STM32L0-ARM
Nucleo Board
User's Manual

D.K. Blandford
August 1, 2019
Section 1: Architecture and Introduction

Note: Much of these notes are taken from the book: The Definitive Guide to the ARM Cortex M0 by Joseph Yiu.

The ARM (Advanced RISC Machine) is a 32-bit microcontroller created by a consortium of companies and manufactured in many different versions. It is widely used in modems, cell phones, personal audio, pagers, cameras, and many more embedded high end applications.

ARM is produced by many different companies and each adds its own variations. We will use the ARM Cortex M0 produced by STMicroelectronics as the STM32L031K6.

The ARM microcontrollers are classified as RISC machines where RISC stands for Reduced Instruction Set Computer.

RISC/CISC controversy and history

• In the 1970s as IC manufacturers were able to put more and more on a single chip, computer architects needed to decide what to do with the extra space.
• The prevailing idea was to create more complex instructions which are then executed in microcode. This reduces the use of program memory to fetch instructions and should run faster. This was called CISC for Complex Instruction Set Computer.
• A counter idea, created by IBM, was to use the extra space for things like more registers, cache memory, and numerical processing ability. This solution kept the instruction set very simple so that it took up little extra circuitry. This was called RISC and began about 1980.
• Many new RISC machines were built such as the SUN Sparc. The Intel x86 line represents an architecture that leans toward the CISC side.
• By the mid-1990s it was clear that the ideal machine was neither RISC nor CISC but something in between. Today most machines have features of both.

Generic RISC features:

• A large number of general purpose registers along with compiler technology to optimize register usage.
• A limited and simple instruction set
• An emphasis on optimizing the instruction pipeline
• Load and store architecture with simple addressing modes

Today, most CPU's have some RISC-like features but pure RISC machines are uncommon. The ARM processor has its origins in 1983 from Acorn Computers Ltd. In 1990 Acorn spun off a new company called Advanced RISC Machines Ltd. which licenses its ARM core to other companies for manufacture. Many other companies now make versions of the ARM processor including Intel, Atmel, Texas Instruments, Freescale, and others. The Cortex M0 has fewer features than many of the other processors in the Cortex M series but it is very low cost and has more than enough processing power for most of our applications.
Feature Summary ARM Cortex M0 core [5]

- 32-bit RISC processor (32-bit data & address bus)
- Big and Little Endian operating modes
- High performance RISC
- Low power consumption
- Fully static operation
- Fast interrupt response for real-time applications
- Excellent high-level language support
- Simple but powerful instruction set

STM32L031K6 FEATURES [7]

Processor
ARM Cortex-M0+ processor, running at 32 MHz.
STM32L031K6T6

Interrupts
ARM Cortex-M0 built-in Nested Vectored Interrupt Controller (NVIC).
Non-Maskable Interrupt (NMI).

Memory:
32 kB on-chip flash programming memory.
8 kB SRAM.
1 kB EEPROM

Programming
In-System Programming (ISP) and In-Application Programming (IAP) via on-chip bootloader software. Flash is programmed with ST-Link via USB in Keil.

Digital peripherals:
GPIO pins can be used as edge and level sensitive interrupt sources.
High-current output driver (16 to 22 mA) on one pin.
High-current sink drivers (16 to 22 mA)

Timers
Timer 2 is a 16-bit counter/timers with a 16-bit prescaler. It has 4 channels and can generate a DMA request and do PWM.
Timer 21 and 22 are 16-bit counter/timers with a 16-bit prescaler. Each has 2 channels. These cannot generate a DMA request but both can do PWM.
Low Power timer is a 16-bit counter/timer with a 3-bit prescaler. If this is clocked by LSE or LSI or by an external clock it can continue running in stop mode and used as a wake-up timer.
Watchdog timer is a 12-bit down counter with an 8-bit prescaler. It is clocked from the 37 KHz internal RC source.

Analog peripherals:
12-bit ADC with input multiplexing to 10 channels depending on package size. There is no D/A Converter but it does support PWM.

Serial interfaces:
UART and USART with fractional baud rate generation, internal FIFO, and RS-485 support.
SPI controller with SSP features and with FIFO and multi-protocapabilities
I²C-bus interface supporting full I²C-bus specification and Fast-mode Plus with a data rate of 1 Mbit/s with multiple address recognition and monitor mode.

