Wolfson School of
Mechanical, Electrical and Manufacturing Engineering

Loughborough University

ELC055 – Laboratory 1

Timers and GPIO

Introduction

The purpose of the following laboratory exercises is to implement and test some of the theoretical ideas that are introduced in lectures. All the labs will be conducted using a basic microcontroller development board, namely the STM32 F3 Discovery board. This board is similar to an 'Arduino Uno' type device, but with a much higher performance specification and, surprisingly, a lower price tag.
The development board contains very little else other than a microcontroller and a small number of peripheral devices. We therefore have good access directly to the pins of the microcontroller itself allowing us to focus on the mechanics of interfacing with it. We will make use of the onboard peripheral items (LEDs and the like) to test our understanding of the system, then expand this to additional, external devices in later labs.

The learning outcomes for this laboratory are:

- Operate the Kiel uVision IDE to build and load written code to the STM32
- Write appropriate code to configure GPIO ports to output binary signals
- Write appropriate code to implement timer-based interrupts

Hardware Overview

The STM32 F3 Discovery board contains a STM32F303VC microcontroller, which is itself formed from the ARM Cortex M4 architecture. The key features of the microcontroller are [1]:

- 72MHz processing speed, 256KB of Flash memory, 48KB or RAM
- 5 x 16 bit GPIO ports, 1 x 8 bit GPIO port
- 4 x ADC (supporting up to 39 channels, up to 12-bit resolution)
- 2 x 12-bit DAC
- 4 x Operational Amplifiers
- Up to 13 timers, running up to 8MHz
- Communication interfaces covering CAN, I2C, USART/UART, SPIs and USB 2.0

The board is designed in a fairly minimal way in that it has enough circuitry to enable the fundamental functionality of the controller, as well as housing a few peripheral devices, namely;

- Extension header for access to all LQFP100 I/Os (Low-profile Quad Flat Package)
- 8 LEDs connected to Port E with pull-down resisters to ground
- L3GD20, ST MEMS motion sensor, 3-axis digital output gyroscope (connected via I2C)
- LSM303DLHC, ST MEMS linear acceleration sensor (connected via I2C)
- Two push-buttons

Dr. Peter Hubbard - *p.d.hubbard3@lboro.ac.uk*

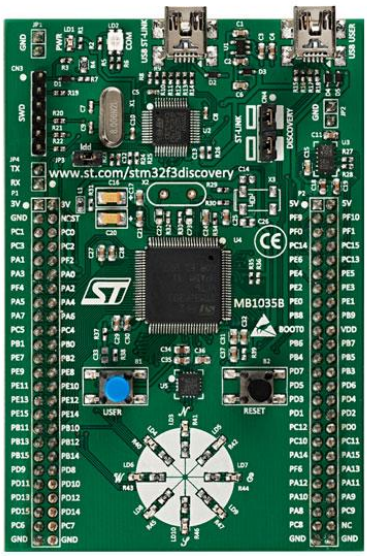


Figure 1- STM32 F3 Discover Board [1]

Integrated Development Environment (IDE) Overview

For the purposes of code development, we will be using the Kiel uVision 5 development environment. The Kiel development package provides a number of fundamental header files that allow us to programme and set system registers. A further attribute of the Kiel IDE is that it provides a 'de-bugging interface' that allows the monitoring of ports and variables during programme execution.

Figure 2 - Kiel uvision 5 IDE [2]

The general structure of development environments is that they attempt to 'ease' the process of managing the registers in the microcontroller with a series of APIs. Whilst possibly doing this, they also add a layer of confusion that can both generate unnecessarily cumbersome code, and can disguise the actual functions occurring. During the labs, as many of the middle layer processes will be circumnavigated to create simple, low-level, high-performance code with small overheads.

To aid the completion of the laboratory activities, you will be provided with an outline uVision project structure that will gather all the necessary header files required for code generation. The complete listing of files is shown in Figure 3.

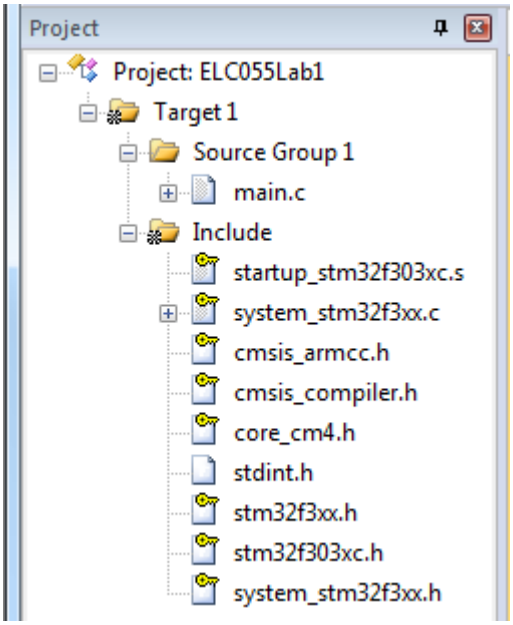


Figure 3 - Project Window for Laboratory distribution

The files have the following functions:

main.c	This is the primary file that you will edit to define the require functions
stdint.h	Defines basic 'c' operators and variables such as integers
stm32f303xc.h	Defines the data structures and address mapping for all the system peripherals, including the peripheral register declarations
startup_stm32f303xc.s, system_stm32f3xx.h, stm32fx3xx.h	Fundamental files that define the link between the user code and the chip architecture
core_cm4.h	CMSIS Cortex-M4 core peripheral access layer header file
cmis_compiler.h, cmsis_armcc.h	Necessary information to convert 'c' to distributable machine code

