Introductory Laboratory

The STM32F4 Discovery and Keil µVision programming environment

Aim

There are two aims for this introductory lab.

1. Develop an understanding about the important general characteristics of the equipment, software and resources that will be used to gain practical experience in embedded systems.
2. Develop the skills and knowledge that will allow you to take the steps required to use the equipment and develop a basic program. At the end of this lab you will be in a position to conduct independent inquiry based learning (I encourage you to explore the capabilities of the equipment as much as possible in the available time) and to tackle the first assessed lab.

Learning process

Once you have collected your STM32F4Discovery kit you will be able to proceed with this lab. This lab sheet takes you through the steps required to create a project using the STM32F4 Discovery and Keil µVision programming environment. It is quite a long document, because it explains each stage in the process of creating a project and contains example code and solutions to the tasks. It is generally self-contained. However, at times you will need to refer to other documents. You will be prompted when this is necessary and please take the effort to do so, as you will need to use these other documents to complete the assessed labs and assignment.

The purpose of the take home kit is to allow you to spend as much time as possible using the kit. It also allows you to learn independently and to explore the capabilities of the kit on your own. Please spend as much time as possible with the kit and work through the introductory lab sheet in your own time. You might complete this introductory lab without any problems and find you do not need much support in the structured lab sessions. Otherwise, you might need some support from the demonstrators in the structured lab sessions. Either way, if you attend the structured lab sessions you must be prepared with specific questions. The demonstrators will not sit down and take you through the laboratory step by step. **It is your responsibility to make a reasonable attempt at the laboratory before you seek assistance.**

Date of lab sessions: Group 1 – Week 2 Tuesday 1200 and Friday 1100
Group 2 – Week 2 Wednesday 1400 and Friday 1300

Support

If you have any problems with the equipment please contact Dr S A Pope immediately using the contact details provided below. **These inquiries should only relate to problems with the equipment, such as suspected faults.** Any inquiry should include a clear description of the problem. For example "It isn't working" would not be an acceptable inquiry on its own. **Any learning based inquiries must be brought to the timetabled support sessions (lectures, webinars and laboratories).**

Contact details

Dr S A Pope

Email: s.a.pope@sheffield.ac.uk

Internal telephone extension: 25186

Room: C07d Amy Johnson Building

Contents

1. Description of the equipment

The carry box contains the following:

1 x STM32F4Discovery

1 x STM32F4Discovery base board (already attached to the STM32F4 Discovery)

1 x Mini to standard connector USB cable

1.1. The STM32F4Discovery

The STM32F4Discovery (Figure 1) is an embedded system that is used for both teaching purposes and for developing more advanced embedded systems. It is based around a 32-bit ARM Cortex-M4 processor. ARM processors are one of the most widely used types of processors in embedded systems due to a combination of low power usage and flexibility. ARM Holdings designs the architecture and the licence for its use is sold to a wide range of processor manufacturers. The microcontroller used here is the STM32F407VGT6 manufactured by STMicroelectronics. Some of the specific details of the microprocessor are covered in Lecture 2 and associated material (demo, further reading sheet and lab).

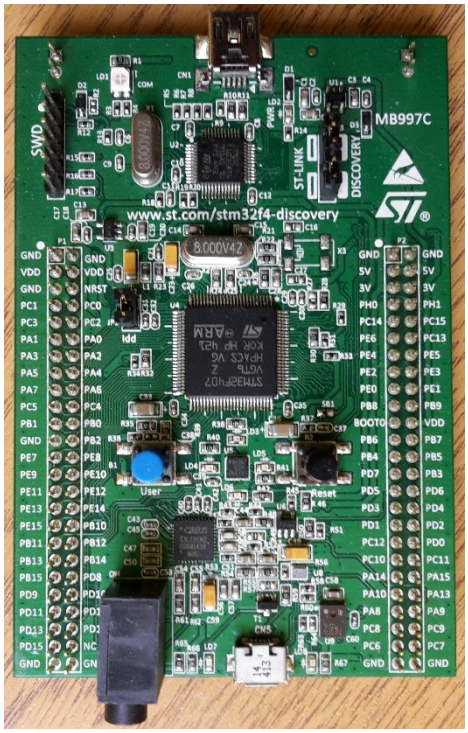

Figure 1 - The STM32F4Discovery (figure taken from the STMicroelectronics UM1472 User manual).

In addition to the ARM processor, the Discovery board contains a number of peripheral components, including an accelerometer (LIS3DSH type), microphone, speaker driver, LEDs and pushbutton. As such it is an embedded system that we can use to gain practical knowledge of many of the aspects of embedded systems design, simulate the operation of real-world embedded systems and use as the foundation for the design and build of more complex embedded systems.

Full details of the STM32F4Discovery is included in its user manual - "STM32F407Discovery - User Manual" – available on MOLE. You will use information in this document (among others) to implement the designs used in this module. Before proceeding further please open and view this document. You will see that it starts with a general description of the device and its layout. It then proceeds to cover its features and peripherals in more detail in section 4.1-4.10. Sections 4.12, 5 and 6 provides details on how the components on the board are connected to the ARM microprocessor and will be required in some of the later labs.

1.2. The STM32F4Discovery base board

The STM32F4Discovery board in your kit is connected to a base board (Figure 2). This base board provides easy access to some of the common interfaces provided by the STM32F407VGT6 microcontroller and allows external peripherals to be easily interfaced with the pins of the microcontroller. In addition it contains standard physical connectors for some of the common interfaces, including Ethernet and Micro SD card. In this module we will not use the interfaces on the

base board during the practical work, but you should be aware of their capability and you might be required to discuss their use in the final assignment.

1.3. The mini to standard connector USB cable

The USB cable is used to connect the Discovery to a PC for both software development and analysis.

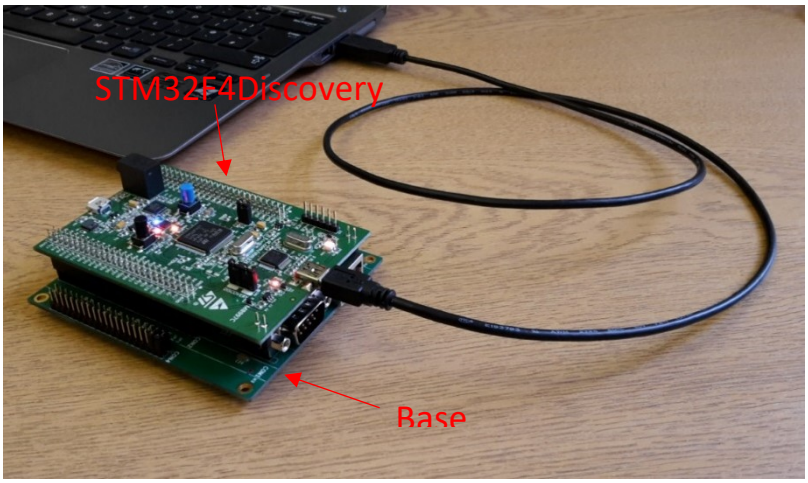

Figure 2 - The STM32F4Discovery, base board and USB connection to a PC.

The mini connector attaches to the Discovery board and the standard connector to the PC (Figure 2).

Important: Please disconnect the USB cable from the Discovery before storing the equipment in the carry box. This will help prevent damage to the sockets on the board.

1.4. The shared Google Drive folder

A google drive folder called "**Embedded Control Systems resources"** has also been shared with you at the start of this module. You will have access to this folder for the duration of the module. It can be accessed either using the hyperlink below, the web-link on MOLE, or directly through your University Google Drive account. Access using private google accounts will not be granted. **If you do not have access to this folder first check to make sure that you are logged into your University Google account and not your private Google account (if you have one).**

Embedded Systems Google Drive folder

The shared Google Drive folder contains the software and resources required to use the Discovery (most of these are also available on MOLE). The folder named "Documentation" contains several separate sub-folders. For easy access you might want to download this folder to either your own computer or your University file storage. The documents in these sub-folder will be referred to at numerous points throughout this module.

