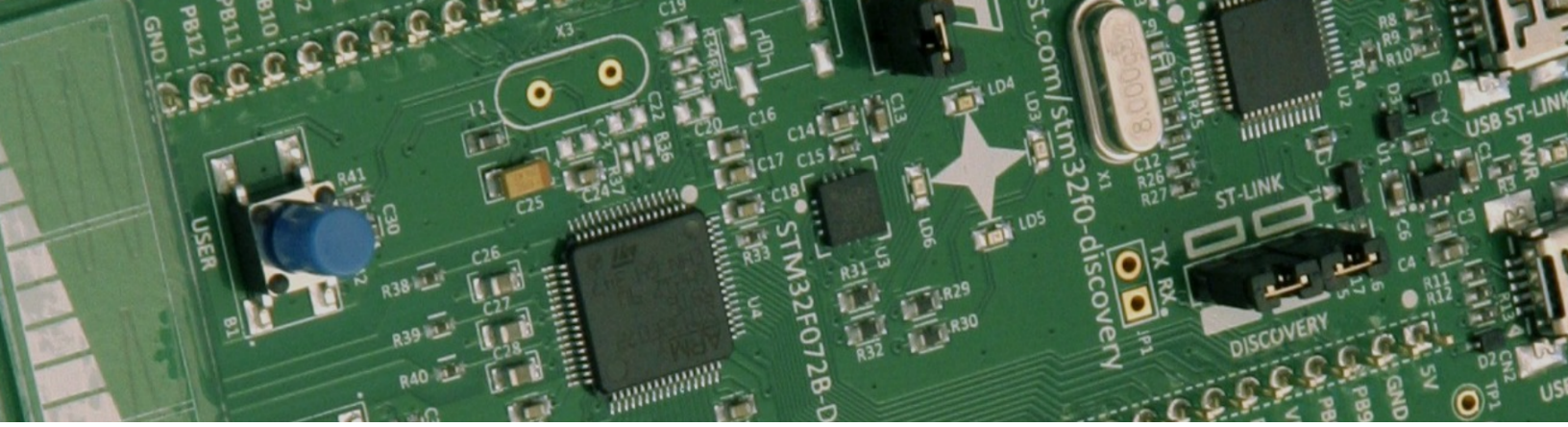


Contents

0. The STM32F072 Discovery and Toolchain

INTRO 0.1 Introduction to the STM32F072 Discovery Board

This lab introduces the STM32F072 Discovery board from ST Microelectronics and guides through the creation of a simple application. The exercises in this lab form the basic starting point for all future activities. After completing this lab, you will understand the basic operation and structure of an MDK:ARM code project and be able to use ST provided utilities to generate a pre-configured project template.

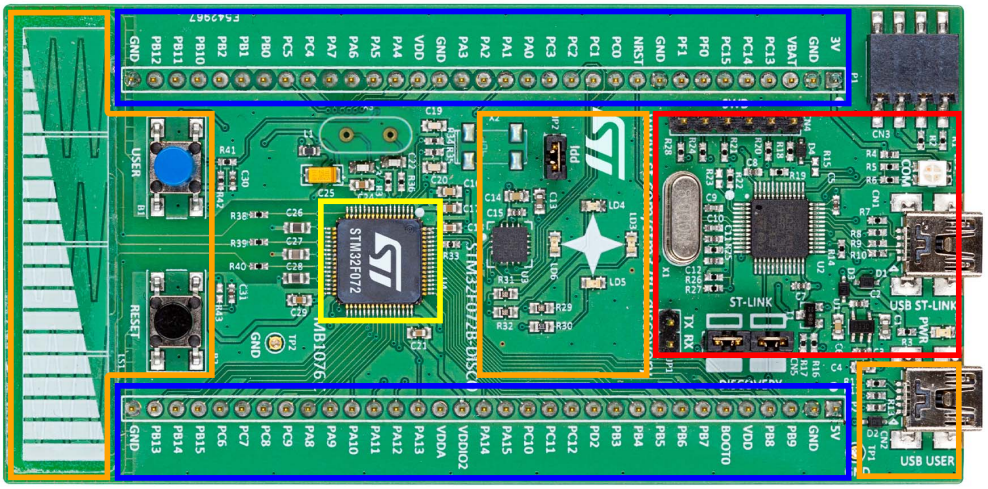


Figure 1: STM32F072 Discovery Board from ST Microelectronics

One unique aspect of embedded systems programming is that the generated code typically does not run on the system used to develop the software; instead, the compiled binary is loaded onto an external device for testing and debugging. You'll be using a generic test board containing a processor and peripherals to target during development.

The STM32F072 Discovery board is a low-cost environment which provides all of the hardware you need to start developing basic projects. There are four main parts of the Discovery board:

STM32F072R8 ARM Cortex-M0 Processor (YELLOW)

The STM32F072R8 features an ARM Cortex-M0 RISC core that can operate up to 48 MHz clock frequency. It has 16 Kbytes of static RAM and 128 Kbytes of Flash memory for program and data storage. It operates on voltages between 2.0 and 3.6 V in a 64-pin low-quad-flat-pack (LQFP) package. Typically these labs will use STM32F0 as a shorter name for the STM32F072R8.

The STM32F072R8 processor's name can be broken down into a couple of different parts. These parts identify the processor family to which it belongs, its storage capabilities, and chip package which the device possesses.

- **STM32F0** – Identifies the chip as a Cortex-M0 produced by ST Microelectronics
- **72** – Specifies the device as part of the STM32F0x2 sub-family; each sub-family has minor differences in design and available peripherals.
- **R8** – Indicates the FLASH (program storage) capacity and number of pins on the chip.

Figure 2 on the following page shows the entire STM32F0x2 (USB line) device sub-family. This figure demonstrates that there are multiple groupings of STM32F0 devices, each featuring different capabilities and containing a variety of parts.

ST-Link Chip Programmer and Debugger (RED)

Unlike developing on many traditional systems, the STM32F0 cannot communicate directly with the machine that compiles the code; therefore, multiple methods of transferring a compiled binary to the STM32F0 are available (some can communicate directly), but the most robust method is a hardware debugger.

Typically many device manufacturers prefer to develop their own debugging hardware. ST Microelectronics' self-developed and preferred system is the ST-Link: the ST-Link is itself just another ARM processor, programmed to connect via USB to the host machine. On the host machine, a number of programs can connect to the ST-Link and download a binary to the target device. The protocol between the ST-Link and the STM32F0 is *serial wire debug* (SWD). We will discuss the details of this protocol in later chapters.

Processor Breakout Pins (BLUE)

The Discovery board "breaks-out", or exposes, all of the STM32F0 processor pins on the two side connectors: these pins enable connecting additional circuits to the Discovery board. The STM32F0 can control most of the pins that are available. However, some pins handle system functions, such as debugging, which the user should not use for other purposes.

Figure 2: Entire STM32F0x2 device sub-family

Experimental Hardware (ORANGE)

Although the STM32F072 Discovery development board is simple in comparison to other platforms, it offers several devices for experimentation.

- Light emitting diodes (LEDs) in Green, Orange, Blue, and Red
- "USER" button for general purpose use
- "RESET" button
- Linear capacitive-touch sense bar
- L3GD20 3-axis MEMS Gyroscope
- User programmable mini-USB 2.0 port

INTRO 0.2 ARM Development Toolchain

A *toolchain* is a collection of software and hardware tools that create a complete system for developing, loading, and debugging embedded applications.

Figure 3 shows the components of the Kiel MDK:ARM toolchain; these provide a foundation for software projects and generate compiled ARM binaries. The toolchain consists of two major components, the MDK tools, and supporting software packs.

MDK Tools

These tools form the basis of the Kiel MDK:ARM system. The major portion that we will use is the µVision IDE, which targets ARM Cortex-M devices. Although not utilized in this course, the DS-MDK tools target combination processors with both application-class (Cortex-A) and embedded (Cortex-M) systems. Both IDE systems use the ARM C/C++ cross-compiler, which generates ARM/THUMB2 assembly language code.

A cross-compiler describes a system which generates assembly languages differing from the machine on which it operates. Although the MDK:ARM system runs on a conventional PC x86-x64 architecture, it generates ARM/THUMB2 codes compatible with the Cortex-M0 STM32F0 processor.

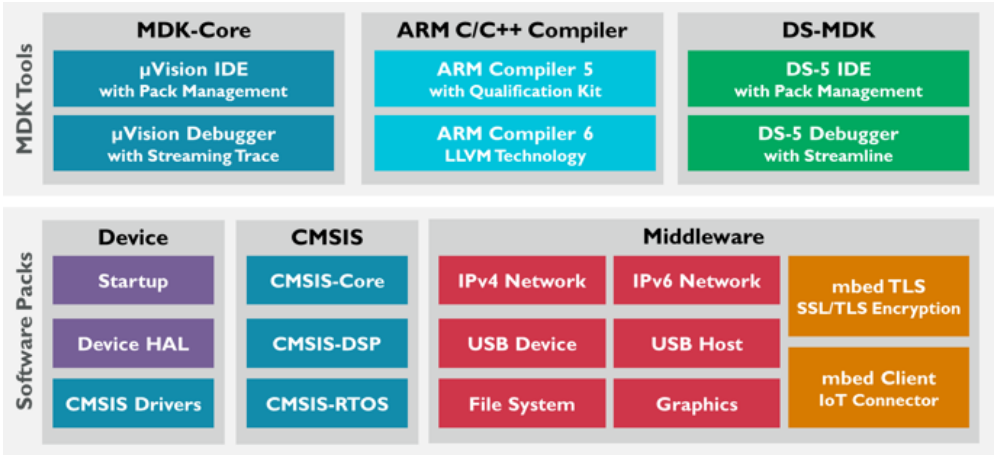


Figure 3: MDK:ARM Toolchain

Software Packs

Similar to many conventional programming environments, embedded systems rely on supporting software libraries; we describe several such libraries below.

Startup Libraries – The STM32F0 processor requires a small amount of chip-specific initialization code to prepare the memory system and load the user application.

Device HAL – STM32F0 devices have a large number of peripherals for tasks such as timing and communication; ST Microelectronics provides files which define the bindings between these and the user application. The *hardware abstraction library* (HAL), provides a C/C++ API for controlling these peripherals.

