edge on pin 18.

```
/* Program Example 9.11: Toggles LED1 every time p18 goes high. Uses hardware build
shown in Figure 9.3.
                                                                                */
#include"mbed.h"
InterruptIn button(p18);    // Interrupt on digital pushbutton input p18
DigitalOut led1(LED1);      // mbed LED1
void toggle(void);          // function prototype
```

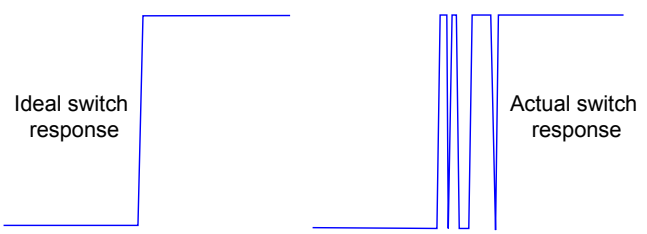

Figure 9.8
Demonstrating switch bounce.

```
int main() {
  button.rise(&toggle);            // attach the address of the toggle
}                                  // function to the rising edge

void toggle() {
  led1=!led1;
}
```

Program Example 9.11: Toggles LED1 every time mbed pin 5 goes high

Implement program 9.11 with a push-button or SPDT-type switch connected between pin 18 and pin 40. Depending a little on the type of switch you use, you will see that actually the program doesn't work very well. It can become unresponsive or the button presses can become out of synch with the LED. This demonstrates the problem with switch bounce.

From Fig. 9.8, it is easy to see how a single button press or change of switch position can cause multiple interrupts and hence the LED can get out of synch with the button. We can "debounce" the switch with a timer feature. The debounce feature needs to ensure that once the raising edge has been seen, no further raising edge interrupts should be implemented until a calibrated time period has elapsed. In reality, some switches move positions cleaner than others, so the exact timing required needs some tuning. To assist, switch manufacturers often provide data on switch bounce duration. The downside to including debouncing of this type is that the implemented timing period also reduces the response of the switch, although not as much as that discussed with reference to polling.

There are a number of ways to implement switch debouncing. In hardware, there are little configurations of logic gates which can be used (see Ref. [1] of Chapter 5). In software, we can use timers, bespoke counters or other programming methods. Program Example 9.12 solves the switch bounce issue by starting a timer on a switch event, and ensuring that 10 ms has elapsed before allowing a second event to be processed.

```
/* Program Example 9.12: Event driven LED switching with switch debounce
                                                                        */
#include"mbed.h"
InterruptIn button(p18);    // Interrupt on digital pushbutton input p18
DigitalOut led1(LED1);           // digital out to LED1
Timer debounce;                  // define debounce timer
void toggle(void);               // function prototype
int main() {
  debounce.start();
  button.rise(&toggle);          // attach the address of the toggle
}                                          // function to the rising edge
void toggle() {
if (debounce.read_ms()>10)      // only allow toggle if debounce timer
  led1=!led1;                                       // has passed 10 ms
  debounce.reset();              // restart timer when the toggle is performed
}
```

Program Example 9.12: Event-driven LED switching with switch debounce

Exercise 9.12

1. Experiment with modifying the debounce time to be shorter or greater values. There comes a point where the timer doesn't effectively solve the debouncing, and at the other end of the scale responsiveness can be reduced too. What is the best debounce time for the switch you are using?
2. Implement Program Example 9.12 using an mbed timeout object instead of the timer object. Are there any advantages or disadvantages between each approach?
3. Rewrite Program Example 3.3 to have the same functionality but with an event-driven approach (i.e., using interrupts). How much improvement can be made to the system's responsiveness to switch changes?

■

9.11 Where Do We Go From Here? The Real-Time Operating System

9.11.1 The Limits of Conventional Programming

Programs in this book have so far been almost all structured as a main loop (sometimes called a *super loop*); in most case, this itself includes further loops and conditional statements embedded within. Our programs can also now include interrupts. The resulting structure is symbolized in simple form in Fig. 9.9. This structure is adequate for many programs, but there comes point when the structure is no longer adequate; the loop might become just too big, or some of the tasks are only intermittent, or the tasks or ISRs cause unacceptable delay to each other.

