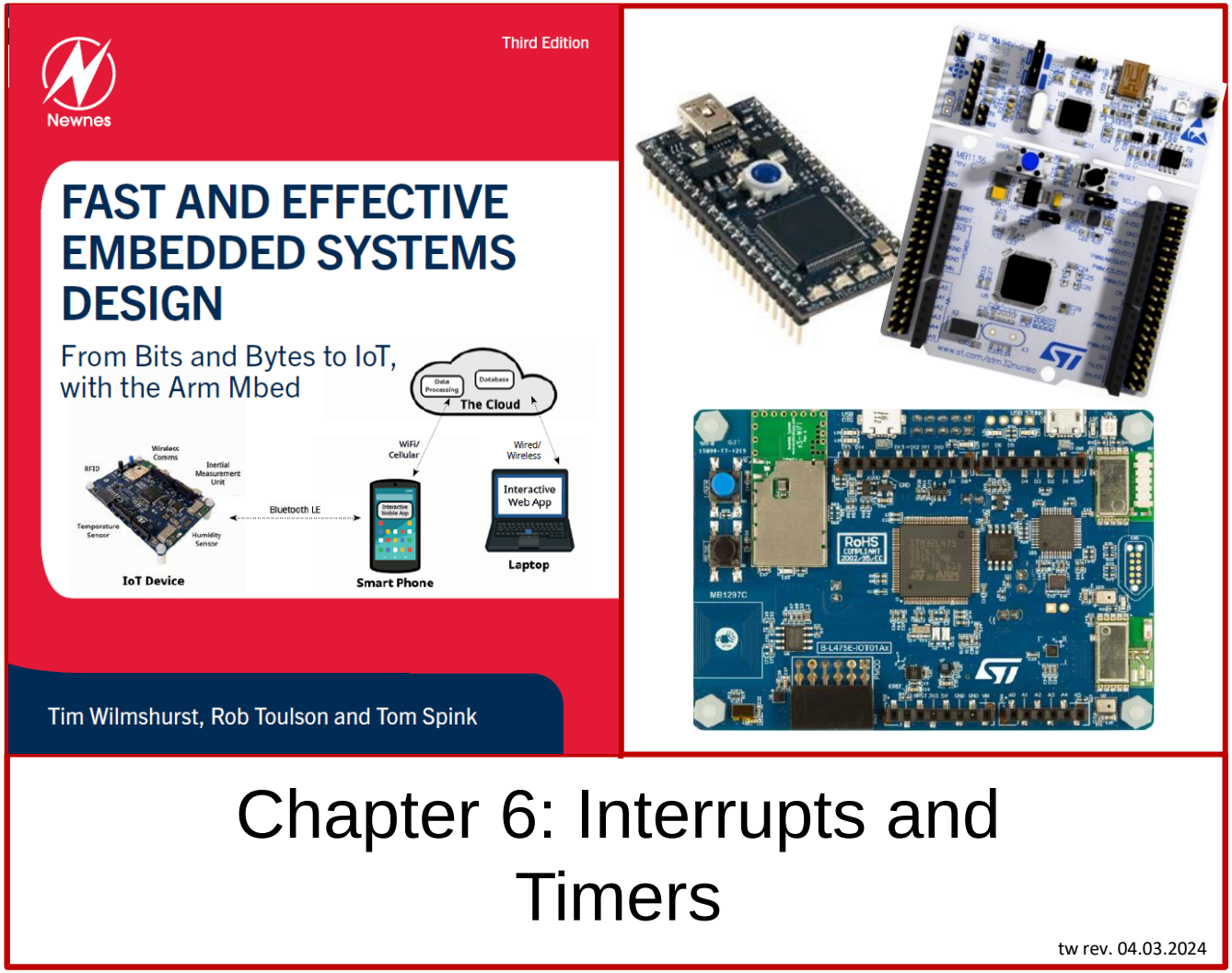


Chapter 6: Interrupts and Timers

tw rev. 04.03.2024

If you use or reference these slides or the associated textbook, please cite the original authors' work as follows:

