Chapter 9: Interrupts, Timers and Tasks

www.embedded-knowhow.co.uk

Timers and Interrupts

The very first diagram in this book, repeated below, 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:

- Measure time durations;

- Generate time-based events, which may be single or repetitive;

- Respond with appropriate speed to external events, which may occur at unpredictable times.

(User Interface)

TIME

Embedded Computer

Input Variables

Software

Hardware

Output Variables

(Link to other Systems)
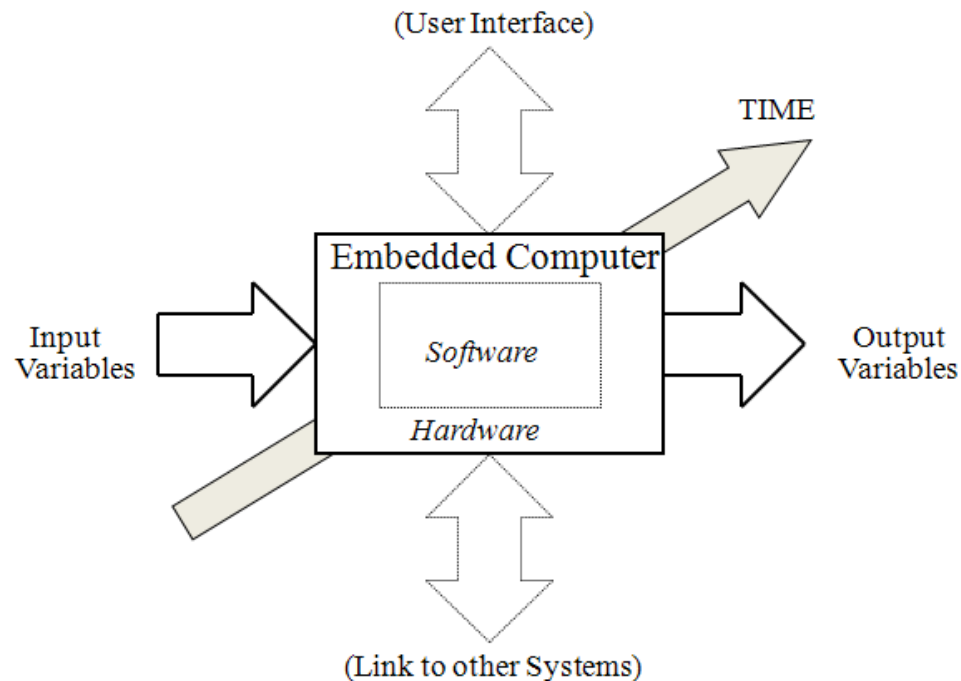
Tasks: Event-Triggered and Time-Triggered

In almost all embedded programs, the program has to undertake a number of different activities. We call these distinct activities *tasks.*

Once a program has more than one task, we enter the domain of *multi-tasking.*

Tasks performed by embedded systems tend to fall into two categories:

* event-triggered; occur when a particular external event happens, at a time which is not predictable;

* time-triggered happen periodically, at a time determined by the microcontroller.

As a simple example, a room temperature controller may have the tasks shown below.

| 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

Polling

One way of programming for an event-triggered activity, like a button push, is to continuously test that external input. This is illustrated below, where a program is structured as a continuous loop. This way of checking external events is called *polling*.