An introduction to Timer-based Interrupts

The STM32 has up to 13 timers on-board the chip. A timer is simply a binary counter device that is driven by automatic clock pulses. Timers not only provide a temporal measuring point, but are able to generate interrupt signals to create PWM signals and manage system functions. It is also possible to use external triggers as a pulse input to the counter should this be useful.

This laboratory will use timer-based interrupts. This method provides a way of creating regular, repeated functions that operate in priority over other non-time-critical tasks. This explanation will work in two steps: Firstly, how to correctly setup a timer such that an interrupt is generated at the correct time, and secondly how to correctly handle the interrupt so that the required function is initiated.

Timers

The two features of the timer that will be used here are the Pre-scaler (PSC) values and the Auto-Reload Register (ARR). Consider the simplified diagram of the counter architecture, where the original [1] is fairly complicated:

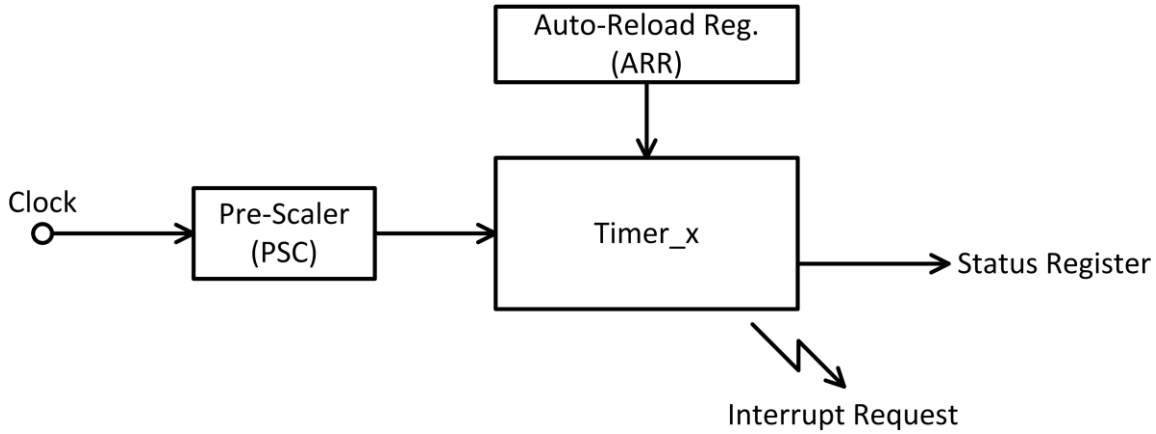


Figure 4 - Counter Architecture

The Pre-Scaler is essentially an item that slows the clock. A value is defined that states how many clock pulses are required before a counter is registered. If this value is '19' for example, then the clock would have to submit 19 pulses before the timer is incremented on the 20th pulse.
The auto-reload register is can be used in a number of ways. In this instance, the timer will count up to the value stored in the ARR, then reset itself and trigger an interrupt request. The clock pulses are directed to the timer by the following command:

```
RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;
```

The formula for calculating the time in seconds before the counter resets is therefore given by:

$$Time\ taken = \frac{(PSC + 1)}{Sysclk}(ARR + 1)$$

Sysclk is the speed the main system clock oscillates, which has a default value of **8MHz** for this board. There appears to be no hard and fast rule about choosing an appropriate value of PSC and ARR to achieve a fixed time, but it seems sensible that a smallish PSC value is chosen in case pulses are missed. For example, if an interrupt is required once every 1 second, a PSC of 7.999999M seems excessive. These values are set by defining the values in the TIMx type_def structures using the following method:

```
TIMx->PSC = 100; // prescalor value in Timer 'x' as 100
TIMx->ARR = 1000; // Auto-Reset Register of Timer 'x' set to 1000 counts
```

Note: 'x' refers to the Timer number eg: TIM3 is timer '3'.

The timer action is set in motion with the 'enable' command:

```
TIMx->CR1 |= TIM_CR1_CEN;
```

Interrupts

Interrupts can occur from a number of sources such as a software trigger, a counter overflow or an external input such as a switch or pulse. Even from within a single timer, an interrupt request can be generated from a number of potential triggers. Often, interrupt triggers are grouped such that any single trigger occurring within this group will raise an interrupt request that appears the same as if any other trigger within the group activated it.
Handling requests therefore take two stages; firstly, acknowledging that the interrupt has occurred, then secondly, identifying the specific trigger of the request.