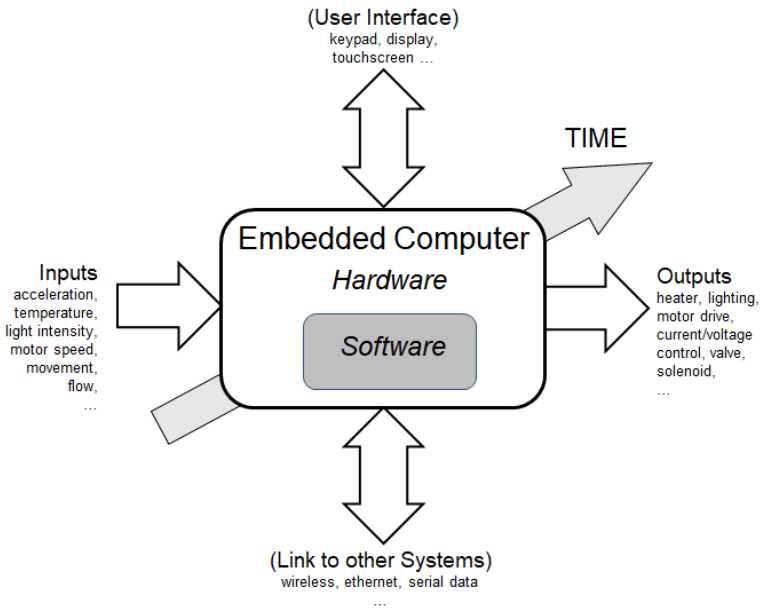
Wilmshurst, T., Spink, T. & Toulson, R. (2024). Fast and Effective Embedded Systems Design – From Bits and Bytes to IoT, with the ARM Mbed (3rd edition), Newnes, Oxford, ISBN: 978-0323951975.

www.embedded-knowhow.co.uk

Thinking about Time 1

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:

- Respond with appropriate speed to external events, which occur at unpredictable times;
- Measure time durations;
- Generate time-based activities, which may be single or repetitive.



Thinking about Time 2

We need a set of tools and techniques to allow effective time-based activity to occur.

A Timer is a digital circuit which enables time measurement, and hence can make things happen when a certain time has elapsed.

An interrupt is a mechanism whereby a running program can be interrupted by an external input, with the CPU then being required to jump to some other activity.

We are going to see some simple activities triggered by external events, and some triggered by time.

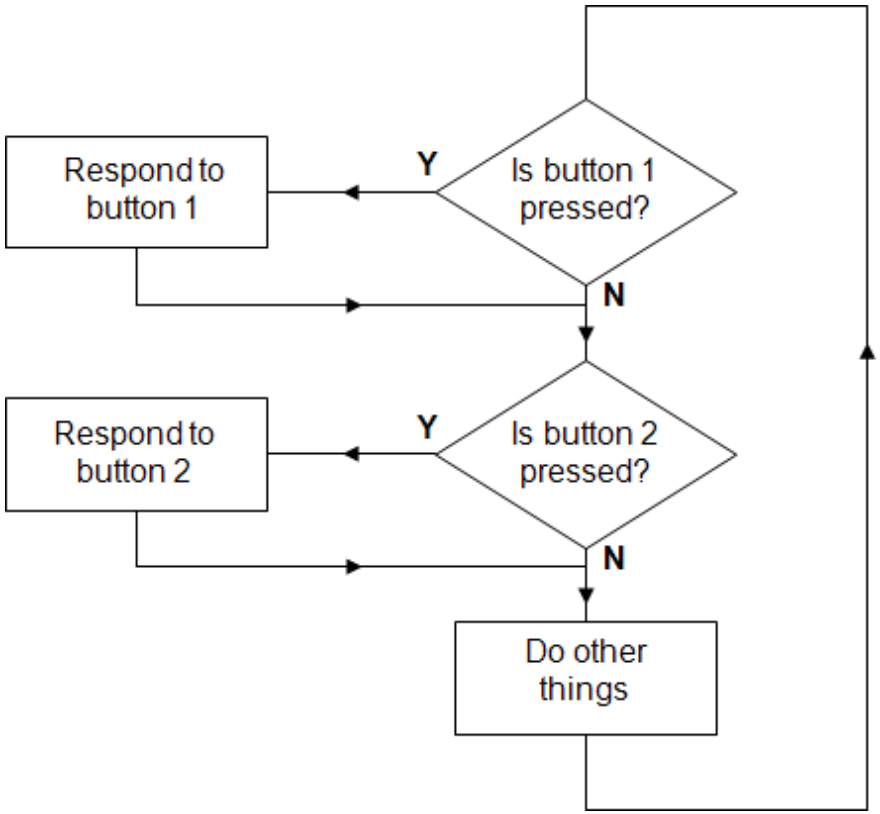
Let’s call these activities *tasks*, and refer to *event-triggered* and *time-triggered* tasks.

Polling

One way of programming for an external event, like a button push, is to continuously test an 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.