1.4.1. Support Documents

The sub-folder called "STM32F4Discovery Documents" contains the data sheets and related documentation for the STM32F4Discovery. These are the:

1. **STM32F407Discovery - User Manual**: Contains information of the Discovery board, including its components and how they are connected. You will need to refer to this when determining how to use the peripheral components on the board.
2. **STM32F4 - Datasheet**: This document contains a summary of all of the important characteristics of the STM32F4 series of processors.

3. **STM32F4 - Processor Reference Manual**: This document contains all of the information that you need to know to use the STM32F407 processor. You will need to refer to it when determining how to setup and use parts of the processor, such as the clocks, timers and ports.
4. **LIS3DSH - Data Sheet**: This document is the data sheet for the accelerometer on-board the STM32F4Disocvery. You will need to refer to it when configuring communication between the processor and accelerometer and when setting up and decoding data from the accelerometer.

Another sub-folder called "General Resources" contains documents that will be either required to conduct the labs, or act as a general resource to aid your learning. In summary, these are:

1. **Embedded Software in C for ARM Cortex M**: This is a user guide for programming the ARM Cortex M-4 processor using the C programming language. It was written by Jonathan W. Valvano and Ramesh Yerraballi. All software written for the Discovery in this module will be written in C. The labs and additional reading document (Bits - Bytes - Logic operations) will take you through some of the main functions and syntax that is required, but you might also need to refer to this document at times when developing your own code.
2. **CMSIS RTOS Tutorial**: This document is a user guide for the Keil RTX Real-Time Operating System (RTOS) that will be used in the final lab before the assignment.
3. **C for Python programmers:** This document by Carl Burch provides an introduction to the C programming language for someone who is familiar with Python.

1.4.2. Keil µVision

Throughout this course on embedded systems we will use **Keil µVision** to develop our projects. You will use this software to write the code for your embedded systems, configure its operation, compile it (convert it form a high level language, which is C in our case, into machine code to run on the microcontroller), load it onto the microcontroller and analyse its behaviour. There are three possible ways in which you can access Keil µVision:

1. **It is available on the managed desktop**. It can be found in the main list of installed software. If it hasn't been used on a particular computer before you will probably need to find and install the software from the Software Centre (you might need to refresh the contents of the Software Centre before it appears in the list). Any projects that you create should be saved to a folder created in the home directory of your U:/ account, otherwise the software might not find the required driver files correctly.
2. **You can install it on your own Microsoft Windows PC**. The folder named "Keil_v5" contains two application files called MDK515.exe. These are ready to install versions of the Keil µVision compiler and development environment. This software is configured to run on a Microsoft Windows based PC (one file is for windows 7 and one for windows 10) and can be installed by double clicking on the MDK515 file. It is a free version and is also available for download from http://www2.keil.com/mdk5/install/ . You can install this version on your own PC (MAC is not supported). To do so please follow the guidance sheet "MDK515_install_RTSlab" which is also in this sub-folder.

We are using a free version of Keil µVision which is limited to a compiled program size of 32KB. This will be sufficient for all work conducted during this module. If your compiled program comes to more than 32KB for any of the work in this module, it is likely to mean that your code is poorly written. An introduction to Keil µVision is covered in the next section of this document.

Important: All software developed during this lab will be in the form of a Keil µVision project and written in C. Software developed using a different language, compiler or development environment will not be assessed if submitted.

1.5. Common Errors with the STM32F4Discovery and Keil µVision

There is a document on MOLE called "Common known errors experienced with the STM32F4Discovery and Keil µVision". It will be updated to include any new issues as the module progresses. Please refer to this document in the first instance if you are receiving errors or having problems with the hardware/software. If this does not solve your problem please contact the teaching team.

1.6. Video guides

In addition to the written lab sheets, you will also find video guides on MyEcho for each of the non-assessed labs. These video guides take you through most of the steps in the lab sheets. They should be used as an additional guide that can aid you in completing the lab. They are a particularly useful aid when trying to implement some of the code or when trying to finding the relevant information. They should be used in conjunction with the lab sheets. Do not rely on the video guides alone. They do not include all of the explanation and detail behind each of the steps in the lab. This can only be found in the lab sheets.

2. Setting up the STM32F4Discovery and starting a Keil µVision project

This section takes you through the steps required to program the STM32F4Discovery with a simple "Blinky" program. Similar setup steps will be required to be completed before the board can be used in the later labs so it is important that you work through these steps carefully. In addition to this detailed step by step written guide, a video guide to implementing these steps is also available on MOLE. However, this video does not cover most of the details behind the steps, it is only a guide to how to implement them. **To be able to successfully complete this module you should consult both this document and the video guide for the introductory lab.**

2.1. STEP 0 - Connecting the STM32F4Discovery and starting Keil µVision

Before you start, first connect the Discovery to a PC using the USB cable as shown in Figure 2. If this is the first time you have connected the Discovery to a specific USB on the computer then a driver will likely be automatically installed.

Second, open Keil µVision. In standard configuration there are several toolbars and drop-down menus across the top. Hovering the cursor momentarily over the top of a button on a toolbar provides some information on its function. Several default windows are created, which contain:

- A "Project" panel on the left which will show the files in any project that you will create. Tabs at the bottom of this window can be used to display other information such as "Books" - i.e. text and pdf files associated with a project.
- To the right of this is a window in which your files will be displayed when opened.
- Across the bottom is the "Build Output" window which shows the result of the output (including error and warning messages) of the compiler.

2.2. STEP 1 - Creating a new project

Click on the “Project” drop-down menu at the top and select “New µVision Project”. In the window that pops up navigate to a directory on the computer or external storage (such as a memory stick or a cloud service such as Google Drive) and create a master folder in which you will store all of the projects that you create for this module. In this master folder create another folder and call it “Intro_lab”. Navigate to this folder and in the “File name” box below the current folder view, type the file name “Intro_lab” and click “Save” in the bottom right of the popup window. These actions have created a folder called “Intro_lab” and in this folder a Keil µVision project called “Intro_lab” has been created. All files created for this project will now be saved in this folder.

2.3. STEP 2 - Selecting the processor type

After you click “Save” another pop-up window will open. In this window you will select the correct processor type for the compiler. The package for the STMicroelectronics family of processors has already been installed on the memory stick. Make sure the “Software Packs” is selected in the pull down menu at the top of the popup window. Next navigate to and select the “STM32F407VGTx” processor using the expansion tabs under the STMicroelectronics tab on the left of the popup window. Click “OK”.

2.4. STEP 3 - Selecting the board options and processor components

The next window to popup is “Manage Run-Time Environment”. In this window you will select all of the components available on the microcontroller that you will need in your project. If you need to change these components at any time later in the project you can re-open the “Manage Run-Time Environment” window using the appropriate icon on the toolbar or the link in the Project->Manage sub-menu. Take some time to have a look at the different components available in this window. In this module we will focus on the components in the “Board Support”, “CMSIS” and “Device” expansion options. For this intro lab you will need to select the following options:

Board Support Select “STM32F4-Discovery” in the pull down menu in the variant column. This tells the compiler that we are using the STM32F407VGTx microcontroller on the STM32F4Disovery board as described in section 1. **Select this option for all projects that you create in this module.**

CMSIS Expand the options by clicking on the plus icon. Select the box for “CORE”. This tells the compiler that we are going to use the main processing core on the microcontroller.