There are two main problems with polling:
- The processor can't perform any other operations during a polling routine
- All inputs are treated as equal; the urgent change has to wait its turn before it's recognised by the computer.
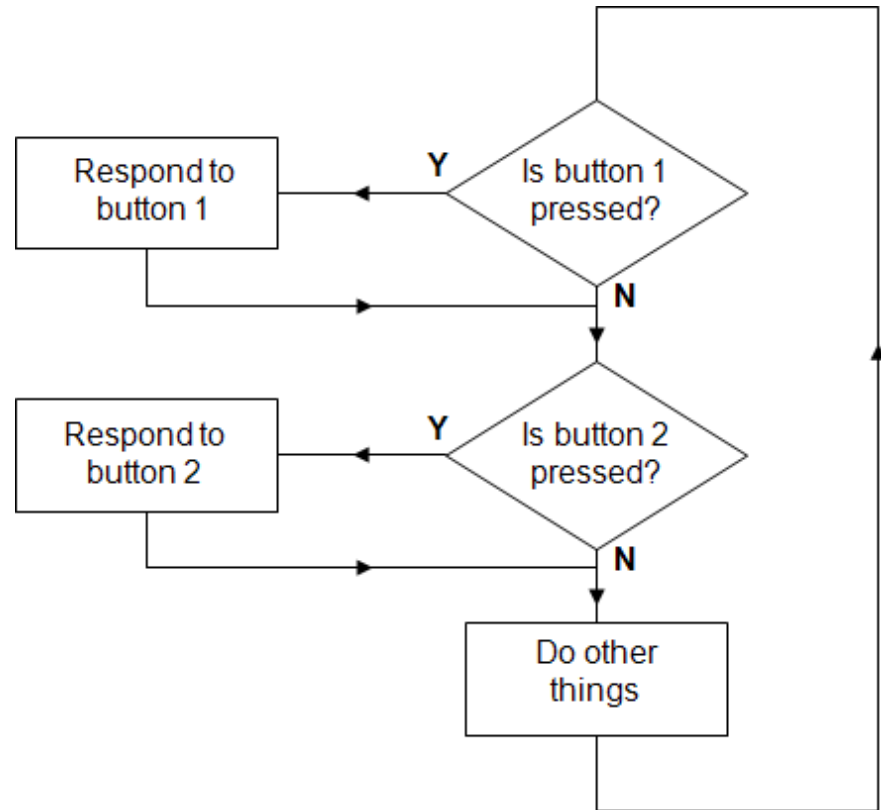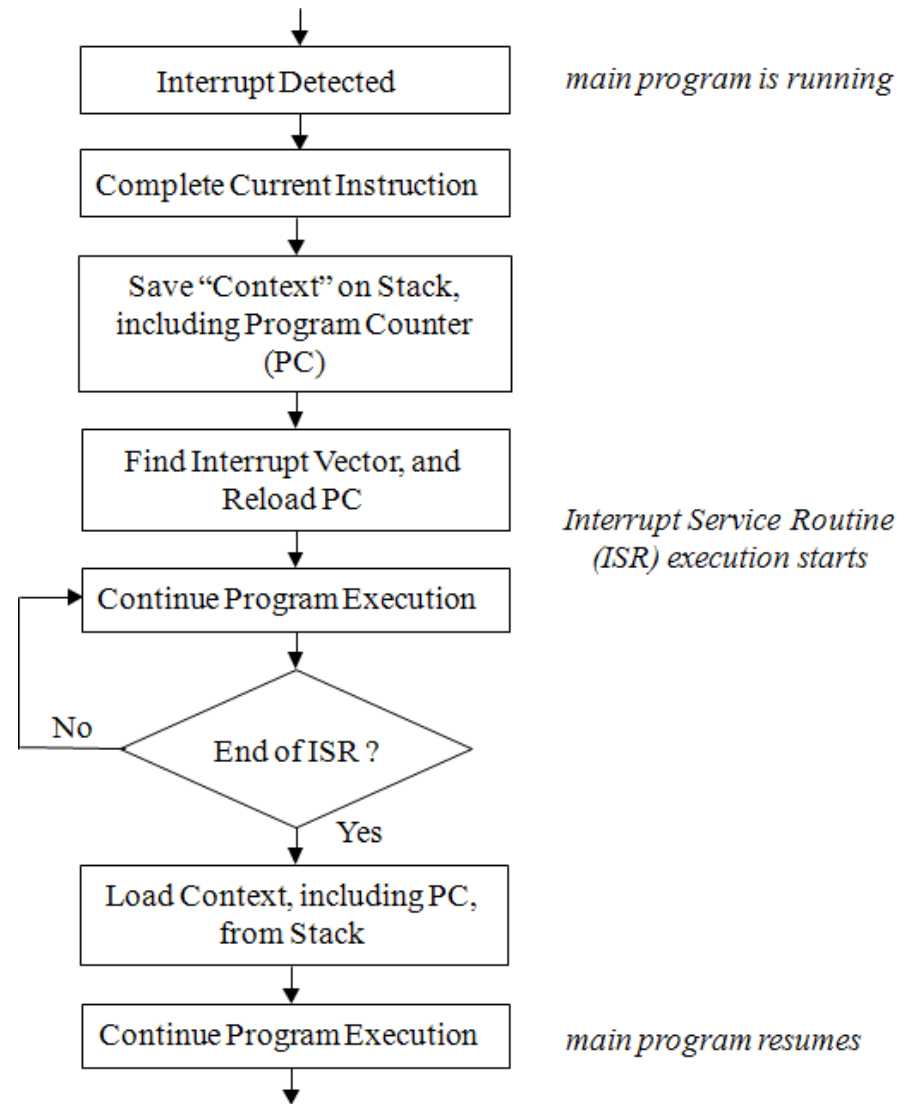
Introducing Interrupts

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.