CMSIS Libraries – The ARM Cortex-M0 core also contains a number of advanced features; these core-peripherals are universal across all devices based on the same ARM core. The *cortex microcontroller software interface standard* (CMSIS), provides a library API for interfacing with these features.

Middleware – Middleware libraries provide support for higher-level functions such as file systems, USB, and graphics.

EXERCISE 0.3 Creating Your First Project

Although we will be using Kiel MDK (µVision) to develop the software for the exercises in this lab, manually creating a project for an STM32F0 processor requires significant configuration. To bypass this, we will use the STM32CubeMX utility to graphically configure the project parameters and generate a ready-to-use µVision project.

0.3.1 Generating a Project in STM32CubeMX

Selecting the Correct Processor

Launch the **STM32CubeMX (STMCube)** utility either from the start menu entry or desktop icon. Once loaded, select the **New Project** link on the welcome page, or use **File Menu →New Project**.

The *New Project* window should open—here, select the particular STM32 device for which STMCube will generate a project. The processor in use on the STM32F072 Discovery board is the STM32F072R8. You can quickly narrow down the available selections by selecting the following criteria in the *MCU Filters* section.

- Select **STM32F0** in the *Series* filter
- Select **STM32F0x2** in the *Lines* filter
- Select **LQFP64** in the *Package* filter

At this point, there should only be a few choices available, select **STM32F072R8Tx** and press the **OK** button. Figure 4 on the next page shows the new project window with the appropriate filters applied.

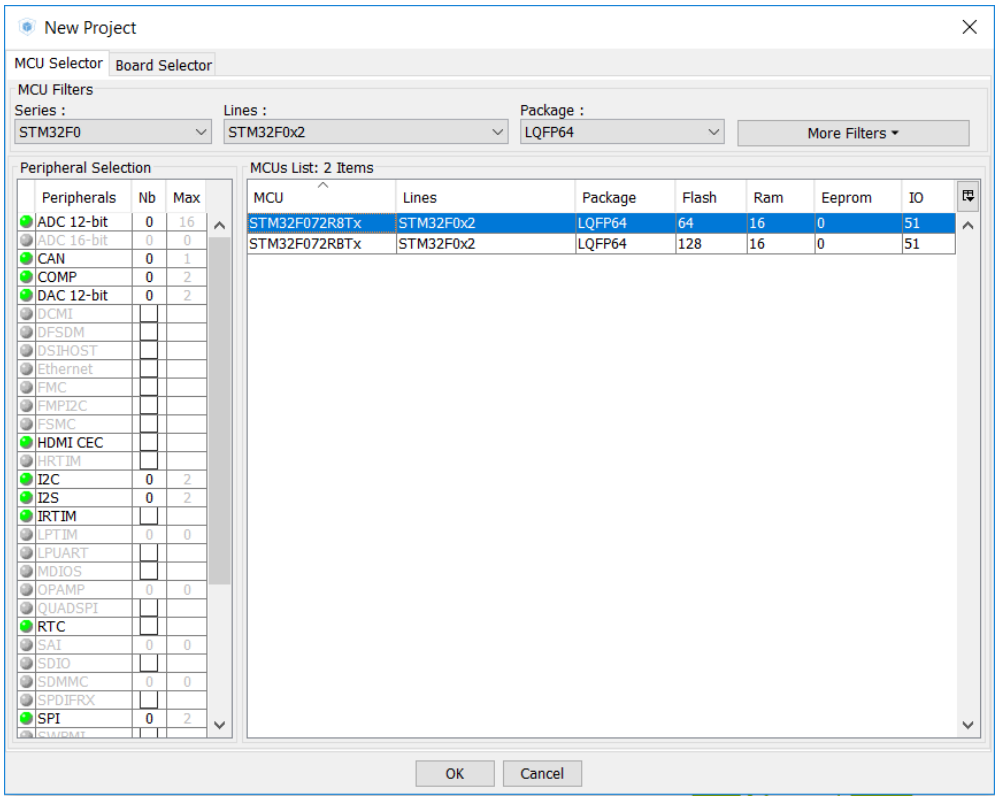


Figure 4: STM32CubeMX new project window with filters applied

Changing Project Generation Settings

After selecting the processor, you see the *Pinout* screen of STMCube. (Figure 5 on the following page) This screen shows all of the available pins on the selected device and can enable and configure many of the internal peripherals within the STM32F0 processor. Although we won't be using any of the features here for this exercise, the pinout tool is very helpful for planning what pins can be used for peripheral outputs. Feel free to explore and enable peripherals on the left pane of the window but be sure to clear them all before moving on. We will be using the real capabilities of STMCube in later labs.

Before we can tell STMCube to generate our template code project, we need to configure a few settings so that it creates the proper files and structure that µVision expects. To do this use **Project Menu →Settings**.

Within the settings dialog shown in Figure 6 on page 10 you will need to configure the following attributes:

- Name the new project
- Select a directory where STMCube can create subfolders to store project files.
- Change the *Toolchain/IDE* dropdown menu to **MDK-ARM V5**.

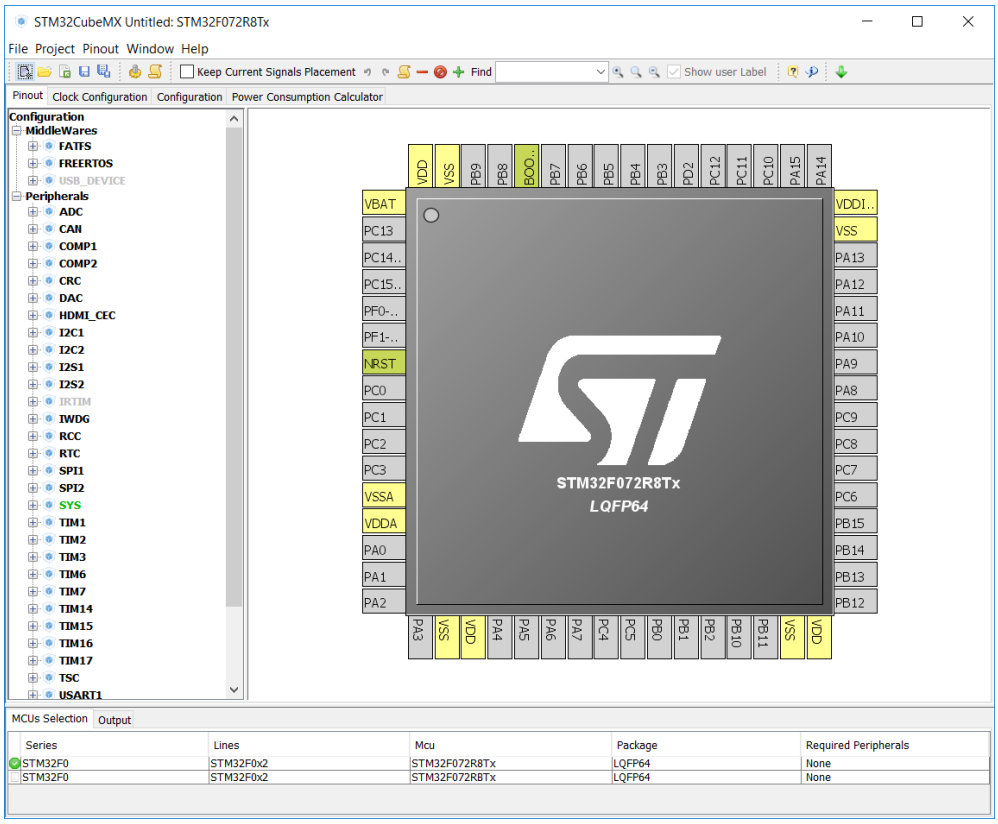


Figure 5: STM32CubeMX Pinout view

STMCube supports a broad range of toolchains and development environments. Unfortunately, many of these are completely incompatible with each other. The Toolchain/IDE menu selects between the supported output environments and ensures that the generated project will be compatible with the IDE you are using.

Once you are satisfied with your settings on the *Project* tab, move to the *Code Generator* tab at the top of the window. Although these settings are not necessarily important at the moment, you can reduce the code size of your project dramatically by selecting the **copy only the necessary library files** option in the *STM32Cube Firmware Library Package* menu.

When the necessary project configurations have been set, close the settings dialog with the **OK** button.

Don't forget to set the proper toolchain in the project settings! Otherwise, it will be impossible to use the generated project files with µVision.

Generating the Project

Since we are not interested in configuring peripherals at the moment, generate the finished project using **Project Menu →Generate Code**.