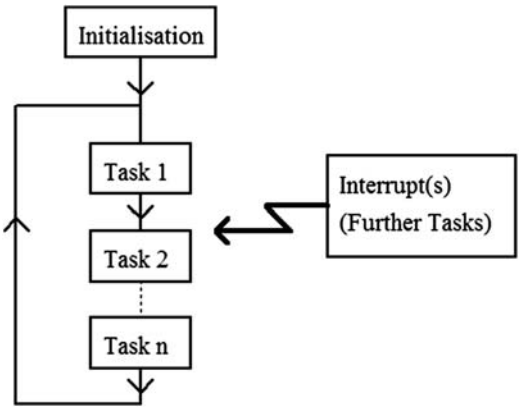

Figure 9.9
A conventional program structure.

9.11.2 Introducing the Real-Time Operating System

The *real-time operating system*, or RTOS (pronounced "Arr—Toss"), provides a completely different approach to program development, and takes us far from the assumptions of normal sequential programming. With the RTOS, we hand over control of the CPU and all system resources to the operating system. It is the operating system which now determines which section of the program is to run for how long, and how it accesses system resources. The application program itself is subservient to the operating system, and is written in a way that recognizes the requirements of the operating system.

A program written for an RTOS is structured into tasks or *threads*. While there are subtle differences between the use of the words task and thread, let's use them interchangeably in this simple introduction. Each task is written as a self-contained program module, a bit like a function. The tasks can be prioritized, though this is not always the case. The RTOS performs three main functions:

- it decides which task/thread should run and for how long;
- it provides communication and synchronization between tasks; and
- it controls the use of resources shared between the tasks, for example, memory and hardware peripherals.

An important part of the RTOS is its *scheduler*, which decides which task runs, and for how long. A simple scheduler, which demonstrates some important concepts, is called the *round robin scheduler*. This is illustrated in Fig. 9.10, where three tasks are running in turn. The scheduler synchronizes its activity to a *clock tick*, a periodic interrupt from an internal timer, like the mbed ticker described in Section 9.8. At every clock tick, the scheduler determines if a different task should be given CPU time. In round robin scheduling, the task is always switched. That means that whatever task is executing must suspend its activity mid-flow, and wait for its turn again. Essential data, like program counter value and register values (called the context, as we saw in Fig. 9.2) are saved in memory for when the task runs again. Meanwhile, the context of the task about to run is

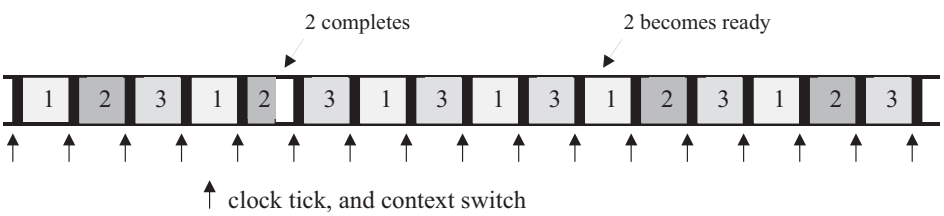

Figure 9.10
Round robin scheduling.

retrieved from memory. This *context switch* takes a bit of time and requires some memory; these are costs incurred by the use of the RTOS.

Round robin scheduling introduces some basic ideas, but doesn't allow task prioritization. There are other forms of scheduling which do allow this (for example, *prioritized preemptive scheduling*), or which reduce the time spent in the context switch (for example, *cooperative scheduling*). Other features of the RTOS allow tasks to be synchronized or to pass data between each other.

9.11.3 A Mention of the mbed RTOS

There is an "official" RTOS available for the mbed, detailed in Ref. [2]. This demonstrates the key features described above and provides an important programming technique for the more advanced programmer or system designer. Rightly or wrongly, we decided not to cover its use in this book. However, we do encourage you to consider it as a useful and natural step forward, as your programming skills develop. An excellent introduction to this RTOS is given in Chapter 6 of Ref. [3]. Programming with an RTOS is an elegant and satisfying activity; use of the RTOS encourages well-structured programs.

9.12 Mini Projects

9.12.1 A Self-contained Metronome

The metronome described in Section 9.8.1 is interesting, but it doesn't result in something that a musician would really want to use. So try revising the program, and its associated build, to make a self-contained battery-powered unit, using an LCD display instead of the host computer screen to display beat rate. Experiment also with getting a loudspeaker to "tick" along with the LED. If you succeed in this, then try including the facility to play "concert A" (440 Hz), or another pitch, to allow the musicians to tune their instruments. This project will work on either a breadboard build or the application board. It is particularly attractive to do it on the latter, with its built-in speaker, LCD and joystick.