When responding to interrupts, microcontrollers follow the general pattern shown.

Simple Interrupts on the mbed

The mbed API exploits only a small subset of the interrupt capability of the LPC1768 microcontroller. Any of pins 5 to 30 can be used as an interrupt input, except pins 19 and 20.

| 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
fall	Attach a function to call when a falling edge occurs on the input
mode	Set the input pin mode

Introductory Use of an mbed Interrupt

When the interrupt is activated, 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. The circuit is simple, as shown.

```
/* 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 the
led = !led;                          //interrupt, i.e. the ISR
}


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);
  }
}
```
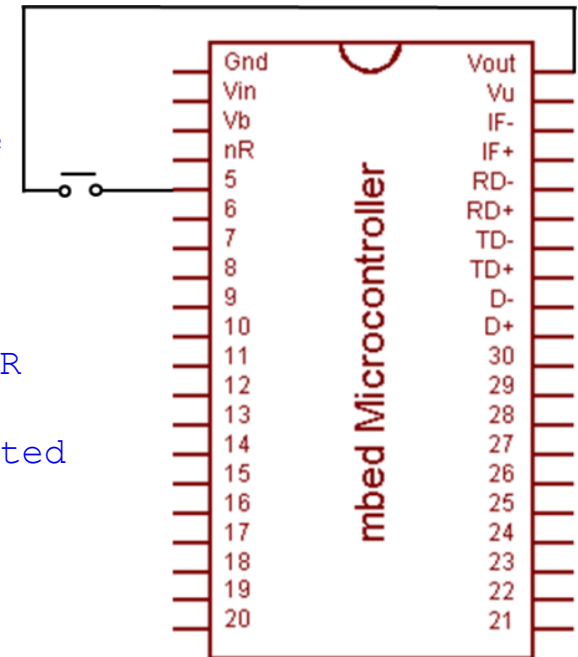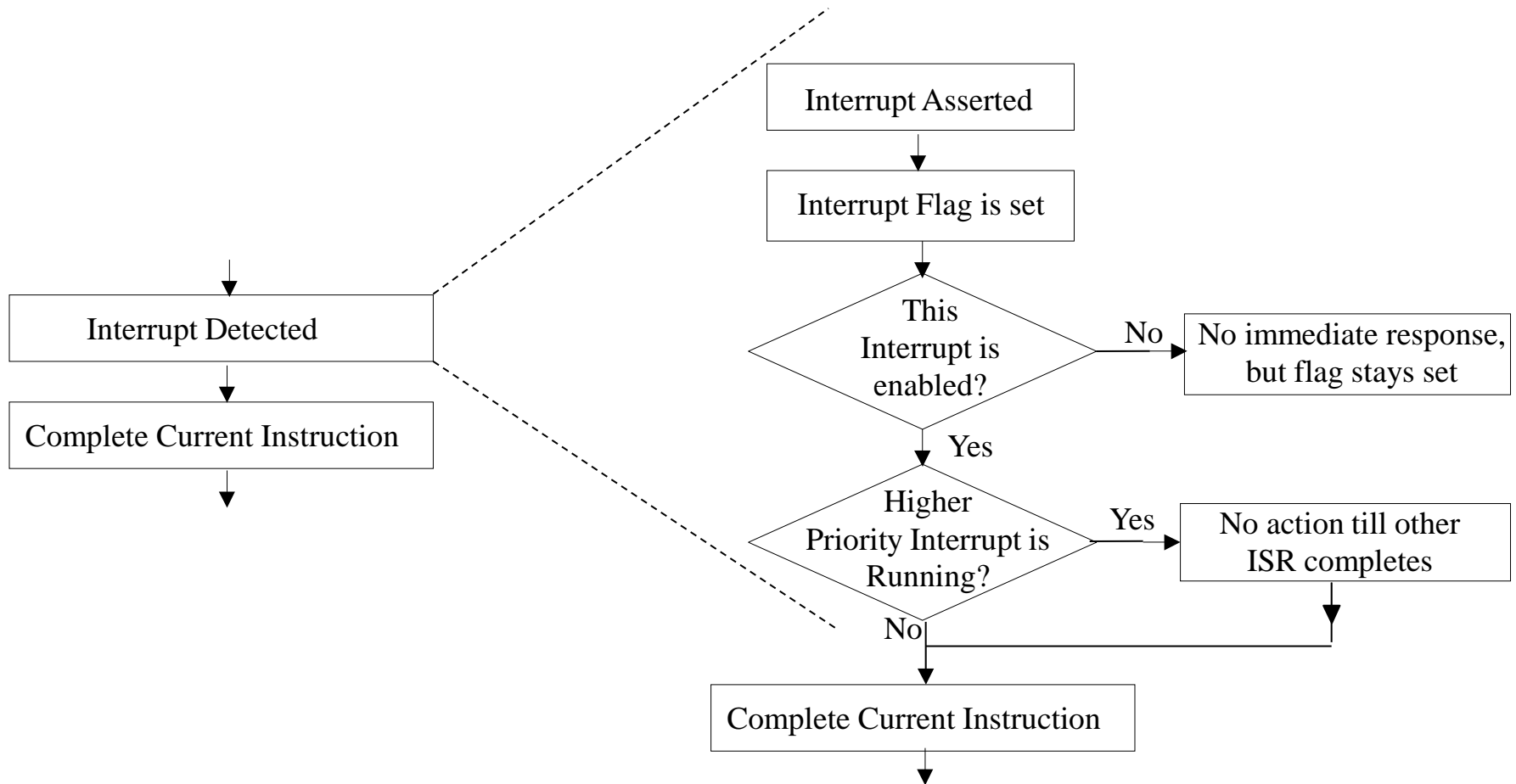
Getting Deeper into Interrupts