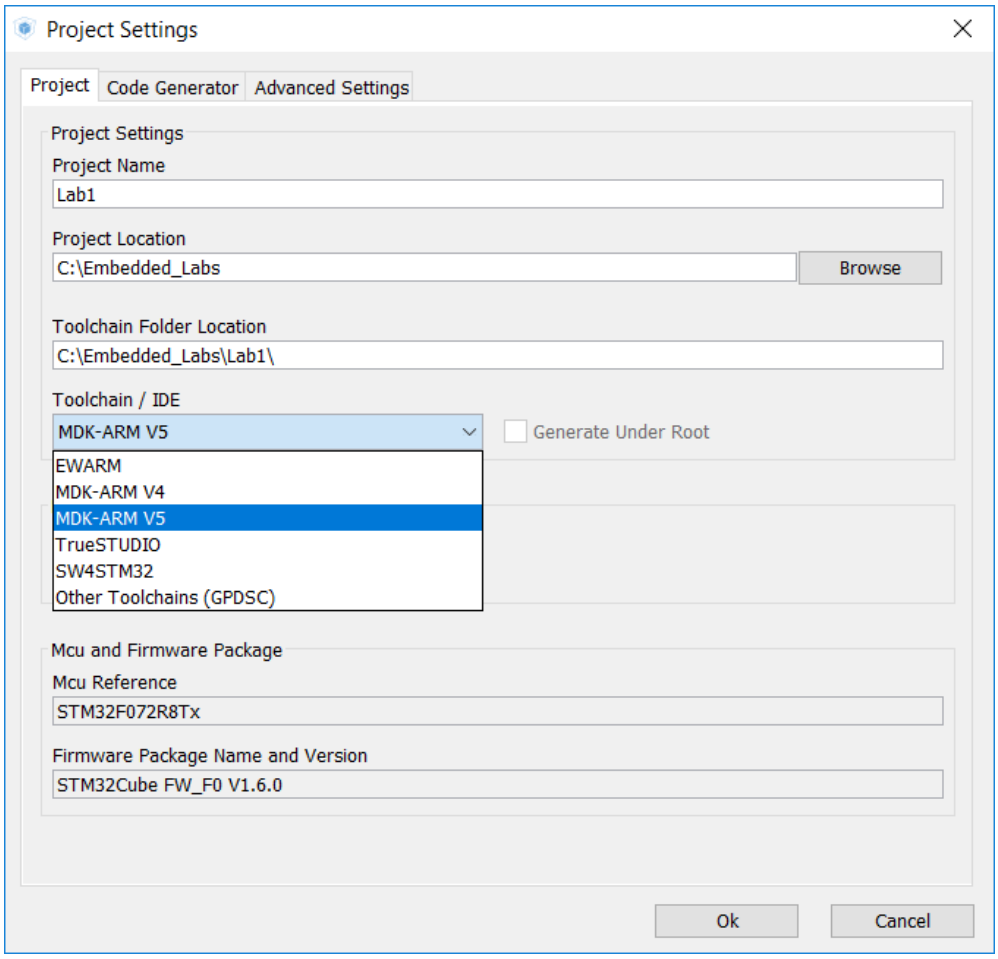


Figure 6: STM32CubeMX Project Settings

STMCube may take a while to copy the files to the directory specified in the settings. Afterward, you may be asked if you want to open the project folder or project file itself. You can select either of these options; we will be exploring both of them in the next section.

0.3.2 Exploring the Project in µVision

Now that we have a project template that is ready to use, it is time to launch **Kiel µVision** from either the start menu or desktop icon and explore the development environment.

Once µVision has loaded, open the newly generated project by selecting the ".uvprojx" file located under the *MDK-ARM* folder of the project directory. Alternatively, STMCube will also automatically launch µVision if the option to open the project was chosen after code generation.

The advantage of using STMCube to generate the project is that no further configuration within µVision is required to start developing software.

After the project has loaded, expand the folders within the *Project* window (left-hand) of the main screen, you should see a folder hierarchy similar to Figure 7 on the next page.

Application/MDK-ARM

This folder contains assembly-language startup code that initializes the STM32F0 processor and starts the user application after reset. These operations depend on the particular memory map for the processor and are unique to each STM32F0 sub-family. These files are provided by ST Microelectronics and customized for Kiel's toolchain.

Application/User

This folder contains all user-level application code. You can find the *main.c* file here, as well as any, included header files.

If you open the physical project directory, you will find that µVision places c/cpp files within the *Src* folder and h/hpp files within the *Inc* folder.

µVision chooses to list header files as sub-elements of the code that reference them. Because this project hasn't been compiled yet, you will be unable to see any of the included header files within the IDE.

Drivers/STM32F0xx_HAL_Driver

The STM32F0 devices are much more than a standalone processor core. Surrounding the ARM Cortex-M0 is a wide variety of peripheral circuits which connect the processor to the outside world. Similarly to the startup code, ST Microelectronics provides a hardware abstraction library. (HAL) These libraries allow the user to control system peripherals via a C-language API.

Drivers/CMSIS

Similar to the ST hardware abstraction library, ARM publishes a software API for the internal features of the Cortex-M0 core. These functions known as the Cortex Microcontroller Software Interface Standard, (CMSIS) provide a universal interface to all ARM devices regardless of the specific chip manufacturer.

0.3.3 Compiling and Loading a Demo Program

Auto-Generated Elements in Main.c

Open the *main.c* located within the Application/User folder. You will notice that the program creates several functions, including the `main()` function. The additional functions initialize the processor clock to the selected speed and source when STMCube generated the project.

Most embedded processors have flexible clock options that allow them to run from both internal or external clock signals to operate over a wide range of frequencies. The default processor speed that STMCube configures is generally 8 MHz. If you open the STMCube files in the project directory, you can select the Clock Configuration tab and view the clock settings.

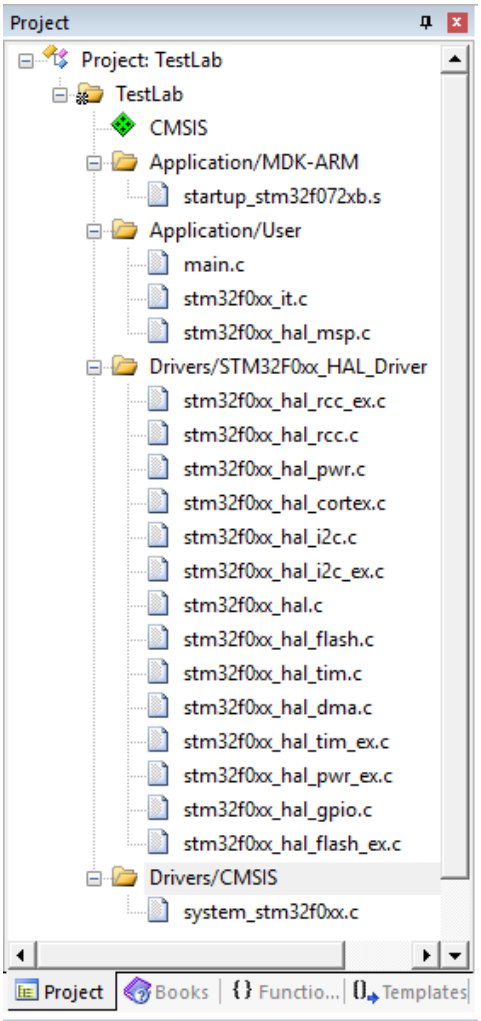


Figure 7: Kiel µVision Project Structure

Within the main function, you will notice some automatically generated comments marking boundaries of user/auto-generated code. STMCube uses these boundaries to locate and preserve the user's code if the user regenerates the project with new settings. You may decide to keep or remove these comments, but we recommend that you manually backup changes to the *main.c* file when regenerating code in STMCube.

Blinky Example Code

Figure 8 on the following page contains a demo program that blinks the green and orange LEDs on the Discovery board. This program (provided in the lab files) functions as a replacement for the *main.c* in your new project. Download this file from the assignment page and replace the *main.c* located in the *src* folder of the project directory.

After replacing the main function with the "blinky" example, compile the project with **Project Menu →Build Target**.

```
int main(void) {
    HAL_Init(); // Reset of all peripherals, init the Flash and Systick
    SystemClock_Config(); //Configure the system clock

    /* This example uses HAL library calls to control
       the GPIOC peripheral. You'll be redoing this code
       with hardware register access. */

    __HAL_RCC_GPIOC_CLK_ENABLE(); // Enable the GPIOC clock in the RCC

    // Set up a configuration struct to pass to the initialization function
    GPIO_InitTypeDef initStr = {GPIO_PIN_8 | GPIO_PIN_9,
                                GPIO_MODE_OUTPUT_PP,
                                GPIO_SPEED_FREQ_LOW,
                                GPIO_NOPULL};
    HAL_GPIO_Init(GPIOC, &initStr); // Initialize pins PC8 & PC9
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_SET); // Start PC8 high
    while (1) {
        HAL_Delay(200); // Delay 200ms
        // Toggle the output state of both PC8 and PC9
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_8 | GPIO_PIN_9);
    }
}
```

Figure 8: Example “blinky” program.

Kiel µVision uses a custom build/dependency system similar to GNU-Make to check and compile all dependencies for each application file. You can see the output from the build system in the console at the bottom of the main window.

If you generated the template project and copied the provided code correctly, you shouldn't have any errors in your completed result. Connect the STM32F072 Discovery board to the host computer using a mini-USB cable and the ST-Link USB port.

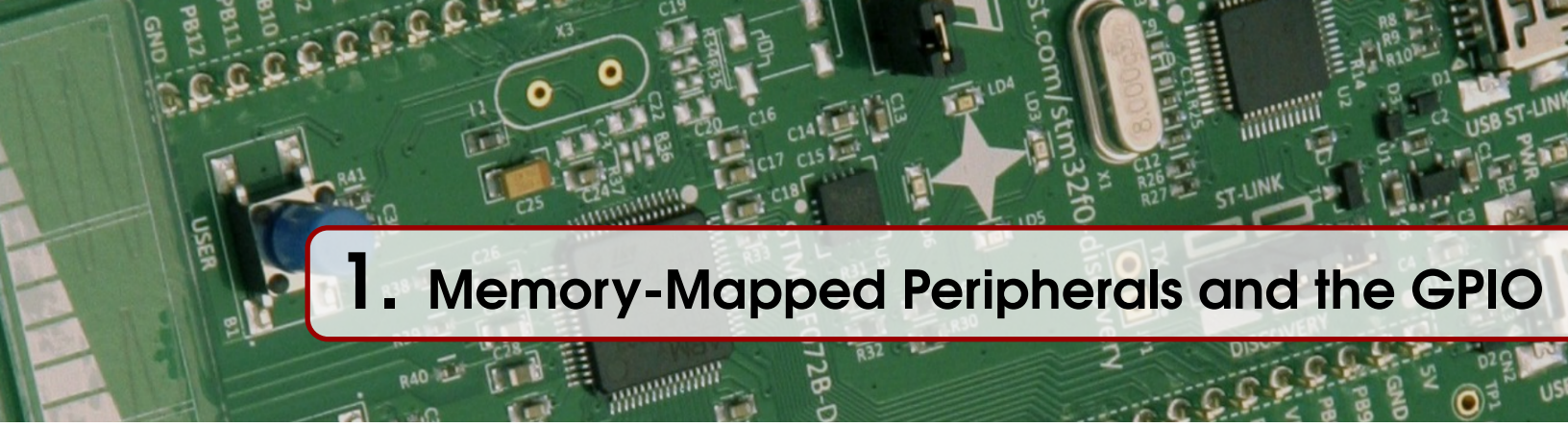
Remember that there are two ports available on the Discovery board: the center port connects to the ST-Link debugger, while the other is for user applications.

