ELEC-240 Lab2

Controlling GPIOs with Blocking Delays on the STM32F429 Nucleo-144 Development Board

1. Introduction

This lab exercise introduces you to Loops and Conditional Loops in embedded systems. Loops have a variety of functions one of which is creating crude delays.

To generate a crude delay the programmer can command the micro-controller to perform a pointless task, of counting upto a large number, which takes up time before moving on to the next relevant tasks.

The easiest way to achieve this is to get the micro-controller to increment a variable from zero up to a pre-defined amount to generate the desired delay. The larger the pre-defined value, the longer it will take the micro-controller to count to that value hence a longer delay.

A VERY approximate equation for determining the pre-defined value is

$$Val = \frac{F_{APB}}{4} \times T_{Delay}$$

Where F_{APB} is the speed of the Advanced Peripheral Bus. The default the speed is 16Mhz on bootup. Depending on the timing specificness of the application this value can then be fine-tuned through trial and error.

Delays of this type are extremely crude as the actual time taken to count up to the pre-defined value can be affected by many factors such as how long it's taken to run the other tasks in the program where the delay is located and interrupt service routines that may be running which will put the delay loop on hold while they run.

Note: In some instances the compiler can optimise this type of delay and remove it altogether!, so beware

1.1 Learning Outcomes

- By the end of this lab exercise you should be able to:
  - ✓ Demonstrate an understanding of how blocking loops work
  - ✓ Create and edit blocking loops to achieve a desired time delay
  - ✓ Create subroutines of various types and demonstrate an understanding of their benefits
  - ✓ Demonstrate an understanding of masking and its importance when manipulating control register values

1.2 Code management

It is highly recommended to either download a fresh copy of the example code or to make a copy of the previous Lab example code before editing it for a new lab or task. This is highly advantageous as it allows you to retain older versions of the code (aka Revision History) which can be recalled in the event that you make any irreparable mistakes when editing for a new task.

Please keep this in mind throughout all Labs!

2. Masking

Peripherals on the micro-controller such as the GPIOs are controlled via special registers like the MODE Register (***MODER***), which controls the pin function to be one of four possible conditions

- input,
- output
- alternate function (for onboard peripherals such as USART, SPI, I2C etc)
- analogue (for ADC and DAC)

and the Output Data Register (***ODR***), which controls the pin state (high '1' or low '0').

Both of these registers appear in the example code.

Controlling the peripheral often involves manipulating only certain bits inside these registers which means simply overwriting the current value with a new one, using the = operator is incorrect as it will overwrite the states of the other bits which we do not want to change.

Therefore it is necessary to use Masking (ANDing and ORing) where the existing value in the register is ***Masked*** with a second value so as to set or clear only the relevant bits while leaving the others unchanged.

To achieve this using C code we use the **OR** operator is | and the **AND** operator is &.

Some basic examples are:

- 0011b | 0100b = 0111b 0x03 | 0x04 = 0x07
- 0011b | 0010b = 0011b 0x03 | 0x02 = 0x03
- 0011b | 0000b = 0011b 0x03 | 0x00 = 0x03
- 0011b & 0001b = 0001b 0x03 & 0x01 = 0x01
- 0011b & 0111b = 0011b 0x03 & 0x07 = 0x03
- 0011b & 0100b = 0000b 0x03 & 0x04 = 0x00
-

3 Port Control

The STM32F429 family micro-controllers have two ways of controlling output GPIOs.

Writing to the ***ODR*** register performs a **'Read-Modify-Write'** to control the output on the GPIO pins, where the contents of the register are

i. read into the accumulator,
ii. the relevant bits are changed,
iii. then the new value is written back to the register

It can be seen this method takes up three instructions which can be inefficient and has the potential for an error to be caused by race conditions if an interrupt occurs.

Therefore, there exists a Bit Set Reset Register (***BSRR***) which is simply written to and will update a GPIO pin using a single instruction. This is called an ATOMIC instruction since it only takes one clock cycle to execute.

NOTE: The BSRR register uses the = sign to update the register and should not use masking.

Writing a **1** to a bit the **lower half** (bits 0→15) of the ***BSRR*** register will **set** the corresponding pin high;

Writing a **1** to the same bit in the **upper half** (bits 16→ 31) of the ***BSRR*** register will **clear** that same pin;

Task 1

- Download the ***LAB2 example code.zip*** from the DLE and extract to a local directory.

Inside the extracted folder there are several source code files labelled ***Blinky1.c*** to ***Blinky5.c.***

Blinky1.c contains a simple piece of code to flash the Green LED on the Nucleo-144 board. This code is also in its most basic form and while it will successfully build and run it contains many malpractices.

Blinky2.c to ***Blinky4.c*** show step by step how to identify and rectify these malpractices until eventually ***Blinky5.c*** contains the same piece of code but is now written to a professional level.

You can follow step by step how the code is improved by removing ***Blinky1.c*** and importing ***Blinky2.c*** then recompile and run, repeat this process until you reach ***Blinky5.c*.**

Notice that ***Blinky5.c*** contains a ***#include "PLL_Config.c"*** statement at the top of the file that contains the function ***PLL_Config().*** This changes the speed at which the microcontroller runs from the default 16MHz upto 180MHz. The function ***SystemCoreClockUpdate()*** will then update the global variable **SystemCoreClock**.

- Go into debug mode and put a breakpoint where the ***main*** routine calls the function ***SystemCoreClockUpdate()*** and look at the value contained in ***SystemCoreClock*** before it is executed.

  Note: it is a hexadecimal value so you will need to convert to decimal

- Single-step over the function and look at the ***SystemCoreClock*** value after it has been executed. What is the new decimal value?

Task 2

Blinky5.c has shown how to control the Green LED with correctly formatted code.

- Edit the code so the Red LED (PB14) and Blue LED (PB7) replicate the behaviour of the Green LED.

Look at the RM0090 Reference Manual for a summary of the relevant registers involved in setting up and controlling the port pins.

Task 3

- Create a second delay routine called **delay2** that uses a **while** loop instead of a **for** loop to flash the LEDs twice per second and test it. This will involve creating a new subroutine. Refer to Table 2 for information.

Task 4

- Create a third delay routine called **delay3** that can be called with a variable (input argument) that defines the time of the delay in milliseconds. Test it. Refer to Table 2 for information.

Task 5

- ❖ Modify the code to use the **BSRR** register instead of **ODR** to turn the LEDs ON and OFF.

Task 6

- ❖ Write a program to send an **SOS** message in Morse code using the LEDs. Refer to Table 2 for Morse code information.

4. Support Documentation

UM1974 User manual

- Pin identification and the supported special functions
- Circuit schematics
- Jumper and component identification
- Header pinouts

RM0090 Reference manual

- MCU memory and peripherals architecture
- Peripheral control registers, addresses and bit-fields

While Loop (wiki)

- Refer to the “C programming language” section

For Loop (wiki)

- Refer to the “Loop Counter” section for C-programming example

Subroutines in C

Morse Code Information Page (wiki)

- Development and history of morse code
- Brief overview of morse code timing

Morse Code Look-up chart

- Complete morse Look-up chart for alphabetical characters