To deal with more complex interrupt applications, most processors contain four important mechanisms:

- Interrupts can be *prioritised,* - some can be 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. Working with nested interrupts increases the demands on the programmer, and is strictly for advanced players only.

- The location of the ISR in memory can be selected, to suit the memory map and programmer wishes.

*Also - t*he delay between the interrupt occurring, and the processor responding, is called the interrupt *latency.*

While the interrupt is waiting for a response from the processor, it is said to be *pending.*

A Typical Microprocessor Interrupt Response –
some greater detail

Interrupt Detected

↓

Complete Current Instruction

Interrupt Asserted

↓

Interrupt Flag is set

↓

This Interrupt is enabled? —No→ **No immediate response, but flag stays set**

↓ Yes

Higher Priority Interrupt is Running? —Yes→ **No action till other ISR completes**

↓ No

Complete Current Instruction

Testing Interrupt Latency

Latency can be observed with this program by observing on an oscilloscope a square wave input, and on the other beam the interrupt response, i.e. the output at "led". The latency is the delay between the two.

```
/* 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);
  }
}
```

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, as shown.
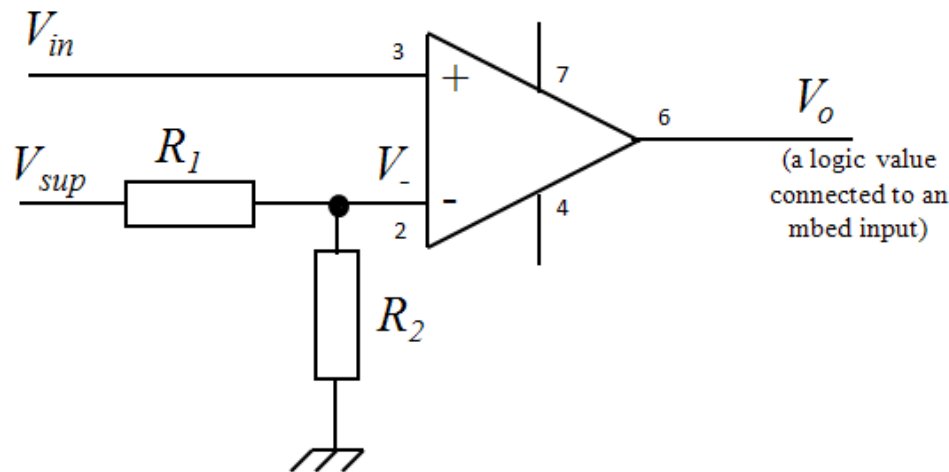
Question from the Quiz

4. A comparator circuit and LM35 are to be used to create an interrupt source, using the circuit of Figure 9.5 (as above). 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.