After connecting the Discovery board, program the target STM32F0 through **Flash Menu →Download**. You will see a brief progress bar and status report in the µVision console, followed by a completion status message.

After the loading process has completed, press and release the **RESET** button on the Discovery board. Assuming that the console reports no errors, you should see the orange and green lights begin to flash in an alternating pattern.

Remember to press the **RESET** button on your STM32F0 board after loading a project to your board; without it, your board will not begin the program you loaded.

Congratulations, you are running a custom application on the STM32F0!



1. Memory-Mapped Peripherals and the GPIO

INTRO 1.1 Overview

This section provides an overview of embedded programs, memory-mapped peripherals, and device documentation. The exercises in this lab provide an introduction to memory-mapped register access by guiding the user through configuration of the "GPIO" or general purpose input-output peripheral. The GPIO is one of the fundamental peripherals available and provides access to the physical pins of the STM32F0 device.

INTRO 1.2 Basic Design of an Embedded Program

The typical layout of an embedded program differs significantly from most standard programming systems: figure 1.1 on page 17 shows a comparison between devices running an operating system and the operation of many embedded applications. In a computer with an operating system, layers of supervisory code manage low-level system operations and provide frameworks for resource sharing between multiple applications. In these systems, a user application may exit, but the operating system will continue to operate and clean up afterward. In contrast, many embedded applications execute directly on the processor core, and no operating system exists to launch or clean up after applications exit. This style of development is called "bare-metal" programming.

The beginning of every STM32F0 binary executable contains a jump to a small region of code called the *Reset Vector*. This reset vector always executes directly after a hardware reset and is responsible for ensuring that the stack, heap, and processor clock initialize to basic settings—allowing more complex code to operate.

Most of the initialization code typically lies outside the reset vector itself: scattered throughout the support files that ST Microelectronics and the MDK:ARM toolchain provide. The HAL library is typically used to perform less-critical initialization steps such as clock speed configuration at the beginning of the user's application.

After device initialization, the startup code calls the `main()` function within the user's application; embedded programs generally begin by configuring hardware peripherals that they require and subsequently enter into an infinite loop where the majority of the application resides. This endless loop is necessary since the main function of an embedded program should never return. Unlike a machine with an operating system, nothing catches the processor's execution after the main program exits; this means that the behavior of the processor after returning is undefined—which could range from resetting the device to executing random data!

INTRO 1.3 Introduction to Device Peripherals

The last lab section introduced the "blinky" example program. (shown in Figure 8 on page 13) This simple program repeatedly toggles pins on the STM32F0 using the GPIO hardware abstraction library (HAL) drivers; this example produces a pattern of alternating flashes on the pins used, which connect to LEDs on the STM32F072 Discovery board.

A closer look at the program structure reveals that it begins by initializing the HAL library and system clock; it then initializes the specific GPIO peripheral controlling the desired pins and enters an infinite `while` loop. Although many of the function calls in this example program are straightforward and simple to understand, an important question to consider is how the HAL library functions modify the physical peripherals of the STM32F0 device.

1.3.1 Memory-Mapped Peripherals on the STM32F0

Figure 1.2 on page 18 contains the system block diagram for the STM32F0 family of processors. As shown, an enormous number of peripherals are available, many with complex interconnections. Luckily, the toolchain hides most of the complicated hardware-level implementation from the end user.

However, how does the processor interact with the complex hardware surrounding it? An engineer can choose among many methods when designing a processor core. One method is to develop the machine language interface to understand specific instructions telling it to manipulate the hardware in a certain way. This method seems straightforward, but it has the disadvantage of increasing the size and complexity of the instruction decoder hardware. Likewise, if the processor has a significant number of peripherals, the resulting instruction set becomes enormous.

ARM processors belong to the *reduced-instruction-set-computing* (RISC) class of devices. Processors of this type have relatively simple instruction sets that and typically perform a limited number of tasks. These tasks/instructions primarily involve arithmetic operations and moving data around the system's memory.

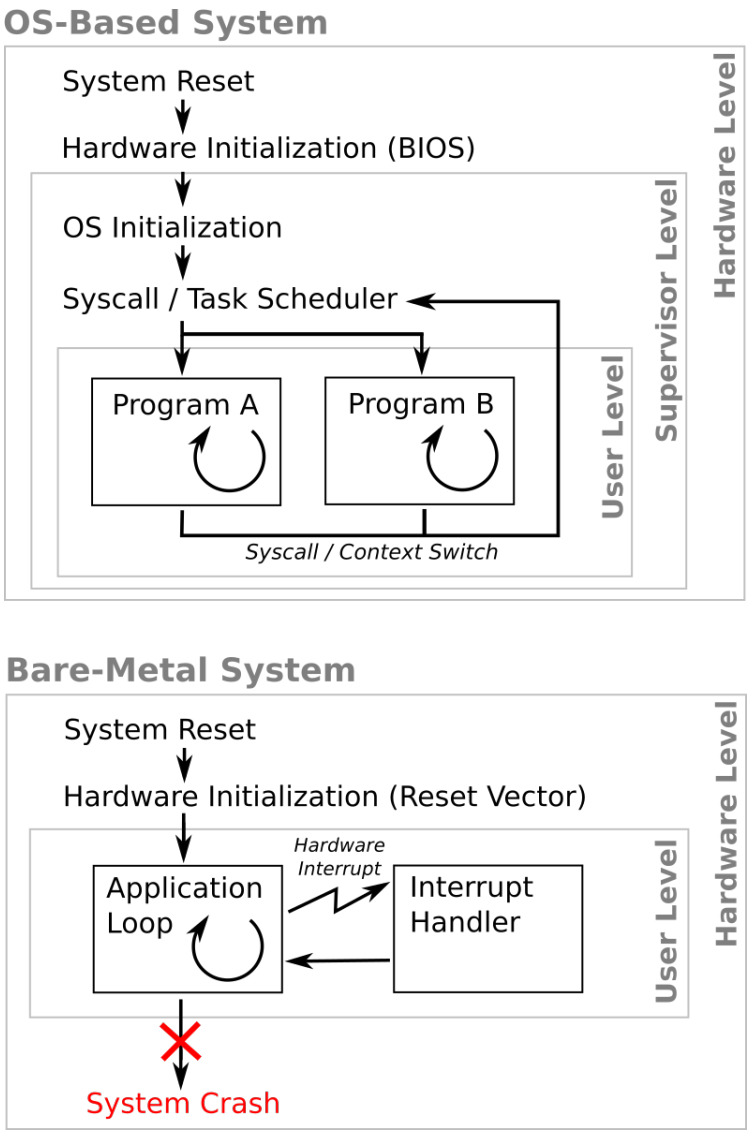


Figure 1.1: Comparison of operating-system and bare-metal systems.

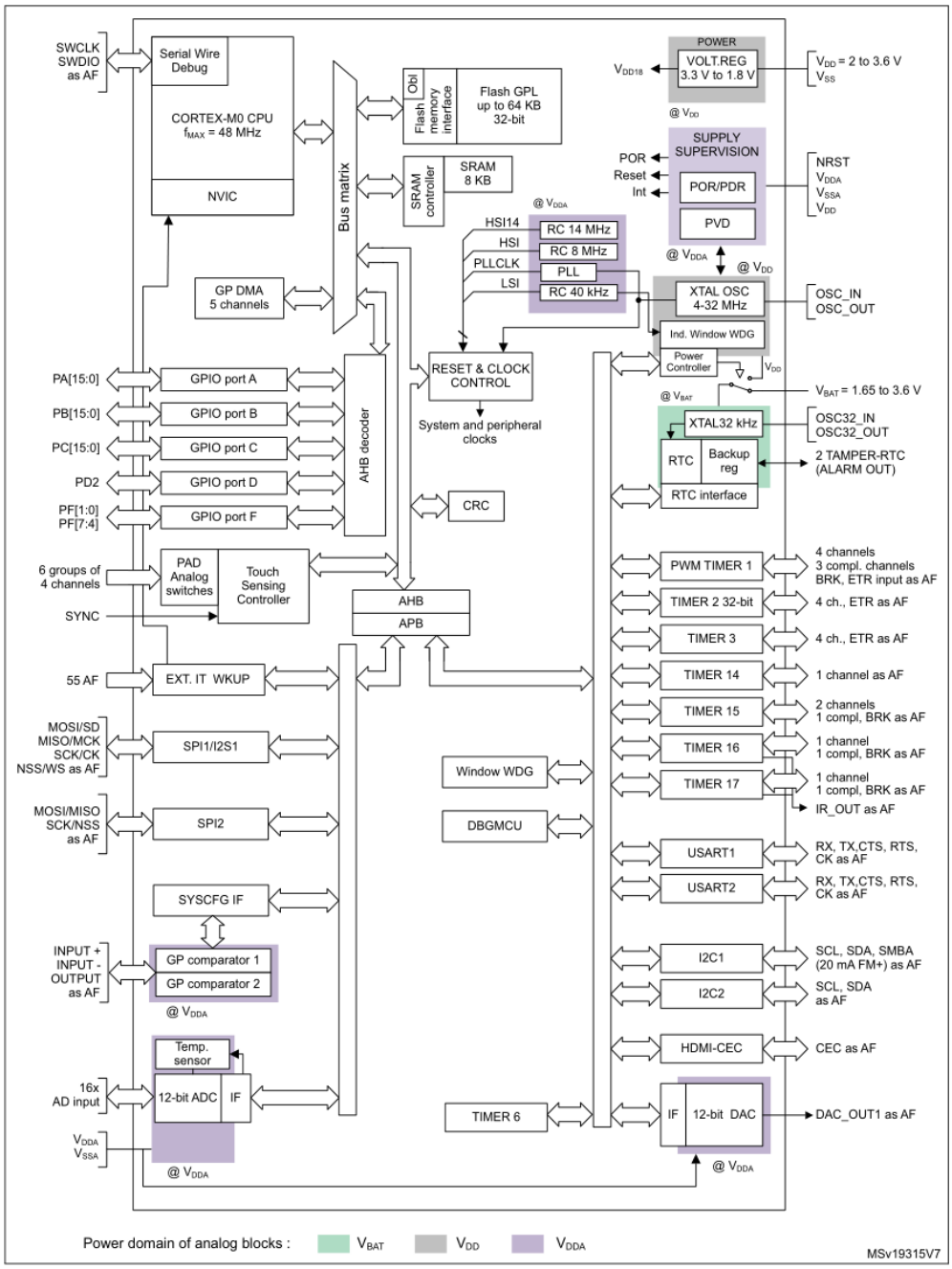


Figure 1.2: Peripheral block diagram for the STM32F0 devices

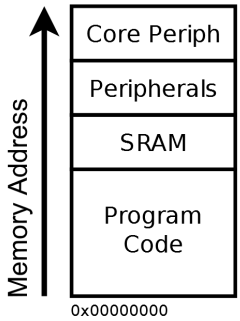


Figure 1.3: Simplified memory table for STM32F0 devices.