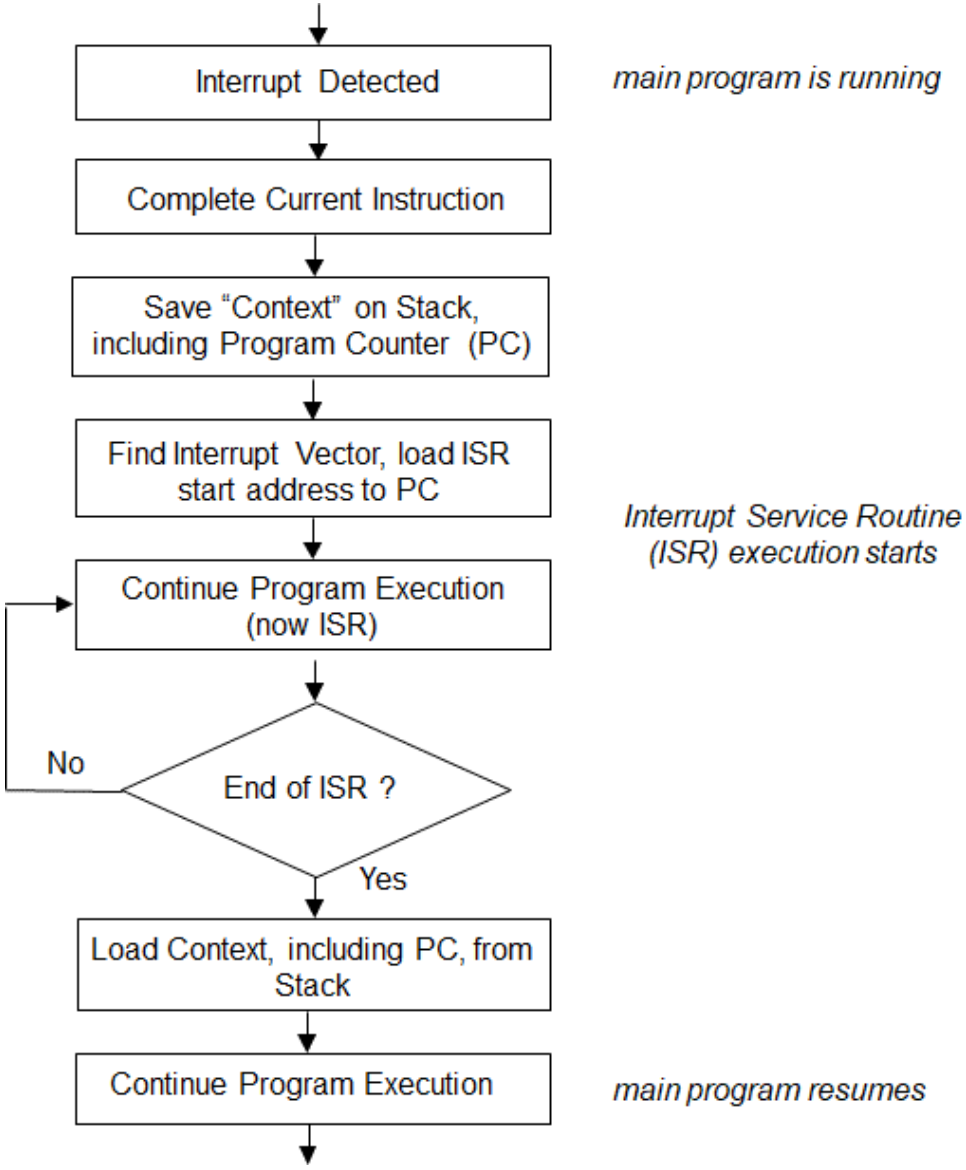


Introducing Interrupts

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.



Interrupts with the Mbed API

The interrupt capability of most microcontrollers, like the LPC1768 or FE401RE, is sophisticated and complex. The Mbed API exploits only a small subset of this capability, focussing on the external interrupts. Some of the API functions are shown in the Table. Using these we can create an interrupt input, write the corresponding ISR, and link the ISR with the interrupt input. Most digital I/O can be used for interrupts.

Function	Usage
Constructor	
`InterruptIn (PinName pin)`	Create an InterruptIn connected to the specified pin, of given name.
Member functions	
`void rise (callback<void()> func)`	Attach a function to call when a rising edge occurs on the input.
`void fall (callback<void()> func)`	Attach a function to call when a falling edge occurs on the input.
`void enable_irq()`	Enables the interrupt.
`void disable_irq()`	Disables the interrupt.

Introductory Use of an Mbed Interrupt

In this example, 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.

```
/* Program Example 6.1: Simple interrupt example. External input causes interrupt,
while led flashes.
Runs on either LPC1768 or F401RE with pin connections shown below
                                                                */

#include "mbed.h"
//define and name the interrupt input
InterruptIn button(p5);    //Replace p5 with BUTTON1 for F401RE
DigitalOut led(LED1);
DigitalOut flash(LED4);    // Replace LED4 with A0 for F401RE

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(true) {          // continuous loop, ready to be interrupted
       flash = !flash;
       ThisThread::sleep_for(100ms);
    }
}
```