Device Expand the options and select the tick box for “Startup”. This tells the compiler to create a startup file with all of the general configuration parameters for the Discovery.

In the “STM32Cube Framework (API)” option select “Classic”. This option tells the compiler how we will select and setup the microcontroller and component drivers that we will use. With the “Classic” option we will need to manually write the code to configure the microcontroller and components. This is the most general approach and is the best option when learning the fundamental concepts in configuring a microcontroller and associated peripheral components in an embedded system. The box should appear orange when selected. This means that additional packs need to be used with this option. The information on the required packs is shown in the “Validation Output” box in the lower left of the window. These components can be selected automatically by using the “Resolve” button at the bottom. When an option has all required packs correctly selected it will be marked green.

In the “STM32Cube HAL” option (some options will already be selected after resolving the problems with selecting “Classic”) select the “TIM” option. This selects the drivers for the timers available on the microcontroller. Resolve any missing component problems that arise from selecting this option.

Close the “Manage Run-Time Environment” by clicking “OK” at the bottom of the window.

2.5. STEP 4 - Configuring the compiler

Open the "Options for Target" window either from the "Project" pulldown menu in the main Keil µVision interface or using the icon on one of the toolbars (place the cursor over each icon to find out what it means). Configure the following elements by selecting the appropriate tab at the top:

Target — Near the top of the set of options you will find "Xtal (MHz)". This option sets the speed for the external crystal oscillator on the Discovery board that is used to configure the clock frequency. It is probably set to 12.0 as default. Change it value to 8.0 since the Discovery comes with a 8MHz external crystal oscillator.

C/C++ — In the "Define" box near the top type "USE_HAL_DRIVER". This tells the compiler what type of component drivers we are using. In this case we are using the latest "HAL" type drivers supported by the Discovery in all of our work.

Further down in the "Include Paths" box click on the little box to its right that has "…" in it. This brings up a window that allows us to select folder paths that store any files that we will write for our project. Towards the top right of the window select the "New (Insert)" button (the left most of four buttons). This adds a line into the box below. Click on the "…" box in this line and select the folder "Intro_lab" that you created earlier for this project.

Debug — In the "Debug" tab you can select either to use a simulator of the STM32F4 to test our code on (selected using the "Use Simulator" option on the left of the panel) or to select to upload it to the actual hardware (selected using "Use" option on the right). Select to upload it to the hardware by selecting "Use" if it is not already selected by default.

In the pull down menu immediately to the right of the "Use" option select "ST-Link Debugger". This selects the compiler and analysis option that supported by the Discovery.

Immediately to the right of the pulldown menu click on the "Settings" button. A window will open with three tabs at the top. In the "Debug" tab there is a pull down option labelled "Port". In this option select SW, which standards for software. This selects the connection type for the debug adapter. The Box in the top right of this tab should now show that an ARM device is selected (If "JTAG" is selected in this menu then an error should be displayed in this box). Next in the "Flash Download" tab add the "Reset and Run" option. When this option is selected the Discovery is automatically reset and run whenever new a new program is loaded onto it. If this option is not set, then after a new program is loaded onto the Discovery the reset button on the Discovery (the black bush button) needs to be pressed so that the Discovery resets and runs the program. In the "Programming Algorithm" box in the middle of the "Flash Download" tab ensure that the STM32F4xx Flash is shown and its "Device Size" is 1M. If this box is blank or the Flash is not 1M "Remove" any existing connect in this box, then click "Add" and select STM32F4xx Flash with "Device Size" 1M.

Click "OK" to close the "Debug->Settings" window and click "OK" to close the "Options for Target" window.

We have now finished setting up the Hardware and Compiler settings in the Keil µVision project environment. Our next steps are to create some files to contain our code, set some specific options for the operation of the microcontroller in one of the initialisation files and write some code for our project.

2.6. STEP 5 - Adding some source files

We now need to add some source files that will contain the code for our embedded system. In the main Keil µVision window look for the "Project" tab in the box on the left. If you expand all of the sub-folders you will see all of the files currently in your project. At the moment the only files the project contains will be under "Device" sub-folder. These are the driver and startup files for the Discovery that you have selected in the previous step. They contain all of the code (functions and definitions) that are needed to use the components on the Discovery board. The compiler will automatically add these to your project when you compile it. This means that you don't need to add any extra code in your main program to include these. You will see a mixture of different files. The ".h" files are "header" files contain things such as variable definitions. The ".c" files are C program files and mainly contain functions that can be used with the associated component. The startup file is a ".s" file and contains assembly (low level) code used to startup the microcontroller. We won't need to understand the contents of this file during this module.

You can rename both the "Source Group 1" and "Target 1" folders to better represent your project if you desire. For example, "Target 1" could become "STM32F4Discovery" since that is the Target for our compiler. Renaming is done in the same manner as a PC - i.e. slow double click on the folder name to enable the name editor.

The sources files for our project will be added to the "Source Group 1" folder. Right click on this folder and select the option to "Add new Item to … ". In the window that opens select the "C File" option in the panel on the left and type "main" into the "Name" box lower down. If you expand the "Source Group 1" folder you will now see the file "main.c". Repeat the process to create another C file called "my_defines.c" and a header file called "my_defines.h". If after you create the "my_defines.h" file it does not appear in the Source Group 1, do the following. Right click on Source Group 1 and select "Manage Project Items" and towards the bottom right of the popup window click the "Add Files" button. In the window that opens make sure that the current "Look in" directory is "Intro_lab" and change the "Files of type" option at the bottom to "Text file". The "my_defines" file of type "H File" that you have just created should show up in the list of files, select it and click "Add" and then "Close". Click "OK" in the "Manage Project Items" window and "my_defines.h" should now be displayed in the Source Group 1 folder on the left. If you need to create and/or add user files in future projects you can use these same functions.

2.7. STEP 6 - Checking the project configuration, compiling and downloading

Before we proceed further let's check that the project is setup up correctly. First open the "main.c" file by double clicking on it in the Projects tab. It will open in the window on the right and should be blank. You can edit this file by clicking the curser on the first line and by typing. On the first line type:

```
int main(void){};
```

This creates a main function in which your code will run. The "`int`" is a return argument of type integer and is required for a "`main`" function and "`void`" indicates that there is not input argument. Your future program code will go between the "`{}`". Now save this file.

Compile this project either by selecting the "Project" pulldown menu at the top and then clicking on "Build target" or use the shortcut button on the toolbar or use the shortcut key F7. This should compile and the result is shown in the "Build Output" panel at the bottom. There should be no errors and possibly one warning relating to "main.c" ending without a newline. This can be ignored for now. If errors do exist please repeat all of the steps above and if they still exists please contact one of the teaching team for assistance. If you double click on any errors listed in the "Build Output" panel you will automatically be taken to their location in your project.

Next download this compiled project to the discovery board. This is done either by selecting the "Flash" pulldown menu at the top and then clicking on "Download" or use the shortcut button on the toolbar or use the shortcut key F8. The result of the download can be observed in the "Build Output" panel. When complete your project is loaded onto the Discovery. **If you receive an error, please consult the "Common Errors" document as a starting point to resolving this.**

It should be obvious that the Discovery won't do anything (other than the power and communication LED's being active) as the "main" function is empty. In the next steps we will write a simple program to blink a single LED on and off.

2.8. STEP 7 - Configuring the processor clocks

The first step in writing our program is to configure some specific parameters of the STM32F407 microcontroller. The parameters that we are going to configure in this project relate to the processor clock speed. The microcontroller has an internal 16MHz and external 8MHz oscillator. Either of these can be used to generate a basic processor clock signal. Choice of clock speed depends on a number of factors, such as required speed and energy usage. In addition, while the internal 16MHz oscillator is twice as fast as the external oscillator, it does not provide as good a quality signal on which to base the processor clock speed, due to the mechanism through which the signal is produced. Selecting either 8 or 16 MHz as a clock speed is also quite restrictive.