Because of this limitation, one of the most common ways of controlling system hardware is to designate regions of memory as *peripheral registers*. When an application writes to the memory address of one of these registers, the memory bus controller recognizes the address as belonging to a specific peripheral; it then routes the data to the appropriate control circuitry for that peripheral instead of system memory. Because these hardware registers are mapped into the system's memory address space, the processor simply considers them as conventional variables.

Figure 1.3 shows a simplified memory map for a STM32F0 processor. The STM32F0 devices have only a few kilobytes of combined physical memory, program storage, and mapped peripheral registers; however, they have a full 32-bit (4GB) address space. Therefore, huge blocks of unimplemented addresses separate each functional region of the address space (program storage, physical memory, and peripheral registers). These gaps in the address space ensure that each type of memory has a unique-bit-pattern and provides for future expansion when developing new chips.

1.3.2 Differences Between ARM-Core and Device Peripherals

One unique feature of all ARM chips is that the manufacturers develop the processor core separately from the rest of the device. ARM Ltd. develops the Cortex processors and licenses the designs to hardware manufacturers to implement into real systems.

Due to this separate development, the processor core contains its own set of peripherals. These peripherals have a much closer relationship to the operation of the processor and involve actions that directly manipulate the flow of instruction execution. Hardware interrupts and program debugging are examples of this. Because core peripherals are separate physically from the others within the device, they are memory-mapped within a separate region. However, the biggest difference to system programmers is that these peripherals have separate documentation from the others.

1.3.3 Device Datasheets and Manuals

Efficiently searching datasheets and manuals is one of the most important skills that an embedded engineer can develop. This trait is important because the sheer amount of information required to understand the operation of many embedded devices completely is staggering. Engineers who can rapidly search for information within device documentation need not memorize operation details; likewise, it becomes easier to work with an unknown device.

Four files provide the documentation for the exercises in these labs. You may download these from the following links. However, it is always a good idea to know how to find them on a manufacturer's website.

1. **(STM32F072RBT6 Datasheet) DM00090510.pdf**
   - The chip datasheet provides device-specific details for the processor; this includes pin connections for available chip packages and a list of available peripherals.
2. **(Programming & Core Manual) DM00051352.pdf**
   - The core programming manual provides information on the ARM-core peripherals as well as the assembly instruction set; it is generic to all of the processors within the STM32F0 family.
3. **(Peripheral Manual) DM00031936.pdf**
   - The peripheral reference manual contains detailed information on all peripherals available within an STM32F0 device; however, not all STM32F0 devices contain every peripheral! The chip datasheet is necessary to determine which peripherals are available for use.
4. **(Discovery Board Manual) DM00099401.pdf**
   - The Discovery board manual contains schematics and tables that show the onboard devices and connectors attached to the STM32F0; the Discovery board silkscreen also documents many device connections.

These manuals are large! Do not expect to read them fully: they simply contain too much information; for example, the peripheral reference manual contains over one thousand pages of detailed descriptions about each peripherals operation and configuration.

STM32F0 1.4 Using Peripheral Control Registers

1.4.1 The General Purpose Input-Output Peripheral (GPIO)

The GPIO is one of the most fundamental peripherals within the STM32F0. Although one of the simplest peripherals, it is still surprisingly complex and very powerful.

Open the STM32F0 peripheral reference manual to page 118; this should contain the heading "General-purpose I/Os (GPIO)." The reference manual includes bookmark links; if the PDF viewer used can show these (see Figure 1.4 on the following page), use them to navigate to desired sections quickly.

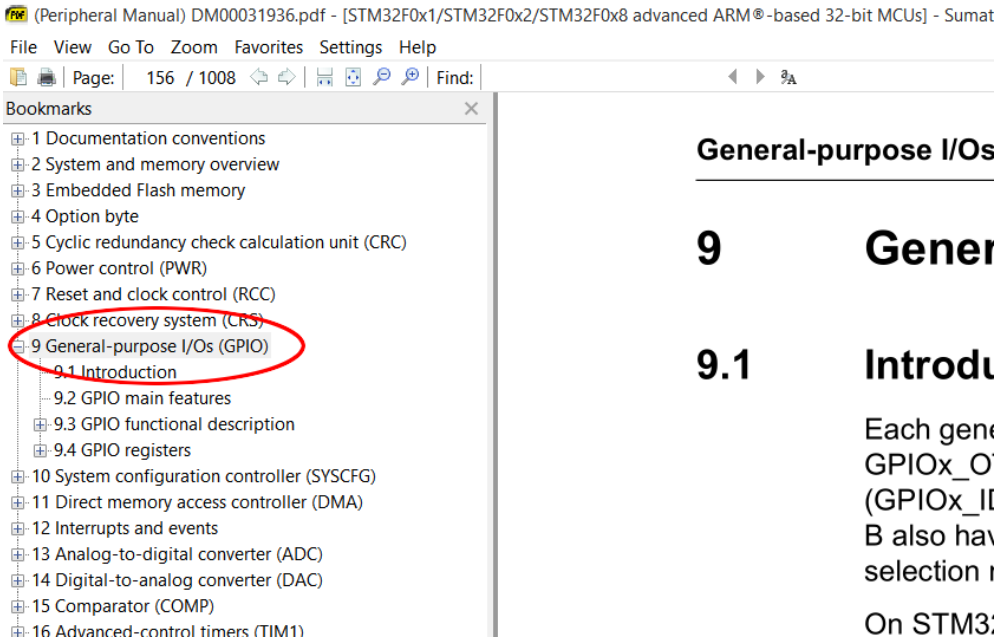


Figure 1.4: Navigation bookmarks in the peripheral reference manual.

Each section of the reference manual first introduces the available features and modes of the peripheral. It subsequently explains in detail the operation of the different modes—including graphs, figures, and tables, all showing characteristics of the peripheral's operation. Finally, it provides detailed explanations of each user-accessible control register that configures the peripheral.

To understand the GPIO control registers, it is helpful to discuss the different modes in which it can operate: go back to page 118 in the reference manual (beginning of GPIO section) and look through the information on that page.

The *GPIO main features* section indicates that a GPIO pin can have the following modes:

- **General Purpose Output Configuration**
  - *Push-Pull* – logic '1' pushes pin to Vcc; logic '0' pulls to GND.
  - *Open-Drain* – only logic '0' pulls pin to GND; floats when logic '1'.
- **Digital Input Configuration**
  - *Floating* – pin has high-impedance, voltage "floats" unless externally pulled to a state.
  - *Internal Pull-Up/Down* – pin has internal "resistor" to Vcc or GND.
- **Analog Input Configuration**
  - Pin connects to the system analog peripherals instead of a digital comparator.
- **Alternate Function Configuration**
  - Pin connects to a selection of internal peripherals, which directly control the pin operation.

Additionally, other features such as configurable speed, register locking, bit-set-reset and more are possible; this lab section only considers two basic configurations: push-pull output and floating input.

Begin by examining the configuration registers within the GPIO peripheral beginning on page 127; the manual begins with the most important register of them all.

GPIO Port Mode Register (GPIOx_MODER)

The port mode is the most important register in the GPIO peripheral; without configuring this register, using GPIO pins as anything aside from digital inputs is not possible. Examining the register map in the manual shows a table of 16 cells titled "MODER0[1:0]" through "MODER15[1:0]"; notice that above these cells are numbers ranging from 0-31 and below are cells with "rw" in them. Breaking these down gives the following explanations:

- The STM32F0 processors have multiple GPIO peripherals known as *ports*, named in the following manner: GPIOA, GPIOB ... GPIOx. Within a GPIO port, there can be up to 16 pins; these pins are named in the following manner: PA0 (GPIOA, Pin 0) ... PA15 (GPIOA, Pin 15), PB0 (GPIOB, Pin 0) ..., etc.
- STM32F0 devices do not recognize inputs/outputs on a chip by physical pin numbering. This is due to different chip packages with differing numbers of pins, and the pin ordering between these is inconsistent; GPIO pins are instead labeled with a port name (PA0 for example) which describes where to go to configure it. Within the chip datasheet, we see a table mapping GPIO pin names to physical pin numbers on the specific chip package.
- Since there can be up to 16 pins in a GPIO bank, the mode register contains 16 "MODER" cells (pairs of bits); because GPIO pins start counting from 0, the 16 pins number from 0 through 15.

- The number above each cell in the register map indicates the bit number within the register. Each "MODER" cell consists of 2 bits; this represents the four main configuration modes for those pins. A description of the different modes, as well as the bit patterns used to select them, lie directly below the register map. To select one of these modes, set the correct bits within the appropriate "MODER," cell with the corresponding bit pattern.
- The "rw" below each of the cells in the register map indicates that the user may read and write to that bit; sometimes individual bits or entire registers are read-only or write-only, as we shall see later on.

■ **Example 1.1 — Setting a Pin to Output Mode.** The following steps configure pin PB3 as an output. Figure 1.5 on the next page outlines these using the GPIOx_MODER register map in the peripheral reference manual.

1. The name PB3 indicates that the pin belongs to the GPIOB peripheral and is the fourth pin in the control register (pin numbering begins at zero).
2. The fourth cell in the GPIOx_MODER register map is "MODER3" and contains bits 6-7.
3. Examine the bit patterns listed below the register map; one of these {`01`} corresponds to "general purpose output mode."
4. The output mode bit pattern indicates that for the two configuration bits; set the lower bit to '1', and clear the upper bit to '0'.
5. Since bits 6-7 in the GPIOx_MODER register control pin PB3, setting bit 6 and clearing bit 7 configures the pin to output mode.