The Digital Counter

It is easy in digital electronics to make electronic counters by connecting together a series of bistables or flip-flops.

If the input is connected to a clock signal then the counter will count, in binary, the number of clock pulses applied to it.

The digital number held in the counter can be read, and it is possible to arrange the necessary logic to preload it with a certain number, or to clear it to zero.

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.

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.

We can preload

Input

| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |

an interrupt at the moment of overflow

We can read
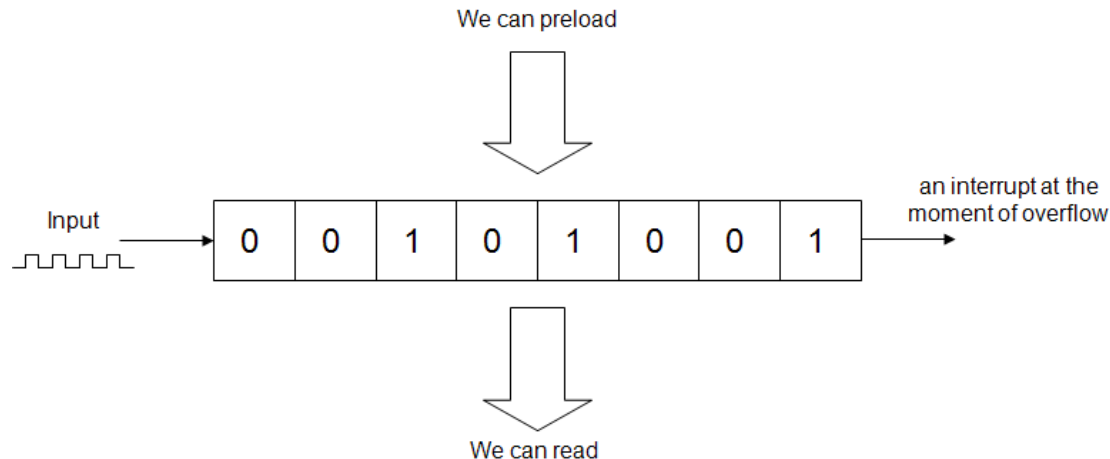
Counting and Timing

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.

If the clock source is a known and stable frequency, then the counter becomes a timer.

For example, if the clock frequency is 1.000 MHz (hence with period of 1 us, as shown by T_C below), 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 microsecond. This can be used to measure time, or trigger an event when a certain time has elapsed.

In general, if T_C is the clock period, and n cycles are counted, then the period during which counting has taken place is nT_C.
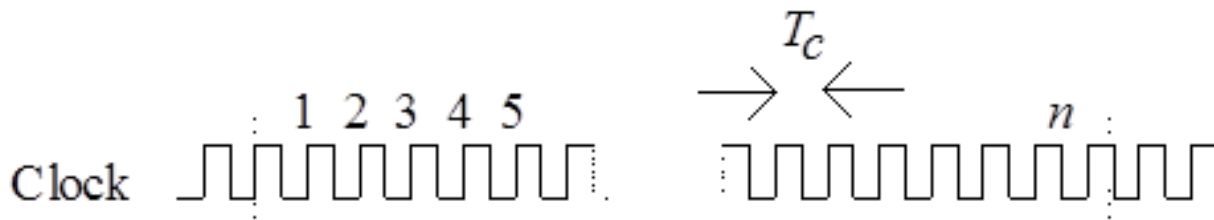
Counting and Timing – Interrupt on Overflow

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. The effective range of the counter has been extended.