Some applications might require other processor clock speeds. This need is provided on the STM32F407 by a mechanism called a Phase Locked Loop (PLL). A PLL is a simple feedback system that allows an output signal to be generated with a frequency that is a multiple of the frequency of an input signal. We are not concerned with the characteristics or implementation of this control system in this module. Our focus is on setting the parameters of the PLL to achieve a desired clock frequency.

We will set the STM32F407 to work at its maximum clock frequency of 168MHz. To achieve this clock speed we need to configure several internal registers of the STM32F407 so that it uses the internal 16MHz clock and the required parameters for the PLL. The following takes you through the code that you need to modify and add to your project to set the clock speed.

IMPORTANT: When working with digital systems and in particular when working with embedded systems, data is stored as bits (0's or 1's). We are using a 32-bit processor, so the standard data word length is 32 bits, but half-word (16-bit)s, double word (64-bits) and other variants do exist. We often need to change individual bits (0's or 1's) in a series of digital bits. Reasons why we need to do this include memory efficiency and individual bits in a series of bits might have individual important meaning. For example, processors and many other digital devices have different registers (reserved locations in memory) for all of the different functions that they incorporate. Individual bits in these registers can contain information on different operating modes, or can be used to change and configure different modes of operation.

This concept is covered in more detail in later lectures and labs. However, to manipulate individual bits in a word we usually use **bitwise logic functions** (or convenience functions which use bitwise logic functions – there are many pre-defined functions in the driver files and we will use some of these later). If you need further information on bitwise logic functions (it is important that you understand them at this stage), please refer to section 3 of the **"Bits-Bytes-Logic Functions"** additional reading document for lecture 1. The main basic logic functions and C equivalents are listed below:

Logic Function	C Function
OR	|
AND	&
NOT (or Inverse)	~
Shift bits left	<<
Shift bits right	>>

We will add this clock configuration code to the "system_stm32f4xx.c" file since this file is run on system start up and contains other code related to clock configuration. Find this file in the list of files in the Project window of Keil µVision and open it (double click). Several modifications need to be made to this file:

1. The definition of "HSE_VALUE", which is the external clock frequency, needs to be changed to match what is present on the STM32F4Discovery. As discussed above the external clock frequency is 8MHz. HSE_VALUE is defined on or near lines 69-71. The value is defined in Hz so it needs to be changed to 8000000 (it is probably currently set to 25000000).

   The HSI_VALUE (defined just below the HSE_VALUE), which is the internal clock frequency, should already be set to 16000000 as this is standard for the STM32F4 range of processors - but check to confirm this by finding the code that defines HSI_VALUE.

2. Next find where the function "SystemInit" is defined. This should start on or near line 173. This is the function that configures and turns on the system clocks.

   Locate the following lines of code, which you can find on line 192. The code that you need to add will be placed immediately after this code.

```
/* Disable all interrupts */
RCC->CIR = 0x00000000;
```

IMPORTANT: When working with binary, particularly for long lengths of bits, we usually use hexadecimal (a number system to the base-16; binary is a base-2 system and decimal a base-10 system). Hexadecimal numbers can be identified by their prefix (0x in this case, but H and h is also common). Each digit has 16 possible values from 0,…,9,A,..,F i.e. numbers 10 to 15 are given by the letters A to F. This means that each digit corresponds to 4 binary bits, so 32-bit number is given by 8 hexadecimal digits. We will use hexadecimal regularly. For more information on hexadecimal please refer to section 2.3 of the "Bits-Bytes-Logic Operations" additional reading document.

The code that you need to add is the following (you can copy and paste this code directly into the "system_stm32f4xx.c" file):

```
// Sets PLL to give 168MHz when using HSI
RCC->PLLCFGR = RCC->PLLCFGR & ~0x00007FC0; /*Un-sets all PLLN bits */
RCC->PLLCFGR = RCC->PLLCFGR + (336<<6); /* Sets PLLN to 336 */
RCC->PLLCFGR = RCC->PLLCFGR & ~0x00030000; /*Un-sets all PLLP bits */
RCC->PLLCFGR = RCC->PLLCFGR + (1<<16); /* Sets PLLP to 4 */
RCC->PLLCFGR = RCC->PLLCFGR & ~0x0000003F; /*Un-sets all PLLM bits */
RCC->PLLCFGR = RCC->PLLCFGR + 8; /* Sets PLLM to 8 or can use 8 instead of hex */

// Enables the HSE clock
RCC->CR |= RCC_CR_HSEON;

// Waits for the HSE clock to become stable and ready to use
do
{
} while(((RCC->CR & RCC_CR_HSERDY) == 0));

/* Enables high performance mode, System frequency up to 168 MHz */
RCC->APB1ENR |= RCC_APB1ENR_PWREN;
PWR->CR |= PWR_CR_PMODE;

RCC->PLLCFGR |= RCC_PLLCFGR_PLLSRC_HSI; // Selects the HSI (internal oscillator) as
the PLL source
```

```
/* Enables the main PLL */
RCC->CR |= RCC_CR_PLLON;

/* Waits until the main PLL is ready */
while((RCC->CR & RCC_CR_PLLRDY) == 0)
{
}

/* Configure Flash prefetch, Instruction cache, Data cache and wait state */
 FLASH->ACR = FLASH_ACR_ICEN |FLASH_ACR_DCEN |FLASH_ACR_LATENCY_5WS;

/* Selects the main PLL as the system clock source */
RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_SW));
RCC->CFGR |= RCC_CFGR_SW_PLL;

/* Waits until the main PLL is used as the system clock source */
while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS ) != RCC_CFGR_SWS_PLL);
{
}


RCC->CFGR |= RCC_CFGR_PPRE2_DIV2; // Sets the prescaler for the APB2 high speed clock
                                     to 2, which gives an APB2 clock frequency
                                     of 84MHz if the system clock is 168MHz

RCC->CFGR |= RCC_CFGR_PPRE1_DIV4; // Sets the prescaler for the APB1 low speed clock
                                     to 4, which gives an APB1 clock frequency
                                     of 42MHz if the system clock is 168MHz
```

We have now configured the STM32F4Discovery processor clock. Save the “system_stm32f4xx.c” file.

For further information on the PLL, how we calculate the PLL parameters and how this code presented in this step works, please refer to Appendix A. The appendix takes you through alternative steps that explains the required calculations, what this code does and how to write it. It is not crucial that you understand the PLL code in this step in detail, but if you have time to read Appendix A it will broaden your understanding and might help you later in the module.

2.9. STEP 8 - Writing a basic program

Time to write the code to blink the LED on and off. The code that we will write to do this will go in the "main.c" file. Most of this code will go between the `{}` brackets associated with the `main()` function. `{}` brackets are used in C to enclose the sub-code associated with a generic function. Other functions that use `{}` include FOR and WHILE loops.

The next points take you through the steps required to write your first program. The complete code is then presented at the end and subsequently modified to add more/better functionality. *This code and the operations used will be explained in more detail in lecture 2.*

2.9.1. Including the header files

First we need to add one line of code before the `main()` function. We are going to use some pre-defined parameters for some of the components of the STM32F407 microcontroller in a similar manner as we did in the previous step. This will make writing our code easier and it can also make it more readable. The line of code that we will add will tell the compiler where to find some of the pre-defined parameters that we are going to use in "main.c". The function that we use to do this is `#include` and the file that we want to include is the header file "stm32f4xx.h". The compiler knows that we are using the 407 variant of the STM32F4 family of microcontrollers as we have set this in step 2.