■

Exercise 1.1 — Configuring GPIO Pin Modes . As demonstrated in the previous example, find and list the bit numbers to set or clear in the GPIOx_MODER register (and indicate if you would set or clear those bits) to put the following pins into the requested modes:

- **PC6** (RED LED) – General Purpose Output
- **PC7** (BLUE LED) – General Purpose Output
- **PC8** (ORANGE LED) – General Purpose Output
- **PC9** (GREEN LED) – General Purpose Output
- **PA0** (USER Button) – Digital Input

■

GPIO port output type register (GPIOx_OTYPER)

This register selects the output mode you want for each pin. For pins without output mode configured, the bits in this register have no effect. Since we have only two possible modes, we use one bit per pin; the rest of the register is "reserved", and you shouldn't try to write anything there.

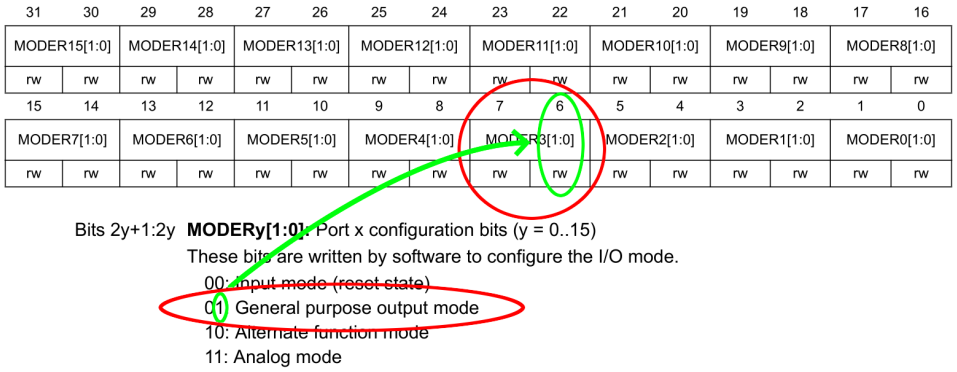


Figure 1.5: Configuring the mode register to set a pin to output mode.

GPIO port output speed register (GPIOx_OSPEEDR)

One of the major selling points of ARM processors is that they typically use very low power; one way that they manage to reach such low power consumption is that the processor disables and/or reduces the speed of the peripherals by default. GPIO pins on an STM32F0 have three different speed modes, with the slowest using the least amount of power and the fastest using the most.

You might notice that the reference manual does not indicate how fast each speed mode operates; this is due to the dependence on the specific STM32F0 processor. You can find out the different speed modes by looking in the device datasheet, section 6.3.14 (Electrical characteristics →Operating conditions →I/O port characteristics →I/O AC characteristics).

GPIO port pull-up/pull-down register (GPIOx_PUPDR)

This register connects internal pull-up or pull-down "resistors" to a pin; these resistors are actually transistors that are not completely turned on but will leak enough current to act as a high-value resistor (typically, >100kOhm). Generally, only input modes use the internal pull resistors.

Notice that one of the bit patterns remains reserved for this register; that is,you should never set the bit-pairs in the register to that state, although it is possible to turn on both the pull-up and the pull-down resistors and waste power.

GPIO port input data register (GPIOx_IDR)

All of the registers that we've seen so far have been read-write: we can both read and write to them to modify pin configurations. If you look at the register map, you'll find that all the bits in this register are read-only.

The data input register always reports the logical state of each pin in the GPIO port. If the pin is an input, then the matching bit will show the logical state to which the outside world is driving that pin; if configured as an output, then it will show whatever logical state that you have set in the output register for that pin.

GPIO port output data register (GPIOx_ODR)

This register sets the logical state of configured output pins. Writing a '0' to this register will pull the output low for that pin; writing a '1' to the bit will drive the output high.

GPIO port bit set/reset register (GPIOx_BSRR)

This register is write-only: the reason is that this register is a shortcut to set and clear bits quickly in the output register. Typically you can't just overwrite an entire register with a new value for a single bit because there are other bits that you don't want to overwrite; you first must read the register value, perform a bitwise operation to modify the desired bit(s), and then write it back to the register.

The bit set/reset register allows much faster modification to the output register because you only change the desired values. You may simply overwrite the entire register; it will *only* modify the output register on the bits that you have set.The lower half of this register sets bits in the output, and the upper half clears/resets them.

GPIO port configuration lock register (GPIOx_LCKR)

The configuration lock register locks the other configuration registers for the associated pin; this can prevent a malfunctioning program from accidentally changing a pin into an undesired mode. Once activated, you may not edit the locked register without first processing a timed sequence of writes on a specific bit. This is an advanced feature that you typically won't use.

GPIO alternate function low/high registers (GPIOx_AFRL/GPIOx_AFRH)

Notice that every pin has four bits for configuration; this means that there are two 32-bit registers (AFRL & AFRH) necessary to configure alternate functions for all 16 pins. The alternate function connections for each pin depends on the specific processor and pin. The device specific datasheet contains a table which lists the possible alternate functions for each pin, as well as the number of each function.

We will use alternate functions in the next lab to connect GPIO pins to timer peripherals.

GPIO port bit reset register (GPIOx_BRR)

This last control register in the GPIO peripheral is very similar to the bit set/reset register; although you can clear bits using the bit set/reset register, all of the clearing bits are in the upper half of the register. This reset-only register is essentially a copy but with the clearing bits in the lower half. If you look at this register and the BSRR and their effects on the output register, you might see why some people wanted the clear bits also in the low positions.

Exercise 1.2 — Configuring GPIO Pin Details. Use the GPIO register documentation in the peripheral reference manual to determine the bits to modify in order to configure the listed parameters for the following pins:

- **PC6, PC7, PC8, PC9** (LED PINS)
  - *GPIOx_OTYPER* – Push-Pull Output Type

9.4.1 GPIO port mode register (GPIOx_MODER) (x =A..F)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 1.6: GPIO mode register with keyword names circled

  - *GPIOx_OSPEEDR* – Low Speed
  - *GPIOx_PUPDR* – No Pull-Up or Pull-Down
- **PA0** (USER Button)
  - *GPIOx_OSPEEDR* – Low Speed
  - *GPIOx_PUPDR* – Pull-Down Resistor

■

1.4.2 Finding Register Definitions in ST's Header Files

Take a look at the following line of code: you'll notice that I cast a hexadecimal memory address to a pointer, dereference that pointer and write data to the memory location that the dereferenced pointer references. This code accesses the GPIOC mode register (GPIOC_MODER) and configures pins PC8 and PC9 to output mode.

```
*((volatile uint32_t*) 0x48000800) = 0x50000;
```

While pointers and memory addresses are technically the only method available to interact with peripheral registers, the pointer cast and dereference syntax is not particularly convenient.

Although most embedded code simplifies into statements like this, we do not typically look up memory addresses and make/dereference pointers as previously shown; ST provides header files which name and organize hardware registers into structures that are reasonable to use. Using these header files, the pointer code above becomes much simpler as shown below.

```
GPIOC->MODER = 0x50000;
```

Looking back to the GPIO register pages in the peripheral reference manual, you will notice that each register has an abbreviated name; you will see that the bits within the registers also have names. Figure 1.6 shows the highlighted names for the GPIO mode register.

A good question at this point would be: "Where do we find the names for registers, and how does a C program access them?" The answer lies within the *CMSIS Cortex-M0 Device Peripheral Access Layer Header Files*, also known as *stm32f0xx.h* and its derivatives. Originally, *stm32f0xx.h* defined all peripheral information for the entire STM32F0 line. However, in recent versions of the HAL, ST Microelectronics has split the information for each device sub-family into separate files.

```
386 /**
387   * @brief General Purpose I/O
388   */
389
390 typedef struct
391 {
392   __IO uint32_t MODER;    /*!< GPIO port mode register,               Address offset: 0x00      */
393   __IO uint32_t OTYPER;   /*!< GPIO port output type register,        Address offset: 0x04      */
394   __IO uint32_t OSPEEDR;  /*!< GPIO port output speed register,       Address offset: 0x08      */
395   __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register,  Address offset: 0x0C      */
396   __IO uint32_t IDR;      /*!< GPIO port input data register,         Address offset: 0x10      */
397   __IO uint32_t ODR;      /*!< GPIO port output data register,        Address offset: 0x14      */
398   __IO uint32_t BSRR;     /*!< GPIO port bit set/reset register,      Address offset: 0x1A      */
399   __IO uint32_t LCKR;     /*!< GPIO port configuration lock register, Address offset: 0x1C      */
400   __IO uint32_t AFR[2];   /*!< GPIO alternate function low register,  Address offset: 0x20-0x24 */
401   __IO uint32_t BRR;      /*!< GPIO bit reset register,               Address offset: 0x28      */
402 } GPIO_TypeDef;
```

Figure 1.7: GPIO structure definition

Open a µVision project and expand the project hierarchy under *main.c*. If µVision doesn't show any included headers after expanding the hierarchy, compile the project, and it should update after parsing the code dependencies. Within the list of these files, you should see the both *stm32f0xx.h* and *stm32f072xb.h*. Alternatively, you can locate these files manually by searching for them within the project directory under: "Drivers\CMSIS\Device\ST\STM32F0xx\Include".

Searching the *stm32f072xb.h* File

Open the *stm32f072xb.h* file, and look for the introductory header comment which gives a summary of the functionality provided. Many IDE's like to collapse large sections of commented text, so you may need to expand the comment with the small "+" icon near the line numbers. Because the *stm32f072xb.h* file is large (>10000 lines) you will need to use the text search features of the IDE to navigate; to do this, we must get familiar with how the file organizes peripherals and their registers.

Move down to line 390; you will notice that this is a `typedef` for a C-structure representing a GPIO peripheral; this structure type creates a neat package for named pointers to each of the control registers for a peripheral. Since this code is a C-type definition, the system can create named copies of the structure for every GPIO peripheral in the device. The *stm32f072xb.h* file also defines the contents of the register pointers in each peripheral structure, so we don't need to deal directly with memory addresses.