9.12.2 Accelerometer Threshold Interrupt

We met the ADXL345 accelerometer, with its SPI serial interface, in Section 7.3. Although we didn't use them at the time, it is interesting to note that the device has two interrupt outputs, as seen in Fig. 7.6. These can be connected to an mbed digital input to run an interrupt routine whenever an acceleration threshold is exceeded. For example, the accelerometer might be a crash detection sensor in a vehicle which, when a specified acceleration value is exceeded, activates an airbag.

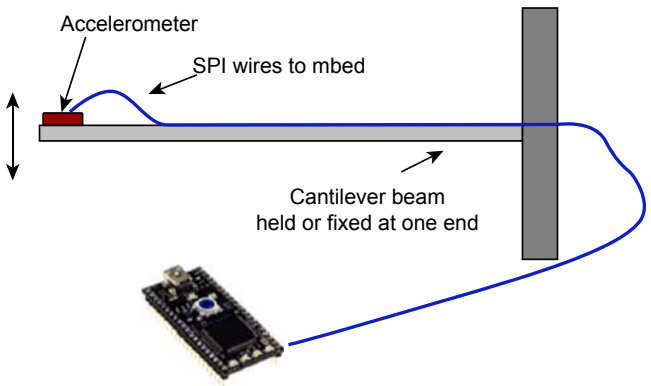

Figure 9.11
The accelerometer setup.

Use an accelerometer on a cantilever arm to provide the acceleration data. Fig. 9.11 shows the general construction. A plastic 30-cm or 1-foot ruler, clamped at one end to a table, can be used. Set the accelerometer to generate an interrupt whenever a threshold in the z-axis is exceeded. Connect this as an interrupt input to the mbed, and program it so a warning LED lights for one second whenever the threshold is exceeded. You can experiment with the actual threshold value to alter the sensitivity of the detection system.

Chapter Review

- Signal inputs can be repeatedly tested in a loop, a process known as polling.
- An interrupt allows an external signal to interrupt the action of the CPU, and start code execution from somewhere else in the program.
- Interrupts are a powerful addition to the structure of the microprocessor. Generally, multiple interrupt inputs are possible, which adds considerably to the complexity of both hardware and software.
- It is easy to make a digital counter circuit, which counts the number of logic pulses presented at its input. Such a counter can be readily integrated into a microcontroller structure.
- Given a clock signal of known and reliable frequency, a counter can readily be used as a timer.
- Timers can be structured in different ways so that interrupts can be generated from their output, for example, to give a continuous sequence of interrupt pulses.
- Switch debounce is required in many cases to avoid multiple responses being triggered by a single switch press.

Quiz

1. Explain the differences between using polling and event-driven techniques for testing the state of one of the digital input pins on a microcontroller.
2. List the most significant actions that a CPU takes when it responds to an enabled interrupt.
3. Explain the following terms with respect to interrupts:
   a. Priority
   b. Latency
   c. Nesting
4. A comparator circuit and LM35 are to be used to create an interrupt source, using the circuit of Fig. 9.5. The comparator is supplied from 5.0 V, and the temperature threshold is to be approximately 38°C. Suggest values for R_1 and R_2. Resistor values of 470, 680, 820, 1k, 1k2, 1k5, and 10k are available.
5. Describe in overview how a timer circuit can be implemented in hardware as part of a microprocessor's architecture.
6. What is the maximum value, in decimal, that a 12-bit and a 24-bit counter can count up to?
7. A 4.0-MHz clock signal is connected to the inputs of a 12-bit and a 16-bit counter. Each starts counting from zero. How long does it take before each it reaches its maximum value?
8. A 10-bit counter, clocked with an input frequency of 512 kHz, runs continuously. Every time it overflows, it generates an interrupt. What is the frequency of that interrupt stream?
9. What is the purpose of the mbed's Real Time Clock? Give an example of when it might be used.
10. Describe the issue of switch bounce and explain how timers can be used to overcome this. What is the disadvantage of this approach?

References

[1] Mbed handbook Time page. https://developer.mbed.org/handbook/Time.
[2] The mbed RTOS home page. https://developer.mbed.org/handbook/RTOS.
[3] Trevor Martin, The Designer's Guide to the Cortex-M Processor Family, Elsevier, 2013.