The required line of code is shown below and you should add this on line 1 of "main.c" and before the `main()` function. We normally start any by using `#include` to tell the compiler which files contain any extra information that we are using when writing our code.

```
#include "stm32f4xx.h"
```

2.9.2. Initialising the LED ports

In this intro lab we will use the green user LED. The LED is a peripheral component on the Discovery board and is connected to an external port on the microcontroller. To determine which port it is connected to we need to refer to the "STM32F407Discovery - User Manual". Checking the contents section of this user manual tells us that the LEDs are described in section 4.4 on page 16.

By reference to this section we see that the orange, green, red and blue user programmable LEDs are LED 3, 4, 5 and 6 respectively. The remaining LEDs are associated with pre-defined operations. We will only use these four user programmable LEDs in this module. Section 4.4 tells us that LED 4 is the green LED and it is connected to the "I/O PD12". This means that it is connected to pin 12 of the General Purpose Input-Output Port D. This is usually abbreviated to GPIOD pin 12. Thus to control the LED we need to toggle a single pin of GPIOD.

The first code that we need to add will initialise GPIOD. First we need to turn on the clock associated with GPIOD. All I/O ports are connected to a clock which is used, among other things, to signal bits on the pins. Information on the peripheral clocks can be found in the "STM32F - Processor Reference manual". This is a long document at 1718 pages, but is contains all of the information that is need to configure and use all of the different components of the STM32F407. It is important that you become familiar with the layout of this document as it will be used throughout this module. Learning how to configure the GPIO ports is a good introduction to understanding and using the information in this document.

Find the contents section of this document. The information about the processor clocks can be found in Chapter 6. Sections 6.3.5-6.3.19 contain the information on the peripheral clock control registers and in particular the normal mode clock enable registers are sections 6.3.10-6.3.14.

Go to the first of these (6.3.10) on page 180-182. This defines the RCC_AHB1ENR register. Looking at the table representation of the control register, you will see that the GPIO ports span the lower 11 bits. GPIOD enable is mapped to bit 3. Finding bit 3 in the list after the table, you will see that the clock for GPIOD is enabled when this bit is set on. The first line of code that we will add between the `{}` brackets of the main function will set on bit 3 in the control register RCC_AHB1ENR.

First add the following line at the start of the `main()` function on a new line in the `{}` brackets:

```
// Initialize GPIO Port for LEDs
```

This is a comment that describes what the lines of code will do. You can add comments in two ways. Either `//Comment goes here` or `/*Comment goes here*/`. ***When you write code you should always add a comment to explain what it does. This will both help you when you refer back to what you have written and help someone else understand your code. For example, if it is not clear what your code does or how it does it, you risk losing marks in assessment.***

Next add the following code on a new line after the comment. This code turns on the clock associated with GPIOD.

```
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOEN; // Enable GPIOD clock
```

The following explains the operation of this line of code:

> The control registers have been pre-defined in the main system header file "stm32f407xx.h" and have been associated with intuitive names. We can access the RCC_AHB1ENR register using the syntax `RCC->AHB1ENR`. `RCC` is a structure that contains the clocks associated with all of the peripherals and `AHB1ENR` is a sub-structure in `RCC`.
>
> In addition, we find that the bit of RCC_AHB1ENR associated with GPIOD has been predefined as `RCC_AHB1ENR_GPIOEN` - see line 5035 of the header file "stm32f407xx.h" (you can also use the search function ctrl+f to find the location of its definition in "stm32f407xx.h").
>
> In this definition on line 5035, the value on the right is the value that `RCC_AHB1ENR_GPIOEN` is defined as. `uint32_t` is a pre-defined abbreviation for `unsigned 32-bit integer`, which is the standard (unsigned) normal data type for the ARM family of microprocessors. `uint32_t` sets the data type for the definition. The actual value is the hexadecimal 0x00000008 which corresponds to 8 in decimal, which is the third bit in binary, i.e. 2^3=8 or …00001000_2. For more information on the `#define` function please refer to the C programming guide document. For more information on hexadecimal please refer to the additional reading document "Bits - Bytes - Logic operations".
>
> The |= function in the line of code that we have just written is an abbreviated form of the standard | (OR) function. It is equivalent to writing the following line of code.
>
> ```
> RCC->AHB1ENR = RCC->AHB1ENR | RCC_AHB1ENR_GPIOEN;
> ```
>
> If you need a reminder about logic functions, please refer to the C programming guide document and the additional reading document "Bits - Bytes - Logic operations" - **you will need to use these functions and hexadecimal again later in this module**.

Task 1

How would the previous line of code be written without using the pre-defined value for the bit of GPIOD that we are setting? The answer is given in section 3 of this document.

We need to add one more line of code to initialise pin 12 of GPIOD. Information on the GPIO ports can be found in the processor user manual in chapter 8 and starting on page 268.

You will see in section 8.3.3 and 8.3.4 that each GPIO has multiple control and data registers. The control registers are used to put the ports into a particular mode and to provide information on their current status. The data registers are where the value of each pin on the port is contained. In the case here they are 16-bit registers, i.e. there are 16 pins on each GPIO port.

In addition, 8.3.4 tells us that each port has both input and output registers. This makes sense since they are Input-Output ports. However, since we are only concerned with outputting data to the LED we will only use the output data register.

We need to configure one of the control registers so that pin 12 of GPIOD is in output only mode.

First we need to look at the GPIO port mode register information in section 8.4.1 on page 281. There is one of these registers for each of the GPIO ports A-K. The table shows that there are two bits associated with each of the 16 pins of the GPIO. The mode settings for pin 12 is configured using bits 25-24. The list after the table indicates that there are four possible modes of operation.

We want to use the general purpose output mode, which means the bits need to be set to 01_2, i.e. bit 24 is set to 1 and bit 25 is set to 0. As in the previous line of code the register names have been pre-defined. In this case the main pointer is GPIOD and the sub register is MODER. In addition the GPIO mode setting values have also been pre-defined in “stm32f407xx.h” on line 4277 onwards.

Task 2

Based on this information and with reference to the previous line of code that was written to turn on the clock for GPIOD, attempt to write one line of code (that comes after the previous line) that puts the GPIOD into output only mode. When you have done this check your code against the solution in section 3 of this document.

If you are not sure if your code is correct use the code provided in the solution. Then either check with one of the demonstrators to see if your code is correct or when you complete this lab replace the provided line of code with your line of code and see if the program still works as expected.

2.9.3. Toggling the green LED on

We have now written the two lines of code (the first to turn on the port clock and the second to set it into output mode). Next we just need to write a few of lines of code to toggle pin 12 of GPIOD on and off.

The function `main()` will only execute once and the program will then terminate. Therefore if we want a piece of code to repeatedly execute we need to use some type of loop function. We could use either a FOR or WHILE loop. Even if you are not familiar with C, you should be familiar with these functions form your experiences when using MATLAB or Python. In C they work in the same manner, but the syntax (i.e. the formatting of the code) is slightly different. For details on the syntax please refer to the C programming guide document - **you will need to use these functions again later in this module**.

For this program we will use a FOR loop (page 6/9 of the C programming guide document). We want this FOR loop to repeat indefinitely. To do this we leave the input expressions that determine the number of iterations (these are the values between the `()` brackets) blank.

Add the following code to you "main.c" file after the GPIOD initialisation code.

```
for(;;) {}
```

The code that we want to repeatedly execute goes between the { } brackets of the FOR loop.