Returning to the top of the header file, open µVision's text search feature by selecting: **Edit Menu →Find**. Type *GPIO* into the search box and hit the FIND button. More likely than not, µVision's took you directly to line 390 again. Typically, searching with a guess of the peripheral name is the easiest way to start finding the documentation you'll need. Now, open the text search window again, and search for the structure name defining the GPIO peripheral—you'll find it right after the closing curly brace at the end of the GPIO structure code ("GPIO_TypeDef").

Searching for the name of the GPIO structure likely took you to line 785; here, we see the multiple named copies of the GPIO structure: one for each of the physical GPIO peripherals within the STM32F0 chip. These `define` statements show the actual name you will use to access the structure for that peripheral.

Returning again to the structure definition (line 390), try searching on one of the register names it defines. If you searched for the MODER register, you will find that it took you to line 6499, where a list of `defines` gives names to each of the bits in the register. For simple peripherals like the GPIO, we typically won't use the bit names since their purpose is fairly straightforward and all have similar functions (just on different pins); for more complex peripherals, the bit names often tell you their function. We'll see an example of this later.

1.4.3 Bitwise Operations on Peripheral Registers

This lab assumes that you have had prior experience with *bitwise* logic operations; we'll therefore provide only a rapid overview of common tasks. Bitwise logic allows us to manipulate portions of a register without overwriting the others.

Setting bits

To set bits in a register, bitwise-OR its value with a bitmask. Any bits set in the bitmask will set the corresponding bit in the register. The bitwise-OR operator is a single vertical-pipe character '|'.

```
GPIOC->MODER |= 0b00001000; // Sets the 3rd bit
```

Clearing bits

To clear bits in a register, bitwise-AND its value with a bitwise-inverted bitmask: this will clear any bits set in the bitmask. The bitwise-AND operator is a single ampersand character '&', and the bitwise-invert operator character is '~'.

```
GPIOC->MODER &= ~(0b00001000); // Clears the 3rd bit
```

Inverting/Toggling bits

Although we have a bitwise-invert operator '~', it inverts every bit in the target. To selectively invert a few bits in a register, bitwise-XOR its value with a bitmask; any bits set in the bitmask will cause the matching bit in the register's value to invert. The bitwise-XOR operator is a single caret character '^.'

```
GPIOC->MODER ^= 0b00001000; // Inverts the 3rd bit
```

Note that µVision does **not** allow binary constants with the "`0b`" prefix—the binary constants shown here are used only for clarity. Within the actual code, hexadecimal constants with the "`0x`" prefix are typically used when explicit values are required.

Building Bitmasks Easily

Typically building bitmasks using binary or hexadecimal constants makes code difficult to read. There are a number of ways to improve readability, one of which is to use shifted constants. The C compiler is very good at simplifying mathematical expressions that result in a constant value; we use this to our advantage to "build" a bitmask out of multiple statements. The examples in figure 1.8 on the following page build bitmasks by shifting the decimal value '1' (a single binary bit at the lowest position) to the bit position we wish to modify.

```
// Sets the 3rd bit in the GPIOC_MODER register
GPIOC->MODER |= (1 << 3);

// Sets the 3rd and 5th bits in the GPIOC_MODER register
GPIOC->MODER |= (1 << 3) | (1 << 5);

// Clears the 3rd and 5th bits in the GPIOC_MODER register
GPIOC->MODER &= ~((1 << 3) | (1 << 5));
```

Figure 1.8: Common bitwise operations

Exercise 1.3 — Initializing the GPIO Pins. We recommend that you begin with the provided "blinky" example code and make incremental changes from the HAL library calls to bitwise operations on peripheral registers.

Using the bit numbers and registers that you found in the earlier exercises, configure the GPIO pins to the mode and parameters.

- Remove the `GPIO_InitTypeDef` struct and all `HAL_GPIO_Init()` function calls.
- Make sure to reference the correct GPIO peripheral indicated by the pin name.
- Configure LED pins (PC6-PC9) in the following way:
  - General-purpose output mode using the MODER register.
  - Push-pull output type using the OTYPER register.
  - Low speed using the OSPEEDR register.
  - No pull-up/down resistors using the PUPDR register.
- USER Button pin (PA0) should be configured to:
  - Digital input mode using the MODER register.
  - Low speed using the OSPEEDR register.
  - Pull-down resistor using the PUPDR register.

■

Reading the State of a Bit

Many of the hardware registers within the STM32F0 contain status bits: these indicate events that have occurred within the peripheral, or the current state of device inputs. An example of a register used for this purpose is the GPIO input data register (IDR).

When comparing a register against a constant value, the processor considers all bits within the register, including those that may have no relation to the comparison. In many cases, it is desirable, therefore, to separate and check the status of a single bit within a register. Fortunately, you may isolate individual bits using a bitmask and the bitwise AND operation.

```
458 /**
459   * @brief Reset and Clock Control
460   */
461
462 typedef struct
463 {
464   __IO uint32_t CR;       /*!< RCC clock control register,                  Address offset: 0x00 */
465   __IO uint32_t CFGR;     /*!< RCC clock configuration register,            Address offset: 0x04 */
466   __IO uint32_t CIR;      /*!< RCC clock interrupt register,                Address offset: 0x08 */
467   __IO uint32_t APB2RSTR; /*!< RCC APB2 peripheral reset register,          Address offset: 0x0C */
468   __IO uint32_t APB1RSTR; /*!< RCC APB1 peripheral reset register,          Address offset: 0x10 */
469   __IO uint32_t AHBENR;   /*!< RCC AHB peripheral clock register,           Address offset: 0x14 */
470   __IO uint32_t APB2ENR;  /*!< RCC APB2 peripheral clock enable register,   Address offset: 0x18 */
471   __IO uint32_t APB1ENR;  /*!< RCC APB1 peripheral clock enable register,   Address offset: 0x1C */
472   __IO uint32_t BDCR;     /*!< RCC Backup domain control register,          Address offset: 0x20 */
473   __IO uint32_t CSR;      /*!< RCC clock control & status register,         Address offset: 0x24 */
474   __IO uint32_t AHBRSTR;  /*!< RCC AHB peripheral reset register,           Address offset: 0x28 */
475   __IO uint32_t CFGR2;    /*!< RCC clock configuration register 2,          Address offset: 0x2C */
476   __IO uint32_t CFGR3;    /*!< RCC clock configuration register 3,          Address offset: 0x30 */
477   __IO uint32_t CR2;      /*!< RCC clock control register 2,                Address offset: 0x34 */
478 } RCC_TypeDef;
```

Figure 1.9: RCC structure with peripheral clock enable registers highlighted

■ **Example 1.2 — Checking Specific Bits.** In the first code example below, the first constant represents a register value and the second a bitmask selecting the bit(s) to examine. To isolate a specific bit from a register, set the matching bit at the same position within the bitmask. When bitwise AND'ed together, the result will be zero if the checked bit was zero, and non-zero if the checked bit was set.

```
0b01001101 & 0b01000000 = 0b01000000 // Bitmask isolates bit 6
```

```
if(GPIOC->IDR & 0x40){...} // Triggers if bit 6 is set
```

■

Exercise 1.4 — Reading the Button Pin. The logical state of each GPIO pin can be determined by checking the appropriate bit in the GPIO input data register (GPIOx_IDR).

■

1.4.4 Enabling the System Clock to Device Peripherals

Just as each GPIO pin has multiple speed settings to conserve power, most peripherals in the STM32F0 have clock signals disabled by default. Since synchronous circuits don't operate without a clock signal, they are essentially "turned-off"; They consume significantly less power than they would sitting idle, but you cannot read from or write to any of their registers.

The STM32F0 family has a dedicated peripheral called the *Reset and Clock Control* (RCC) which enables or disables clock signals around the chip. To enable a clock for a peripheral, we'll need the proper RCC enable register. Search the *stm32f072xb.h* file for "RCC", you will find a few matches within interrupt definition code, but continue until you find the "RCC_TypeDef" definition. Within this structure, you see that three of the registers have "peripheral clock registers" labels.

These three registers control the clock signals to all other peripherals—except for those in the ARM-core itself. The peripherals which they control are organized by what system communications bus to which they connect. Some high-speed peripherals—such as memory control—connect directly to the *Advanced High-performance Bus* (AHB). However, most peripherals connect to one of two *Advanced Peripheral Busses* (APB).

On the STM32F0, the GPIO peripherals are within the AHB bus. However, in other architectures and brands, they may be on the APB.

■ **Example 1.3 — Enabling a Peripheral Clock in the RCC.** The following code segment enables the peripheral clock for the GPIOB peripheral. Notice that it uses a defined bit name instead of a simple bitmask. In many cases using these definitions results in clearer code.

```
RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; // Enable peripheral clock
to TIMER2
```

■

1.4.5 Slowing the System Down

Even though the STM32F072 operates at a far slower clock rate than a conventional computer, it executes instructions fast enough that a simple pin toggling loop will flash the LED's faster than the human eye can see; therefore, we'll have to introduce our own methods of adding delay into the program.

Delay Loops

One simple (and very inefficient) way to add delay into a program is to make the processor perform lots of useless work in a *busy loop*. A typical method of implementing a busy loop is to repeatedly increment a number until it reaches a limit and exits.

Unfortunately, compilers are pretty good at detecting and removing "useless" code that doesn't seem to be useful anywhere else in the program. To help force the compiler to leave a delay loop alone, you will need to tag the loop variable as `volatile`. One of the uses of the `volatile` keyword is to tell the system that the tagged variable has desired side-effects that the compiler may not see; without this keyword, the compiler will optimize and remove structures like our delay loop.

```
volatile int i;
for(i=0; i < 100000; i++){}
```

```
volatile int i;
for(i=0; i < 100000; i++){}
```

In practice, delay loops are one of the least effective ways to introduce delay into a program. If possible, avoid using delay loops if hardware timing capabilities are available.

Always use the HAL library delay functions when blocking delays are required. Do not use busy-loops when timing hardware is available.

```
#define uint32_t unsigned int   // Unsigned 32-bit integer
#define int32_t int             // Signed 32-bit integer

#define uint16_t unsigned short // Unsigned 16-bit integer
#define int16_t short           // Signed 16-bit integer

#define uint8_t unsigned char   // Unsigned 8-bit integer
#define int8_t char             // Signed 8-bit integer
```

Figure 1.10: Integer variable types defined in ST Microelectronics support files.