Getting Deeper into Interrupts

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

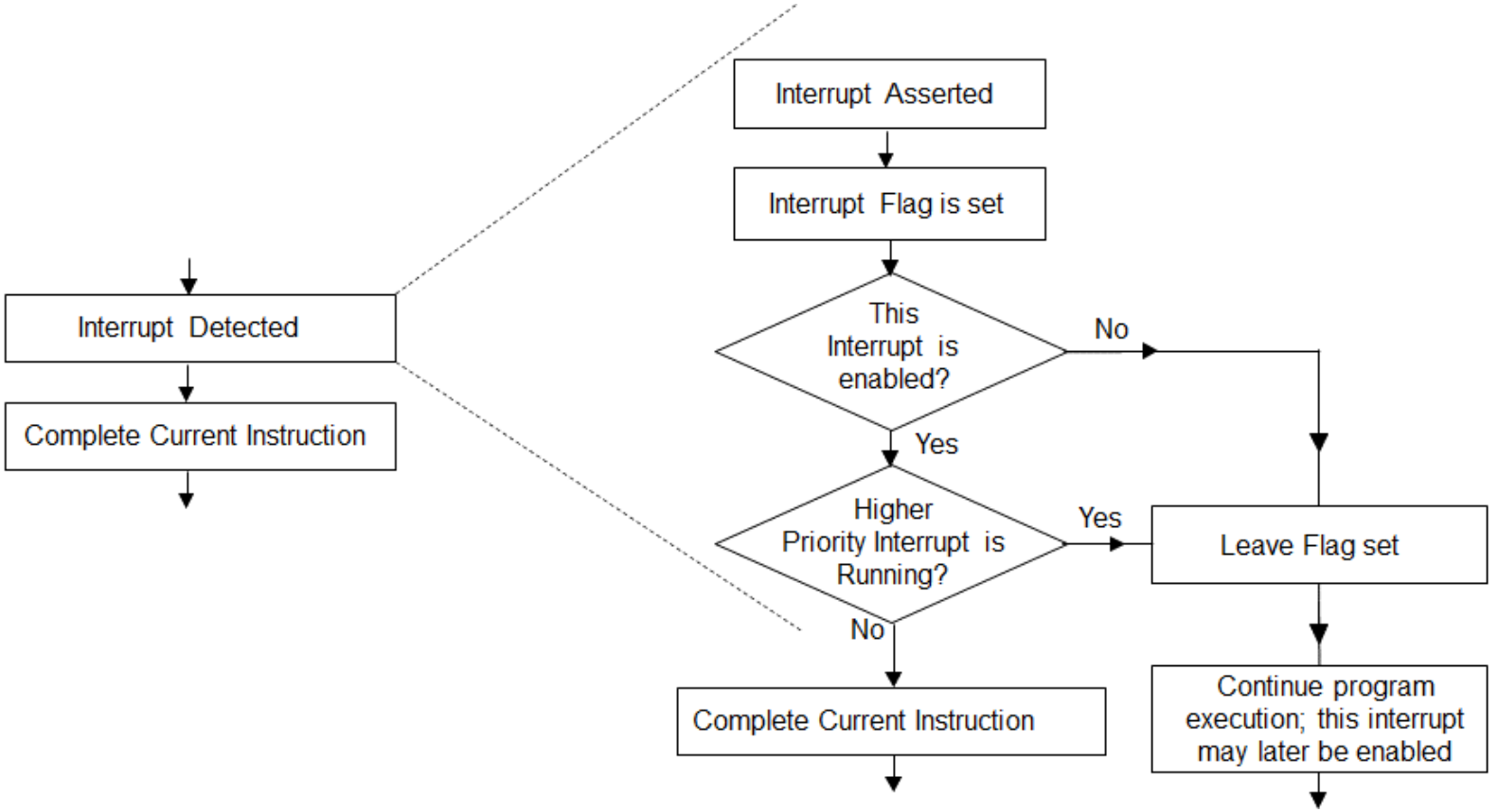
- Interrupts can be *prioritised,* - some can be defined as more important than others. If two occur at the same time, then the higher priority one executes first.
- Interrupts can be *masked*, i.e. switched off, if they are not needed, or are likely to get in the way of more important activity. This 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.

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

The delay between the interrupt occurring, and the processor responding, is called the interrupt *latency.*

A Typical Microprocessor Interrupt Response –

some greater detail



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 6.2: Tests interrupt latency. External input causes interrupt,
which pulses external LED, while LED1 flashes continuously.
Runs on either LPC1768 or F401RE with pin connections shown below
                                                                        */
#include "mbed.h"
//Connect input square wave here
InterruptIn squarewave(p5);    //Replace p5 with D5, for Nucleo F401RE
DigitalOut led(p6);            //Replace p6 with A0, for Nucleo F401RE
DigitalOut flash(LED1);

void pulse() {                 //ISR sets external led high for fixed duration
  led = 1;
  wait_us(2000);
  led = 0;
}