IMPORTANT: To toggle the LED on and off repeatedly we need to set the bits associated with pin 12 in the output data register of GPIOD. We can directly manipulate this register, which is defined in section 8.4.6 of the "STM32F - Processor Reference manual", by writing either 1 or 0 to bit 12 to turn the LED on or off respectively. However, we would normally use a "read-modify-write" sequence so that we only change the bit that we want, while leaving all of the other bits as they were, for example using the OR (|) function. However, this requires multiple instructions to be executed by the processor, i.e. "read", then "modify" and then "write" instructions.

Alternatively we can use the GPIO port bit set/reset register (BSRR) defined in section 8.4.7. Writing 1 to the 16 least significant bits (15 to 0) of this register turns on the pins of the port and writing 1 to the 16 most significant bits (31 to 16) turns off the pins. **This register uses what is called "atomic" bit manipulation and implements the bit change in the quickest time possible on the processor and without the risk of interruption from other tasks.** "Atomic" bit manipulation is therefore usually quicker than "read-modify-write" bit manipulation. We will use this port bit set/reset register (BSRR) and in particular bit 12 to turn on pin 12 and bit 28 (12+16) to turn off pin 12.

As in the earlier code when we set up the clock and GPIOD the bit set/rest register has been pre-defined to correspond to the information of the data sheet (see the heading for 8.4.7). The main pointer is GPIOD and the sub-register is BSRR.

The following line of code can be added inside the `{}` brackets of the FOR loop to turn on the green LED connected to pin 12 of GPIOD. Here we have again use the shift left function to move the 1 twelve places to the left so that it goes in bit 12 of the register.

```
GPIOD->BSRR = 1<<12; // Turn on the green LED
```

Let's test this code to make sure that the LED turns on. The contents of your "main.c" file should be the same as given below. First compile the program (F7) and then down load it to the STM32F4Discovery (F8). The green LED should be permanently on.

```
#include "stm32f4xx.h"

int main(void){

	// Initialize GPIO Port for LEDs
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable GPIOD clock
	GPIOD->MODER |= GPIO_MODER_MODER12_0; // GPIOD pin 12 output

	for(;;){

		GPIOD->BSRR = 1<<12; // Turn on the green LED

	}

}
```

2.9.4. Toggling the green LED off

Now that you have turned on the LED, let's try to write some code to turn it off.

Task 3

Attempt to write one line of code that comes immediately after the line that turns on the LED. This new line should turn the green LED off. When you have done this check your line of code against the solution in section 3 of this document.

If you are not sure if your code is correct use the code provided in the solution. Then either check with one of the demonstrators to see if your code is correct or when you complete this lab replace the provided line of code with your line of code and see if the program still works as expected.

The contents of your "main.c" file should be the same as given after this paragraph. However, with the code that we have written, the green LED will be toggled on and off very quickly since the processor is running at 168MHz. Each oscillation of the clock takes 5.95ns and since simple code like this will only take a few clock cycles to execute, pin 12 of GPIOD will be turned on and off quicker than the LED can respond. If you compile this program (F7) and then down load it to the STM32F4Discovery (F8), the green LED should now appear to be permanently off.

```
#include "stm32f4xx.h"

int main(void){

	// Initialize GPIO Port for LEDs
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable GPIOD clock
	GPIOD->MODER |= GPIO_MODER_MODER12_0; // GPIOD pin 12 output

	for(;;){

		GPIOD->BSRR = 1<<12; // Turn on the green LED

		GPIOD->BSRR = 1<<(12+16); // Turn off the green LED


	}

}
```

2.9.5. Using a delay

What we need to do is insert a delay both after we turn the LED on and after we turn the LED off. There are multiple ways that we can implement delays in an embedded system. Possibly the most straightforward method and the one which doesn't require any other components is to use a FOR loop that doesn't contain any code and which executes a fixed number of times.

The first step in implementing this for loop is to define a control variable which we call `ii`. This is the control variable that the FOR loop will use to implement the number of iterations. Any variable definitions need to appear at the start of the `main()` function and before any executable code. Therefore in our example this definition should appear after the opening `{` of the `main()` function and before the lines of code to initialise GPIO port. The code required is given below.

```
uint32_t ii;
```

This code defines `ii` as an unsigned 32-bit number. We could define `ii` with an actual value, for example 10, by replacing `ii` with `ii=10`. However, for a FOR loop we don't need to assign `ii` a value, as we specify its initial value in the FOR loop.

Finally we need to insert the FOR loop into the code at the correct locations. First we decide how many times we want the FOR loop to execute. Let's say we want the LED to turn on for 1s and turn off for 1s. We know that we have set the system clock to 168MHz. If in addition we know that an empty FOR loop takes an average of 6.5 clock cycles for each iteration, we can work out the number of iterations from equation (1).

$$Number\ of\ iterations = Delay\ (seconds)/(FOR\ loop\ average\ number\ of\ clock\ cycles \times Clock\ period) \quad (1)$$

$$Number\ of\ iterations = {}^{1}\!/_{6.5 \times \frac{1}{168000000}} = 26000000 \text{ (rounded to 2 significant figures)}$$

Therefore we need 26 million iterations of the FOR loop to achieve a delay of 1 second.

Task 4

With this knowledge (use of ii as the control variable and a FOR loop that iterates 26000000 times) together with the information on page 6/9 (page 9 of chapter 6) of the C programming guide, attempt to write a FOR loop that achieves a 1 second delay. You should place one copy of the FOR loop immediately after the code to switch on the LED and a second copy immediately after the code to switch off the LED. The code for the FOR loop is given in the solutions in section 3 of this document.

The contents of your "main.c" file should now be the same as given after this paragraph.

```
#include "stm32f4xx.h"

int main(void){

	uint32_t ii;

	// Initialize GPIO Port for LEDs
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable GPIOD clock
	GPIOD->MODER |= GPIO_MODER_MODER12_0; // GPIOD pin 12 output

	for(;;){

		GPIOD->BSRR = 1<<12; // Turn on the green LED

		for(ii=0;ii<26000000;ii++){} //FOR loop to implement a 1 second
					     delay with a clock speed of 168MHz

		GPIOD->BSRR = 1<<(12+16); // Turn off the green LED

		for(ii=0;ii<26000000;ii++){} //FOR loop to implement a 1 second
					     delay with a clock speed of 168MHz

	}

}
```

Compile this program (F7) and download it to the STM32F4Disocovery. The green LED should now be repeatedly blinking on for 1 second and off for 1 second. To check the delay time (for example using a stop watch) measure the time taken for the LED to go through 5 complete cycles of on and off. You should get a time that is very close to 10 seconds (the error should be +/- 0.5 seconds and is mainly due to human measurement error) indicating a delay of 1 second.

Task 5

With the FOR loop the delay depends on the clock speed. For this project we initially chose to use the internal clock, which together with the PLL gave a clock speed of 168MHz. This was set in step 7, point 2 using the following line of code.

```
RCC->PLLCFGR |= RCC_PLLCFGR_PLLSRC_HSI; // Selects the HSI (internal
oscillator) as the PLL source
```

Find this line of code in the “system_stm32fxxx.c" file and modify it so that the clock source for the PLL is now the external oscillator. Compile and download this program and then observe and explain what happens to the green LED. The answer is given in section 3 of this document.

Task 6

Modify the code written for main() in step 8 so that instead of the green LED, it is the orange LED that blinks on and off. You will need to use the STMF4Discovery kit user manual to find out what port and pin the orange LED is connected to and then change your code accordingly. When you have completed your modification compile and download the program to observe its behaviour. The answer is given in section 3 of this document.

Congratulations, you have now completed your first project in embedded systems. While this is a simple example it introduces some important concepts associated with embedded systems development. One of the most important is delays and timing because embedded systems are usually required to control some device in the real world. In this example it was the green LED.