Hardware SysTick Delay

The *SysTick* timer peripheral is a device which raises a system signal at a configurable periodic rate; since the duration between these signals is a known quantity, the SysTick is useful as an application heartbeat. The HAL library uses the SysTick to trigger periodic tasks such as updating a global system time variable.

As seen in the "blinky" example, the infinite loop calls the `HAL_Delay()` function. This function is one of the delay mechanisms in the HAL library, and it halts the execution of a program by the number of milliseconds given in the argument.

```
HAL_Delay(200); // Delay 200ms
```

Although these labs will typically avoid using the HAL library resources, feel free to make use of the HAL delay functions. One of the exercises in the next lab on interrupts will explore writing a similar delay function using the SysTick peripheral. However, since the HAL delay system initializes automatically during the clock speed configuration, it is easier to use the preexisting functionality for most purposes.

1.4.6 Standard Integer Types

While most of the C-standard variable types work properly on embedded systems, there is always the possibility that they are not the size (number of bits) usually expected. For example, similar to many systems an "int" on the STM32F0 is 32-bits wide. However, on 8-bit processors, an integer type is typically 8 or 16 bits. To avoid confusion and very subtle bugs, most toolchains define explicit numeric types. Figure 1.10 shows the integer types defined and used within ST's provided files.

Avoid floating-point types when possible! Many embedded devices do not have hardware support for floating-point mathematics and must emulate it with large and slow code libraries. Higher-end devices such as the STM32F4 family of chips have a hardware floating-point unit. (FPU)

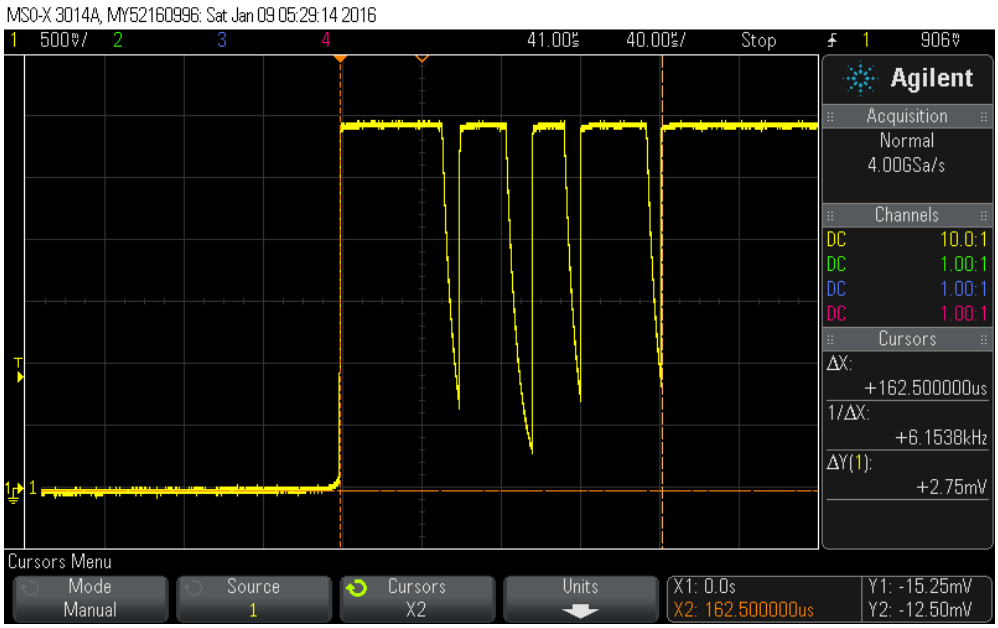


Figure 1.11: Signal bounce on button press.

1.4.7 "Debouncing" External Signals

One complication when working with physical devices (i.e., buttons) is often they have poorly defined transitions. Most mechanical buttons and switches "bounce" when they are pressed; unfortunately, most processors are fast enough to be able to see these bounces as multiple presses! Figure 1.11 shows an oscilloscope trace captured from one of the Discovery board's buttons.

We have multiple ways of dealing with bouncing signals. The Discovery board has circuit footprints available for creating a hardware low-pass filter. However, these pads are not populated with components, and it is necessary to deal with the bouncing signal in software.

One simple way to debounce an input signal is to use a shifted bit vector. A shifted bit vector is an unsigned integer variable which has its lowest bit (rightmost) set to the state of the input signal every iteration through the program loop. As time passes (or each loop iteration) the variable shifts by one to the left. This results in the variable's value representing the input pin's state throughout the bouncing interval. We can reject the bouncing by triggering the code only on steady state values of the variable (for example: 0xFFFF or 0x0000). Figure 1.12 on the next page shows an example debounce routine using a shifted bit-vector.

EXERCISE 1.5 Lab Assignment: Writing Basic I/O Code

These exercises explore two basic operations of the GPIO: Blinking LEDs and reading the state of a pushbutton. After completing these tasks, make sure to show the lab assistant! Most of your points for this lab will come from demonstrating that your code works.

```
uint32_t debouncer = 0;
while(1) {
    debouncer = (debouncer << 1); // Always shift every loop iteration

    if (input_signal) {     // If input signal is set/high
        debouncer |= 0x01;  // Set lowest bit of bit-vector
    }

    if (debouncer == 0xFFFFFFFF) {
        // This code triggers repeatedly when button is steady high!
    }
    if (debouncer == 0x00000000) {
        // This code triggers repeatedly when button is steady low!
    }
    if (debouncer == 0x7FFFFFFF) {
        // This code triggers only once when transitioning to steady high!
    }
    // When button is bouncing the bit-vector value is random since bits are set when
    the button is high and not when it bounces low.
}
```

Figure 1.12: Debouncing method using shifted bit-vector

As mentioned earlier, **you will need to write your applications using *only* peripheral register access.** The single exception to this is the `HAL_Delay()` function.

There are multiple ways to approach these exercises. Many lab assignments begin as empty projects; however, we recommended that for this first application you begin with the provided "blinky" example code and make incremental changes from the HAL library calls to bitwise operations on peripheral registers.

The benefit of converting a working example is that it becomes possible to compile and test the application after replacing each line of code. This breaks down the possible sources of error into manageable portions.

1.5.1 Configuring a GPIO Pin to Output and Blink an LED

Your goal in this lab assignment is to recreate the blinking demo using the red and blue LEDs on the Discovery board.

If beginning with the example "blinky" application, the assignment can be broken down into the following steps:

Converting the HAL Library Calls to Register Access

1. Use the RCC to enable the GPIOC peripheral clock.
   - Remove the `__HAL_RCC_GPIOC_CLK_ENABLE()` HAL library macro.
   - Use the *stm32f072xb.h* header file to determine the register that enables the GPIOC peripheral clock.
   - Set the appropriate bit using bitwise operations and either a bitmask or defined bit name.
2. Configure the LED pins to slow-speed, push-pull output mode without pull-up/down resistors.
   - Remove the `GPIO_InitTypeDef` struct and `HAL_GPIO_Init()` function call.
   - The green and orange LEDs are on pins PC8 and PC9.
   - Set the pins to general-purpose output mode in the MODER register.
   - Set the pins to push-pull output type in the OTYPER register.
   - Set the pins to low speed in the OSPEEDR register.
   - Set to no pull-up/down resistors in the PUPDR register.

   Each register map in the peripheral reference manual documents the starting state of bits in the register. Although clearing bits that should already be zero is not always necessary, it is good style to ensure that every bit is in a known state.
3. Initialize one pin logic high and the other to low.
   - Remove the `HAL_GPIO_WritePin()` function call.
   - Use either the ODR or BSRR register.
4. Toggle both pin output states within the endless loop.
   - Remove the `HAL_GPIO_TogglePin()` function call.
   - Use either the ODR or BSRR register.

5. Leave the HAL delay function within the loop. Otherwise, the LEDs will toggle too rapidly to see. Feel free to change the duration to reasonable values.

Changing to the New LEDs

1. Use the *Hardware Layout* section of the Discovery board manual to find the pins connecting to the red and blue LEDs.
    - Page 13 of the Discovery board manual lists the STM32F0 pins used for LEDs.
    - The red LED is labeled as *LD5* and the blue as *LD6* on the Discovery board.
    - All LED pins are in the GPIOC peripheral.
2. Update the pin initialization and toggle code to use the new pins.

1.5.2 Configuring a GPIO Pin to Input and Reading a Button

Your goal in this portion of the lab assignment is to change your version of the flashing LED program, such that the LEDs toggle on a button press instead of a set delay. This can be broken down into the following steps:

1. Use the *Hardware Layout* section of the Discovery board manual to find the pin connected to the "USER" button.
    - The user button is labeled as *B1* on the Discovery board.
    - Search through and find the material on available buttons.
2. Use the RCC to enable the clock to the appropriate GPIO peripheral.
3. Configure the button pin to input mode with the internal pull-down resistor enabled.
    - Set the pins to input mode in the MODER register.
    - Set the pins to low speed in the OSPEEDR register.
    - Enable the pull-down resistor in the PUPDR register.
4. Monitor the button pin input state within the endless program loop.
    - Use the IDR register.
5. Toggle the LED pins when a button press is detected.
6. Include a software debounce routine to prevent multiple toggles on a single button press.
    - See section 1.4.7 on page 33 for an example debounce routine.

After making these changes to the original application, it is not necessary to have a long duration delay statement each loop iteration. However, some program delay is still required (few milliseconds), otherwise the button debouncing method shown in section 1.4.7 on page 33 will not operate properly.

Some GPIO peripherals contain pins with system critical functions. For example, GPIOA contains pins PA13 (SWDIO) & PA14 (SWCLK) which are used by the debugging hardware to communicate with the STM32F0.

When using GPIO peripherals with special functions, **do not modify the control bits for these pins!** This includes clearing the entire register to ensure that all unknown bit states are set to a known value.

If the configuration for PA13 and PA14 is modified, the debugger will no longer be able to program the Discovery board. If this occurs, it can be fixed using the standalone ST-Link utility.