If the counter is just free-running with a continuous clock signal, then the "interrupt on overflow" occurs repeatedly, as shown below. 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 us (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 synchronise timed activity.
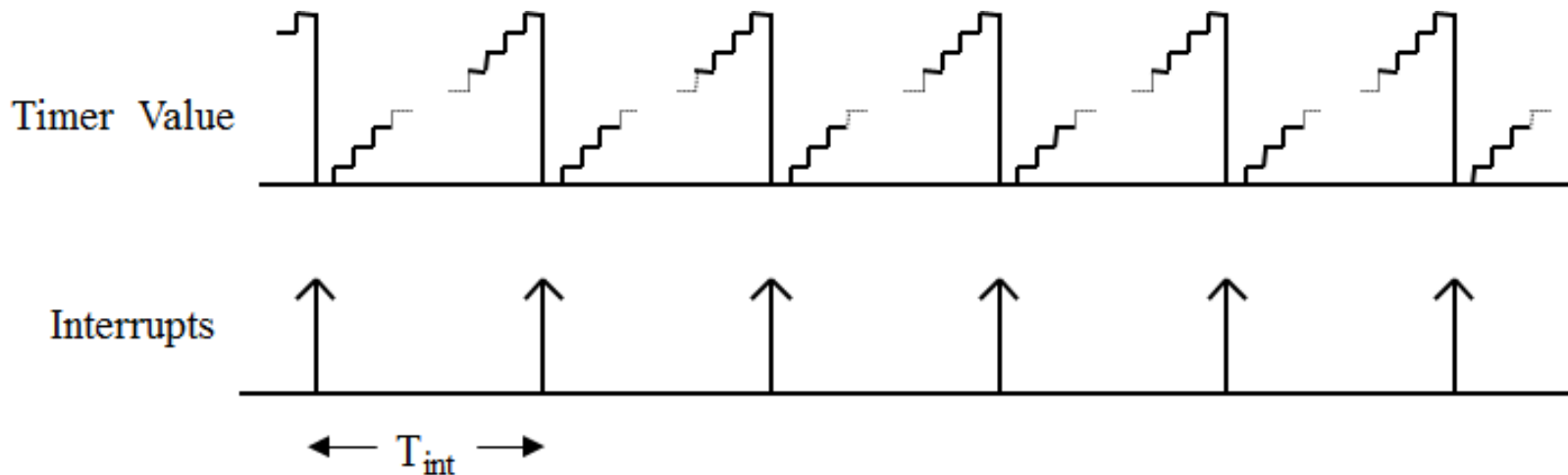
Questions from the Quiz

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?

Using the mbed Timer

The LPC1768 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, the *Timer*, used for simple timing applications, *Timeout*, which calls a function after a pre-determined delay, and *Ticker*, which repeatedly calls a function, at a pre-determined rate.

The mbed also applies a *Real Time Clock* to keep track of time of day, and date.

The mbed Timer allows basic timing activities to take place, for comparatively short time durations. A Timer can be created, started, stopped and read, as shown.

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

A simple Timer Application

This Program Example measures the time taken to write a message to the screen, and displays that message on Tera Term or CoolTerm.

```
/* Program Example 9.3: A simple Timer example, from mbed web
site. Activate Tera Term 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
//print to pc
  pc.printf("The time taken was %f seconds\n", t.read());
}
```

Using Multiple mbed Timers

```
/*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;
}
```

This program creates two Timers, **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.

Using the mbed Timeout

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. The API summary is shown.

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

A Simple Timeout Application

```
/*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);     //blinks in time with main while(1) loop
DigitalOut led2(LED2);     //set high fixed period after button press
DigitalOut led3(LED3);     //goes high when button is pressed

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);
  }
}
```

This Program Example causes an action to be triggered a fixed period after an external event. If the button is pressed, the **blink( )** function gets attached to the **Response** Timeout. The program is a microcosm of many embedded systems - a time-triggered task needs to keep going, while an event-triggered task takes place at unpredictable times.

Using the mbed Ticker

The mbed Ticker sets up a recurring interrupt, which can be used to call a function periodically, at a rate decided by the programmer. The API summary is shown.

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 micro-seconds
attach_us	Attach a member function to be called by the Ticker, specifying the interval in micro-seconds
detach	Detach the function

Applying Ticker to the First Example Program

Creating a periodic event is one of the most common requirements in an embedded system. This program switches the LED every 200 ms, using Timeout rather than a **wait( )** function.

```
/* 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);

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

    }
}
```

The Real Time Clock

The Real Time Clock (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.

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

Switch Debouncing

The mechanical contacts of a switch literally bounce together, as the switch closes. This lasts for a few milliseconds, and can cause a digital input to swing wildly between Logic 0 and Logic 1 for a short time after a switch closes, as illustrated.

There are several techniques, in hardware and software, which allow *switch debouncing*.