However, a problem with using a FOR loop is that it is a little imprecise in achieving a consistently accurate delay. This is because it is based on knowing the average number of cycles of that it takes for each iteration of the FOR loop and the clock speed of the system. In addition, this LED program is very simple. In more complex systems there is much more uncertainty on the operation of the microcontroller due to the use of more of its resources and interactions form the environment.

Therefore, a FOR loop is not generally advisable if you need to create an accurate and reliable delay. Particularly when you consider large number of embedded systems need to implement tasks with "hard" time constraints, such as controlling the valve timing in an engine, where even small delays can lead to complete failure of the system. In later labs we will look at more accurate and reliable ways of implementing delays and achieving other timing constraints.

3. Task solutions

Task 1

As described in the main section the enable bit for GPIOD is mapped to bit 3 of the `AHB1ENR` register. Therefore either of the following two solutions would work (note that the compact |= form is used).

```
RCC->AHB1ENR |= 1<<3;
```

or

```
RCC->AHB1ENR |= 0x00000008;
```

Task 2

You can see I section 8.4.1 of the processor reference manual that the reset state for the two bits is 00. So we can assume that this is the value of the two bits when the system first turns on or when the reset button (black button) is pressed. The following line of code sets on the higher of the two bits (bit 25) while leaving the lower of the two bits (bit 24) unchanged to give a value of 10_2 for pin 12 of the mode register of GPIOD.

```
GPIOD->MODER |= GPIO_MODER_MODER12_0; // GPIOD pin 12 output
```

Task 3

The reset bits are in the top 16 most significant bits of the register so we can take the code used to set on the bit and offset the location by 16. This is the same as setting on bit 28 which corresponds to the reset bit for pin 12.

```
GPIOD->BSRR = 1<<(12+16); // Turn off the green LED
```

Task 4

The code for the FOR loop is given below. The parameters for the control variable `ii` are given inside the ( ) brackets. The first part sets the initial value for the control variable to 0. The second part tells the FOR loop to repeat while the control variable is less than 26000000. The third part tells the FOR loop to increment the control variable by one during each iteration of the FOR loop. You can also have FOR loops that decrement the control variable, i.e. `ii--`. The contents of the FOR loop are left blank, i.e. { }.

```
for(ii=0;ii<26000000;ii++){} // FOR loop to implement a 1 second delay with
                             a clock speed of 168MHz
```

Task 5

The code from step 7 point 5 becomes:

```
RCC->PLLCFGR |= RCC_PLLCFGR_PLLSRC_HSE; // Selects the HSE (internal
oscillator) as the PLL source
```

Then since the external oscillator is 8MHz, which is half the speed of then internal oscillator, the PLL output is halved, i.e. it is 84MHz. This can be explicitly calculated from equations (1) and (2) since the PLL parameters are unchanged. Therefore since the system clock is now half the speed, instructions will generally take twice as long to execute. Therefore the delay implemented by the FOR loop will be doubled to 2 seconds. This can be explicitly calculated by putting the new system clock speed into equation (3) with all other parameters remaining the same.

Task 6

By checking section 4.4 on page 16 of the data sheet, we see that the orange user LED is connected to pin 13 of GPIOD. If we check the processor reference manual then we also find that pin 13 corresponds to bits 26 and 17 of the mode register MODER and bits 13 (set) and 29 (reset) of the bit/set register BSSR. Therefore we need to modify three lines of code in the main() function. The first is to set GPIOD 13 into output only mode, the second is to set on pin 13 of GPIOD and the third is to turn off pin 13 of GPIOD. These three lines of code are in bold italic in the solution for “main.c” given below.

```
#include "stm32f4xx.h"

int main(void){

	// Initialize GPIO Port for LEDs
	RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN; // Enable GPIOD clock
	GPIOD->MODER |= GPIO_MODER_MODER13_0; // GPIOD pin 13 output

	for(;;){

		GPIOD->BSRR = 1<<13; // Turn on the orange LED

		for(ii=0;ii<26000000;ii++){} //FOR loop to implement a 1 second
		                             delay with a clock speed of 168MHz

		GPIOD->BSRR = 1<<(13+16); // Turn off the orange LED

		for(ii=0;ii<26000000;ii++){} //FOR loop to implement a 1 second
		                             delay with a clock speed of 168MHz

	}

}
```

4. Appendix A – Setting the processor clock: Alternative steps with a more detailed explanation

Information on the parameters that need to be set can be found in the STM32F4 processors reference manual. This is a long document at 1718 pages, but is contains all of the information that is need to configure and use all of the different components of the STM32F407. It is important that you become familiar with the layout of this document as it will be used throughout this module.

Find the contents section of this document. The information about the processor clock characteristics, including the internal and external oscillators and PLL can be found in Chapter 6. The PLL is described on page 155-156 and the information on pages 161-164 detail the internal registers on the STM32F407 that need to be configured. Details about how to achieve a specific clock frequency using the PLL is covered at the start of section 6.3.2 on page 163. There are three parameters that need to be set. These are PLLN, PLLM, PLLP and PLLQ. The first three of these are used to set the processor clock frequency (this is "PLL general clock output" frequency in the document) as shown in equation (A1) and (A2).

$$f_{VCO\ clock} = f_{PLL\ clock\ input} \times \frac{PLLN}{PLLM} \quad \text{(A1)}$$

$$f_{PLL\ general\ clock\ output} = \frac{f_{VCO\ clock}}{PLLP} \quad \text{(A2)}$$

PLLQ is used to create an additional clock speed to run some peripheral components. This additional clock speed, together with some other additional clocks present on the STM32F407 will be ignored in this introductory lab and the associated parameters will be left at their default values.

To generate a require clock speed we need to select appropriate values for the parameters in equation (A1) and (A2). In addition we also need to select a signal for the "PLL clock input". This can be either the internal or external oscillator. Using the internal oscillator the highest clock speed that can be generated is 168MHz and with the external clock it is 84MHz. We will start by using the internal clock as the "PLL clock input" and we will select the parameters to achieve the maximum clock speed of 168MHz. General rules for selecting these parameters is provided in section 6.3.2. The parameters that we will use are PLLN = 336, PLLM = 8 and PLLP = 4. Substituting these into equations (A1) and (A2), together with 16MHz for the "PLL clock input" leads to a "PLL general clock output" of 168MHz.

To achieve this clock speed we need to configure several internal registers of the STM32F407. We could write some of this code in the main() function we created in the first step. However, it is better to modify the "system_stm32f4xx.c" file since this file is run on systems start up and contains other code related to clock configuration. Several modifications need to be made to this file:

1. The definition of "HSE_VALUE" needs to be changed to match what is present on the STM32F4Discovery. As discussed above the external clock frequency is 8MHz. HSE_VALUE is defined on or near lines 69-71. The value is defined in Hz so it needs to be changed to 8000000 (it is probably currently set to 25000000).

2. The HSI_VALUE should already be set to 16000000 as this is standard for the STM32F4 range of processors - but check to confirm this by finding the code that defines HSI_VALUE.

3. Next find where the function "SystemInit" is defined. This should start on or near line 173. This is the function that configures and turns on the system clocks. We are going to add some code that sets the correct values for PLLN, PLLM and PLLP into the RCC PLL configuration register that is defined in section 6.3.2 of the processor reference manual. First we need to work out what bits we need to set in thus register. If we look at section

6.3.2 we see that the 32-bit register is shown in table format just below the equations used to calculate the clock speed. You will see that it is a 32-bit register, which is standard for the ARM family of microprocessors. The first row shows bits 31 to 16 and the second row shows bits 15 to 0, i.e. the most significant bit (MSB) is on the left. After this table the bits and their associated properties are listed. From this list we see that bits 17 to 16 define the PLLP, 14 to 6 the PLLN and 5 to 0 the PLLM. From this list we also see that to get the required parameters these bits need to be set as follows:

Bits 17:16 set to 01_2 (i.e. 1 in decimal) for PLLP=4
Bits 14:6 set to 101010000_2 (i.e. 336 in decimal) for PLLN=336
Bits 5:0 set to 001000_2 (i.e. 8 in decimal) for PLLM=8

4. There are numerous different ways of writing code to achieve this. The method that we will use will address each of the different parameters separately and first set all of the associated bits to 0 and then set on the bits that we want. This is a robust way and by addressing each parameter separately it allows the individual parameters to be easily changed at later time if needed.