int main() {
  squarewave.rise(&pulse);  // attach the address of the pulse function to
                                                        // the rising edge
  while(true) {             // interrupt will occur within this endless loop
    flash = !flash;
    ThisThread::sleep_for(250ms);
  }
}
```

Disabling Interrupts

Because interrupts can occur at any time, they can have undesirable side effects.

There may be code sections - sometimes called *critical regions* - when 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.

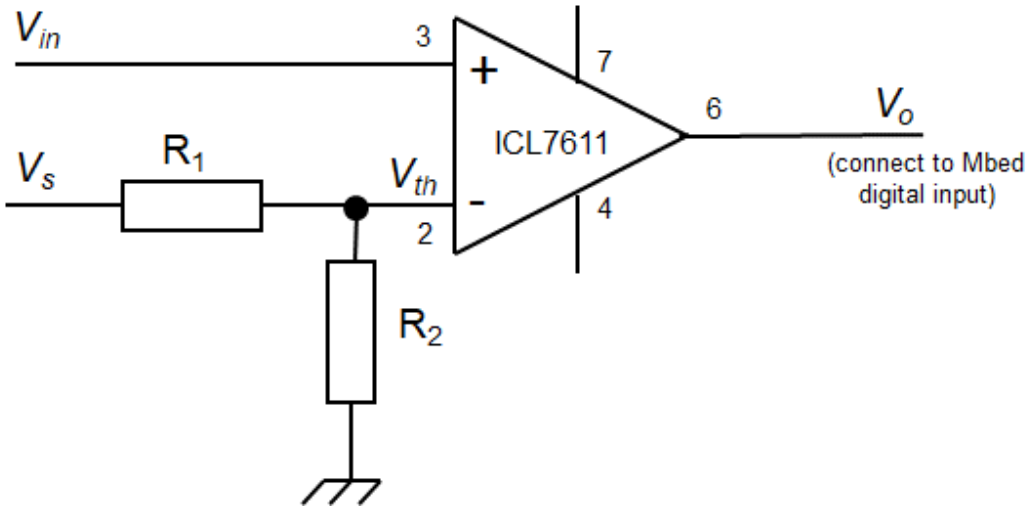
In general an incoming interrupt which is masked will leave its flag set, so a response can be made once the interrupt is enabled again. Of course a delay in response has been introduced, and the latency much compromised.

The compiler allows interrupts to be disabled, for example as below. This may be hardware specific, so needs checking. For the interrupt name used in the previous Program Example, these lines can be used:

```
squarewave.disable_irq();                    //disable squarewave interrupt
... //insert here activity which must not be interrupted
squarewave.enable_irq();                    //enable squarewave interrupt
...
```

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.



The Digital Counter

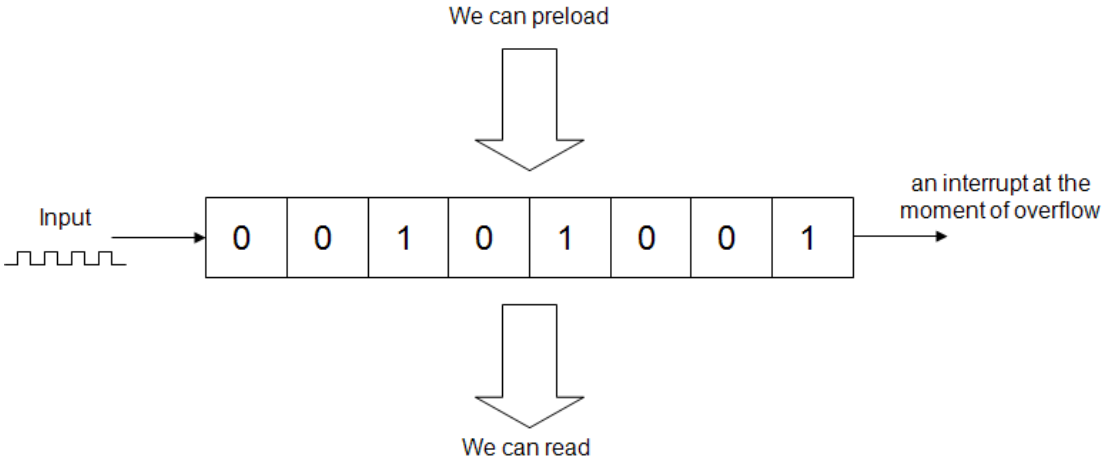
It is easy in digital electronics to make electronic counters by connecting together a series of bistables or flip-flops.

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.



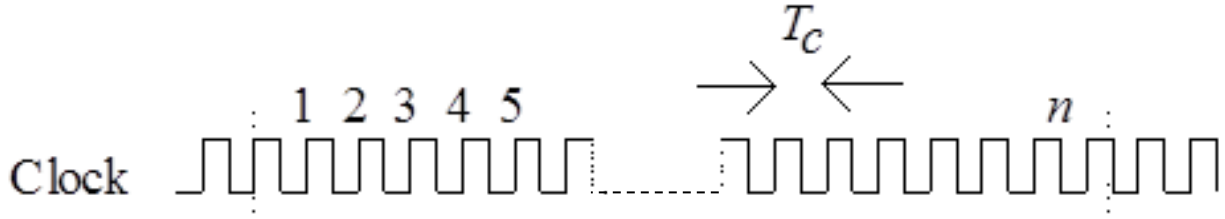
Counting and Timing

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.



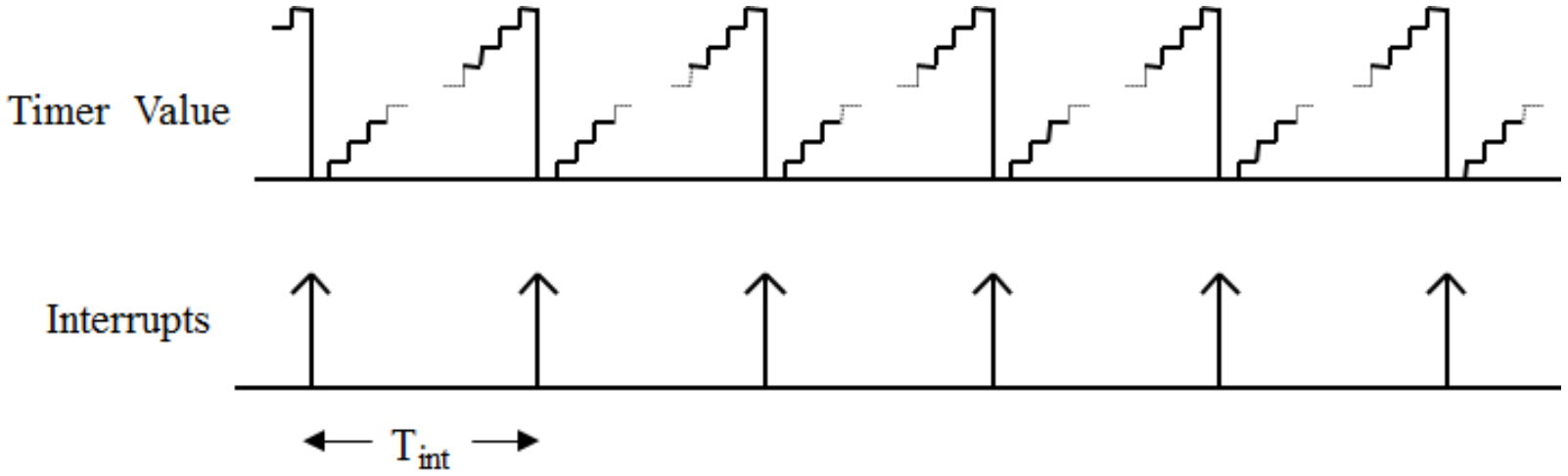


Counting and Timing – Interrupt on Overflow

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.



The Mbed Timer API

The Mbed Timer allows stopwatch-like timing activities to take place. A Timer can be created, started, stopped, read and reset. There is no limit to the number of Timers that can be set up. The API summary is shown below.

Mbed Timers are based on 64-bit signed microsecond counters, giving a timing range of up to 250,000 years!

Function	Usage
Constructor	
`Timer` *name*	Create a Timer, called *name*.
Member functions	
`void start()`	Start the timer
`void stop()`	Stop the timer
`void reset()`	Reset the timer to 0
`duration_cast<milliseconds>(timer.elapsed_time()).count()`	Read from *timer,* if integer milliseconds are needed
`duration_cast<microseconds>(timer.elapsed_time()).count()`	Read from *timer,* if integer microseconds are needed

A simple Timer Application

This Program Example measures the time taken to write a message to the screen. It displays that message on screen, displays the time taken to write it, and the overall elapsed time.

```
/* Program Example 6.3: A simple Timer example, from mbed web site.
  Works on LPC1768 and F401RE, without adjustment or extra components */
  #include "mbed.h"

  using namespace std::chrono;   //Invoke chrono in C++ standard library
  Timer timer_1;    //Create two timers
  Timer timer_2;
  DigitalOut LED (LED1);

  int main(){
    timer_1.reset();
    timer_1.start();
    while(true){
      timer_2.start();
      printf("Hello World!\n");
      timer_2.stop();
      printf("The time taken was %llu milliseconds\n", duration_cast<milliseconds>(time
  r_2.elapsed_time()).count()); //!!ensure statements above/below fall on single lines
      printf("The total time elapsed is %llu milliseconds\n", duration_cast<millisecond
  s>(timer_1.elapsed_time()).count());
      ThisThread::sleep_for(500ms);
      timer_2.reset();
      LED = !LED;
    }
  }
```

Using Multiple Mbed Timers