Ideal switch response

Actual switch response

Demonstrating switch bounce

Event driven LED Switching with Switch Debounce

This program 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.

```c
/* 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
}
```

Introducing the Real Time Operating System (RTOS)

Programs so far in this book have almost all been structured around a main loop (sometimes called a *super loop*), as symbolised. This 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 intermittent, or the tasks or ISRs cause unacceptable delay to each other.
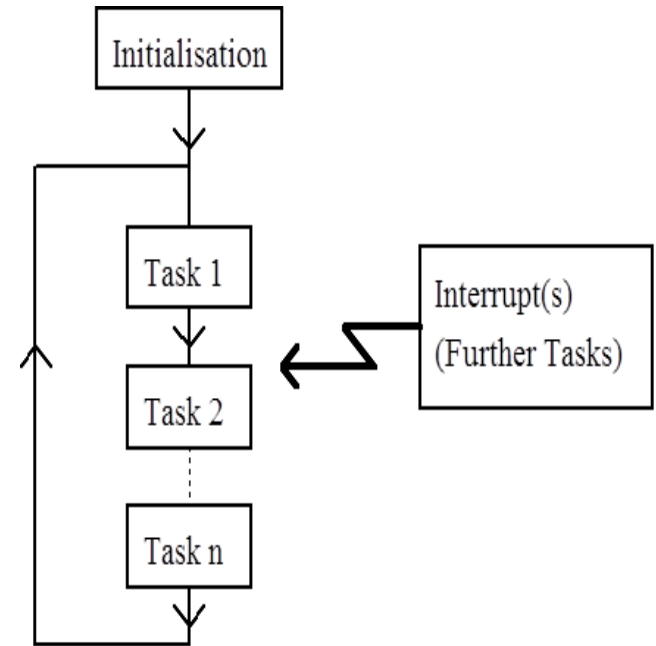
The RTOS provides a different approach to program development. With the RTOS, control of the CPU and all system resources are handed to the operating system (OS). It is the OS which now determines which section of the program is to run, for how long, and how it accesses system resources. The application program itself is subservient to the OS.

RTOS Tasks

A program written for an RTOS is structured into tasks or *threads.* Each task is written as a self-contained program module. The tasks can be prioritised, 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 synchronisation between tasks,

- It controls the use of resources shared between the tasks, for example memory and hardware peripherals.

RTOS Scheduling

An important part of the RTOS is its *scheduler,* which decides which task runs and for how long.

A simple example is the *Round Robin scheduler,* as illustrated. The scheduler synchronises its activity to a *clock tick,* a periodic interrupt from an internal Timer, like the mbed Ticker.

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 - whatever task is executing suspends its activity mid-flow, and waits for its turn again.

Round Robin scheduling doesn't allow task prioritisation. There are other forms of scheduling which do.

Other features of the RTOS allow tasks to be synchronised, or to pass data between each other.

Round Robin Scheduling, Three Tasks Running in Turn
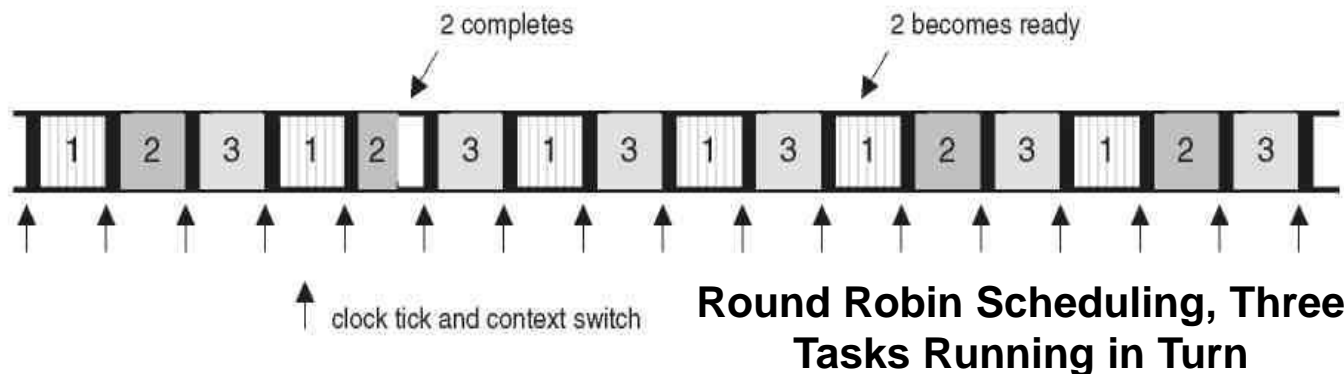
Chapter Review

- 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.