With the timer example, the first task is to set the appropriate registers such that we allow the timer to generate an interrupt. This takes two steps.

1. Set the appropriate bits In the 'TIMx->DIER' register (**DMA/Interrupt Enable Register**) so that the event of 'reaching the ARR value' will enable an interrupt from the timer to be triggered
2. Set the appropriate flag in the **Nested Vector Interrupt Controller** (NVIC) to tell it to look out for an interrupt from Timer 'x'.

These steps are performed using the following lines of code [3]:

```
TIMx->DIER |= TIM_DIER_UIE; // Set DIER register to watch out for an
'Update' Interrupt Enable (UIE) – or 0x00000001

NVIC_EnableIRQ(TIMx_IRQn); // Enable Timer 'x' interrupt request in NVIC
```

When the counter reaches the value stated in ARR, two actions occur as a result:

1. The TIMx->SR (**Status Register** for the timer) is updated to specify the exact cause of the interrupt. In this case, reaching the ARR value triggers an 'Update' interrupt flag, and sets the LSB in the TIMx->SR register to '1'.
2. It will raise an interrupt that is then registered in the NVIQ. This in turns, instigates a function named 'TIMx_IRQHandler'. This action of this function is required to be defined in the main.c file.

For this particular interrupt, the function should take the following form:

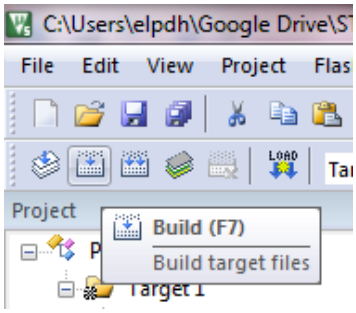
```
void TIMx_IRQHandler()
{
	if ((TIMx->SR & TIM_SR_UIF) !=0) // Check interrupt source is from
		the 'Update' interrupt flag
	{
	...INTERRUPT ACTION HERE
	}
	TIMx->SR &= ~TIM_SR_UIF; // Reset 'Update' interrupt flag in the SR
	register
}
```

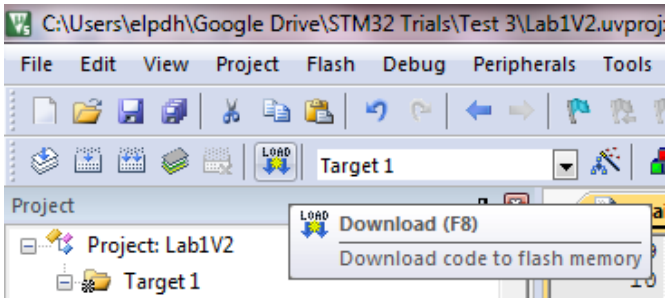
Laboratory Tasks

Task 1: Create, build and deploy 'c' code to development board

This task is simply to obtain the project file from the learn server and execute the 'main.c' file.

1.1. Download the .zip file 'Lab1.zip' and extract the folder contained within it to a working directory
1.2. Open the uVision project file 'ELC055Lab1.uvprojx'. There may be a small delay whilst packages install
1.3. Open the main.c file. Build and deploy the target by either clicking on the icons in the tool bars, or pressing F7 then F8





1.4. When the code has deployed, press the reset button on the board to start the programme.

Task 2: Multiple GPIO Outputs

The code provided makes LED 4 flash intermittently. During this exercise we will be making all 8 LEDs flash in sequence.

2.1. The LEDS are connected to GPIO port E pins 8-15, with pull up resistors to Vcc (5V). The number of the LEDs do not relate to the pin of port E of which they are connected.
   a. Sketch the circuit diagram of the internal and external components for a single LED, using the configuration defined in main.c
2.2. Change the code so all 8 LEDs are activated and flash at the same rate.
2.3. Change the code so that all 8 LEDS are used as a binary counter (i.e. from 0b00000000 to 0b11111111). Try and make the count increment once a second.

Task 3: Replacing 'delay' with interrupt

Currently, the timing of execution is provided by a custom 'delay' (originally lines 38-48). This function simply loops a lot of times in order to occupy the processor. A more scientific approach would be to use an interrupt to accurately manage the time of execution

3.1. Design values for PSC and ARR for an 'Update' interrupt to occur once every second
3.2. In the 'main' function, add the lines of code required to assign these values to Timer 3.
3.3. Enable an interrupt to be recognised for an 'Update' interrupt on Timer 3. Define an interrupt function that makes a single LED flash on and off
3.4. Modify the interrupt function such that the LEDs count up in binary at a rate of 1 increment per second

References

[1] STM, "STM32F3Discovery," 2017. [Online]. Available: http://www.st.com/en/evaluation-tools/stm32f3discovery.html. [Accessed: 16-Nov-2017].
[2] Kiel, "Kiel uVision IDE," 2017. [Online]. Available: http://www2.keil.com/mdk5/uvision/. [Accessed: 16-Nov-2017].
[3] STM, "AN4776 Application note: General Purpose Timer Cookbook," 2017.