```
/*Program Example 6.4: Program which runs two time-based tasks Works on both LPC1768
and F401RE, with connections as indicated below  */
#include "mbed.h"
Timer timer_fast;          // define Timer with name "timer_fast"
Timer timer_slow;          // define Timer with name "timer_slow"
DigitalOut ledA(LED1);     //Use A0 for F401RE
DigitalOut ledB(LED4);     //Use A1 for F401RE

using namespace std::chrono;
void task_fast(void);                //function prototypes
void task_slow(void);

int main() {
  timer_fast.start();    //start the Timers
  timer_slow.start();
  while (true){
    //test fast Timer value
    if (duration_cast<milliseconds>(timer_fast.elapsed_time()).count()>110){
      task_fast();               //call the task if trigger time is reached
      timer_fast.reset();            //and reset the Timer
    }
   //test slow Timer value
   if (duration_cast<milliseconds>(timer_slow.elapsed_time()).count()>200){
      task_slow();
      timer_slow.reset();
    }
  }
}
void task_fast(void){
    ledA = !ledA;
}
void task_slow(void){
    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
Constructor	
`TimeOut name`	Create a TimeOut, called *name.*
Member functions	
`void attach(function address, float delay)`	Attach a function to be called by the Timeout, specifying the **std::chrono** delay to occur before function is called.
`void detach()`	Detach the function.

A Simple Timeout Application

```
/*Program Example 6.5: Demonstrates Timeout, by triggering an event a fixed duration
after a button press.
Works on both LPC1768 and F401RE, with connections as indicated below
                                */

#include "mbed.h"
Timeout Response;          //create a Timeout, and name it "Response"
DigitalIn button (p5);    //Replace p5 with BUTTON1, for F401RE
DigitalOut led1(LED1);
DigitalOut led2(LED2);   //Replace LED2 with A0, for F401RE
DigitalOut led3(LED3);   //Replace LED3 with A1, for F401RE

void blink() {           //this function is called at the end of the Timeout
  led2 = 1;
  wait_us(400000);      //light LED for 400 ms (avoid sleep functions in ISRs)
  led2 = 0;
}

int main() {
  while(true) {
    if(button == 1){
      //attach blink function to Response Timeout, to occur after 2 seconds
      Response.attach(&blink,chrono::milliseconds(2000));
      led3 = 1;                        //show button state
    }
    else led3 = 0;
    led1 = !led1;    //indicate looping in progress
    ThisThread::sleep_for(200ms);
  }
}
```

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.

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
Constructor	
`Ticker name`	Create a Ticker, called *name.*
Member functions	
`void attach(function address, interval)`	
`void attach (callback<void()> func, std::chrono::microseconds interval)`	Attach a function to be called by the Ticker, specifying the **std::chrono** *interval.*
`void 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. We can demonstrate Ticker by returning to our very first program example, switching the LED every 200 ms, using Ticker rather than a wait or sleep function.

```
/* Program Example 6.8: Simple demo of "Ticker".
Replicates behaviour of “blinky”, the first led flashing program.
Works standalone on LPC1768 and F401RE, without adjustment                */

#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(){
    //initialises the ticker
    time_up.attach(&led_switch,(500ms));
    while(true){    //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 implemented on many microcontrollers, including the LPC1768 and STM32F401RE.

It 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 continue in operation, even if the rest of the microcontroller is powered down.

The mbed API implements standard functions from the standard C library, as shown. It applies the **tm** structure, which is a format for storing seconds, minutes, hours and date.

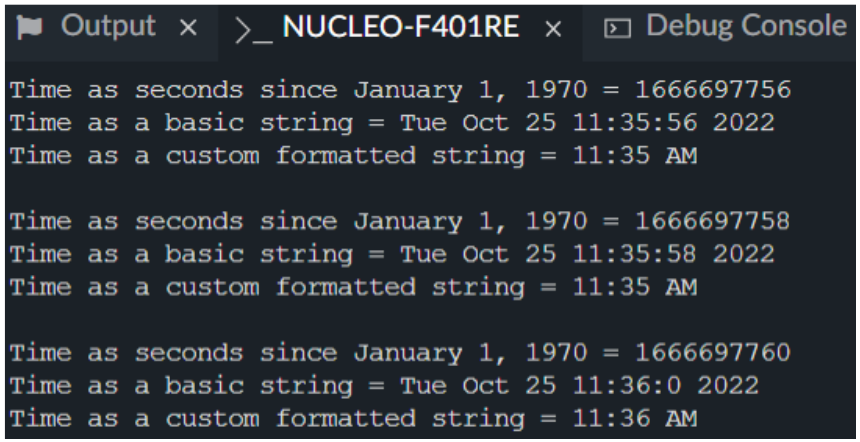
Function	Usage
`time_t time(time_t *t)`	Get the current time (number of seconds since January 1, 1970)
`void set_time(time_t t)`	Set the current time (number of seconds since January 1, 1970)
`time_t mktime(struct tm *t)`	Converts a tm structure to a timestamp
`struct tm localtime(const time_t *t)`	Converts a timestamp to a tm structure
`char ctime(const time_t *t)`	Converts a timestamp to a human-readable string
`size_t strftime(char *str, size_t maxsize, const char *format, const struct tm *t)`	Converts a tm structure to a custom format human-readable string, in destination pointed to by *str*, with specified maximum size and format.

A Simple RTC Application

This example illustrates some RTC features. It starts by loading the RTC with a calculated value for an example time and date in October 2022. At 2 second intervals it then prints out the current time and date in several different formats.