   Find where the function "SystemInit" is defined. This should start on or near line 173 of the "system_stm32f4xx.c" file. This is the function that configures and turns on the system clocks. Locate the following lines of code, which you can find on line 192. The code that you need to add will be placed immediately after this code.

```
/* Disable all interrupts */
RCC->CIR = 0x00000000;
```

   The code to set the PLLN is given below. To fully understand this code you will need to refer to the additional reading document "Bits - Bytes - Logic operations" and the C Programming guide for details of binary, hexadecimal and the operation of bitwise logic functions.

```
// Sets PLL to give 168MHz when using HSI
RCC->PLLCFGR = RCC->PLLCFGR & ~0x00007FC0;
RCC->PLLCFGR = RCC->PLLCFGR + (336<<6); /* Sets PLLN to 336 */
```

   RCC is a pointer to the reset and clock control registers starting address in the processor memory and PLLCFGR is the sub-register for the PLL configuration register in that address space. These symbolic names have been defined in the header file "stm32f407xx.h" (if you open the expansion in the file list for system_stm32f4xx.c you will find this file) and allow the registers to be accessed in an intuitive manner without the need to know the actual address of the register each time we need to use it. The first of these two lines uses the & (bitwise AND) and ~ (bitwise NOT) functions to unset bits 6 to 14 while leaving the remaining bits of PLLCFGR unchanged. 0x00007FC0 is the hexadecimal for bits 6 to 14. The second line sets the desired bits in PLLCFGR. It uses the shift left function which takes the desired value for bits 6 to 14 and then shifts it by 6 binary places to the left so that it is in the correct location in the register (i.e. bits 6 to 14). In other words 336 is $00000000000000000000101010000_2$ in 32-bit form, which shifted 6 places to the left gives $00000000000000001010100000000000_2$. This values is then added to the existing value in PLLCFGR so that only the desired bits are turned on.

Task A1

Attempt to write the next lines of code that set the PLLP and PLLN values. The correct code is provided in the list of solutions at the end of this document.

5. Next we need to turn on the external oscillator (we will use this later in the introductory lab) and enable the high speed mode. The code that does this is shown below. Copy this code and paste it into the "SystemInit" function just below the code that you have just added.

```
// Enables the HSE clock
RCC->CR |= RCC_CR_HSEON;

// Waits for the HSE clock to become stable and ready to use
do
{
} while(((RCC->CR & RCC_CR_HSERDY) == 0));

/* Enables high performance mode, System frequency up to 168 MHz */
RCC->APB1ENR |= RCC_APB1ENR_PWREN;
PWR->CR |= PWR_CR_PMODE;
```

6. Next we need to select the oscillator source for the PLL. As we want the internal clock so that we achieve the maximum 186MHz clock speed the following line of code is need after the code in step 4.

```
RCC->PLLCFGR |= RCC_PLLCFGR_PLLSRC_HSI; // Selects the HSI (internal
oscillator) as the PLL source
```

7. Next we need to turn on the PLL. This is achieved using the following code which comes just after the code in step 5.

```
/* Enables the main PLL */
RCC->CR |= RCC_CR_PLLON;

/* Waits until the main PLL is ready */
while((RCC->CR & RCC_CR_PLLRDY) == 0)
{
}

/* Configure Flash prefetch, Instruction cache, Data cache and wait
state */
FLASH->ACR = FLASH_ACR_ICEN |FLASH_ACR_DCEN |FLASH_ACR_LATENCY_5WS;

/* Selects the main PLL as the system clock source */
RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_SW));
RCC->CFGR |= RCC_CFGR_SW_PLL;

/* Waits until the main PLL is used as the system clock source */
while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS ) != RCC_CFGR_SWS_PLL);
{
}
```

8. Finally we set the high and low speed peripheral clocks (these will be used in later labs) as follows:

```
RCC->CFGR |= RCC_CFGR_PPRE2_DIV2; // Sets the prescaler for the APB2
                                  high speed clock to 2, which gives
```

```
                                        an APB1 clock frequency of 84MHz if
                                        the system clock is 168MHz
    RCC->CFGR |= RCC_CFGR_PPRE1_DIV4; // Sets the prescaler for the APB1
                                        low speed clock to 4, which gives
                                        an APB1 clock frequency of 42MHz if
                                        the system clock is 168MHz
```

Task A2

In point 4-6 we have used pre-defined parameters for the bits that we want to manipulate (turn on and off). For example `RCC_CR_HSEON` is the bit to turn the external crystal oscillator on and off - by cross-reference to the processor reference manual we see that this is bit 16 in the RCC clock control register defined in section 6.3.1. `RCC_CR_HSEON` is defined on line 4808 of the header file "stm32f407xx.h" (you can also use the search function ctrl+f to find the location of its definition in stm32f407xx.h). In step 3 when stetting the PLL values, we manually selected the bits to turn on and off and used the shift left function place them in the correct position in the register. Now modify the code from step 3 so that the symbolic PLL definitions found in the "stm32f407xx.h" file (line 4818 onwards) are used instead. The solution is given at the end of this document.

Solutions

Task A1

The first line sets all of the bits associated with PLLP to zero and the second line sets the desired bits of PPLP on. The third and fourth lines repeat this process for PLLM.

```
RCC->PLLCFGR = RCC->PLLCFGR & ~0x00030000;
RCC->PLLCFGR = RCC->PLLCFGR + (1<<16); /* Sets PLLP to 4 */
RCC->PLLCFGR = RCC->PLLCFGR & ~0x0000003F;
RCC->PLLCFGR = RCC->PLLCFGR + 8; /* Sets PLLM to 8 or can use 8 instead of
hex */
```

Task A2

The key here is to note that the naming of the parameters are defined with reference to each specific bit for the appropriate PLL parameter, but the numeric value is the location of these bits in the PLLCFGR register. So far example `RCC_PLLCFGR_PLLN_0` is the first bit for the PLLN parameter, but it is defined as the numeric value 0x00000040, which is the bit 6 of the PLLCFGR. Therefore the correct solution to this task is as given below. Note that the symbolic parameter with no number attached is defined as all of the related PLL bits, i.e. `RCC_PLLCFGR_PLLN` is all of the PLLN bits in PLLCFGR.

```
// Sets PLL to give 168MHz when using HSI
RCC->PLLCFGR = RCC->PLLCFGR & ~RCC_PLLCFGR_PLLN;
RCC->PLLCFGR = RCC->PLLCFGR + (RCC_PLLCFGR_PLLN_8 + RCC_PLLCFGR_PLLN_6 +
               RCC_PLLCFGR_PLLN_4); /* Sets PLLN to 336 */
RCC->PLLCFGR = RCC->PLLCFGR & ~RCC_PLLCFGR_PLLP;
RCC->PLLCFGR = RCC->PLLCFGR + RCC_PLLCFGR_PLLP_0; /* Sets PLLP to 4 */
RCC->PLLCFGR = RCC->PLLCFGR & ~RCC_PLLCFGR_PLLM;
RCC->PLLCFGR = RCC->PLLCFGR + RCC_PLLCFGR_PLLM_3; /* Sets PLLM to 8 or can
use 8 instead of hex */
```