```
/*Program Example 6.10. Demonstrates the Real Time Clock
Works on LPC1768 or Nucleo F401RE, with no external circuit
*/
#include "mbed.h"

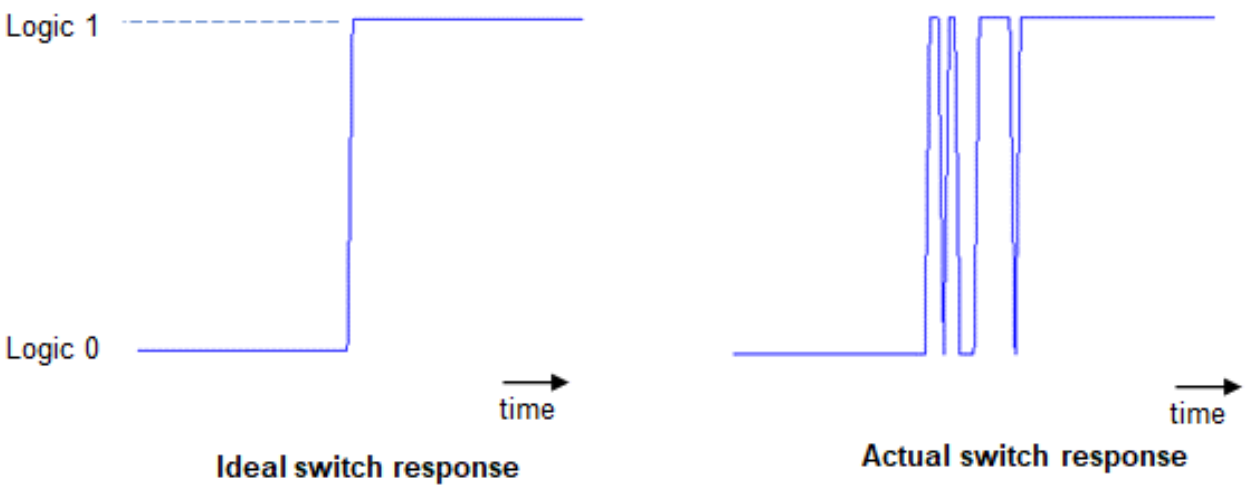
int main(){
  set_time(1666697756);  // Set RTC time to Tues, 25 Oct 2022 11:35:56
  while (true) {
    time_t seconds = time(NULL);   //return elapsed seconds
    printf("Time as seconds since January 1, 1970 = %u\n", (unsigned int)seconds);
    printf("Time as a basic string = %s", ctime(&seconds));
    char buffer[32];
    strftime(buffer, 32, "%I:%M %p\n", localtime(&seconds));
    printf("Time as a custom formatted string = %s\n", buffer);
    ThisThread::sleep_for(2s);
  }
}
```



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*.



Event Driven LED Switching with Switch Debounce

This program is likely to have unpredictable behaviour due to switch bounce.

```
/* Program Example 6.11: Demonstrates switch bounce.
Toggles LED1 every time switch input goes high.
Works on LPC1768 (build shown in Figure 6.3) or F401RE (no external circuit)
                                                                        */
#include "mbed.h"
InterruptIn button(p5);       // Replace p5 with BUTTON1, for F401RE
DigitalOut led1(LED1);
void toggle(void);            // function prototype

int main() {
  button.rise(&toggle);       // attach the address of the toggle
}                                        // function to the rising edge
//This is the ISR
void toggle() {
  led1 = !led1;
}
```

A simple solution to the switch bounce issue is to replace the ISR with this:

```
void toggle() {
  wait_us(4000);
  if(button == 1)
    led1 = !led1;
}
```

The disadvantage of this simple debouncing method is that time is lost during the **wait_us( )** function.

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 elsewhere in the program.
- Interrupts are a powerful addition to the structure of the microprocessor, and allow almost immediate response to external events.
- 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 easily 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 configured through hardware and software in different ways to address different needs; this chapter introduced the Timer, Timeout, Ticker and Real Time Clock.
- Switch debounce is required in many cases to avoid multiple or unreliable responses being triggered by a single switch press.