Clock generation:
There are three different clock sources that can drive the master clock SYSCLK:
• 1-25 MHz high-speed external (HSE), that can supply a PLL (This is not available on the Nucleo board we have but you can add your own external crystal.)
• 16 MHz high-speed internal RC oscillator (HSI), trimmable by software, that can supply a PLL
• Multispeed internal RC oscillator (MSI), trimmable by software, able to generate 7 frequencies (65 kHz, 131 kHz, 262 kHz, 524 kHz, 1.05 MHz, 2.1 MHz, 4.2 MHz).

Power Supply
The board is powered through the PC via the USB +5 volt source. You can also power the board by an external 5 volt source on pin 4 on CN4 on the board. Alternatively, you can power the board with a 3.3 volt external source on pin 14 on CN4 but if you do this you must set the solder bridges 9 and 14 to off. You can also supply power to Vin on pin 1 of CN4 but this is before the regulator and must in the range of 7 to 12 volts.

Integrated PMU (Power Management Unit) to minimize power consumption during Sleep, Deep-sleep, and Deep power-down modes. The chip has power on reset and brownout detection. The chip itself operates from a single power supply of 1.8v to 3.6 v.
Section 2: Using Keil μVision 5

Download the student version of μVision 5 from https://www.keil.com/download/product/. The student version is free but you must sign up for a license. It is limited to 32K of code which is adequate for this class. Note that the GNU compiler is free and has no size limitations but in terms of speed and space it does not fare well against the Keil compiler and there is no simulator.

The Keil compiler does not contain the necessary start up files and the configuration files for every processor. As a result you must use the packet installer to get the proper files from Keil for the processor you are using. The easiest way to do this is to load a project that has already been compiled that uses the processor you are interested in. To do this download STM32L0Template folder from the website at https://csserver.evansville.edu/~blandfor/EE354/index.htm

Open this folder and open the folder MDK-ARM. In that folder you will see the following files.

![Figure 2.1 Files in the STM32L0Template/MDK-ARM folder](image)

Double click on STM32L0Template.uvprojx. Assuming that you have installed μVision 5 this project will open with the packet installer and the proper configuration files will be loaded for the STM32L0031K6T6 processor. In addition the template file will load the debug link that will allow you to download the binary file to the board. Loading the needed packets may take several minutes.

If you are using a different version of μVision 5 than was used to create the template you may get an error that says the "loading of the PDSC Debug file failed". If this happens, close the project, open the STM32L0Template folder and delete the folder that has the name DebugConfig. After deleting this folder close the template, open μVision5 and use the Project→Open Project menu to reopen STM32L0Template.uvprojx. If you still have an error see Jeff or your instructor.

After successfully loading the STM32L0Template project in μVision 5 you should see a screen that looks like that in Figure 2.2.

In the Project Files window double click on main.c to see that file in the Code Window. Click on Project → Build Target to compile the program. If you have been living a virtuous life you should see the message below in the Output window.

"*** Using Compiler 'V5.06 update 6 (build 750)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Build target 'STM32L0Template'
"STM32L0Template\STM32L0Template.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:01"
Figure 2.2 µVision 5 opening screen with successful loading of STM32L0Template project

Loading the project on the Nucleo Board
1. You will need a USB cable that has a Micro-B connector for the board and a standard USB connector to plug into your computer. This is the same cable that many phones use for charging or see Jeff if you don’t have a cable. Use the cable to connect the board to your computer. If your computer does not locate the USB driver you will have to locate the driver and install it. The driver is at c:\Keil\ARM\STLink\USBDriver if you need it. If after installing the USB driver you may also need the upgrade utility which is at c:\Keil\ARM\STLink.

2. We will be using STLink to download the project to the board. STLink should be set up for you as part of the STM32L0Template project. Click on the Debug button or, click on Debug → Start/Stop Debug Session. The project will be downloaded to the board and your screen will switch to the Debug screen similar to that shown in Figure 2.3.

3. At this point your code has been downloaded but it is not running. Click on Debug → Run. You can use an oscilloscope to look at any of the pins on Port B (see Figure 5.4) which will have a square wave running at about 4 MHz. The program is running out of flash memory which is non-volatile so you can quit debugging (Debug → Start/Stop the Debugger) and the program will continue to run. You can also unplug the USB cable. When you plug it in again the system will reset and the program will start again.
Note that if you use the template for new programs, you will not have to set up the options for STlink. Details of those options are Appendix A.

You can alter the code by exiting the debugger and adding an empty delay loop like this: Rebuild and download the program after a successful recompile.

```c
int main(void)
{
    SystemClock_Config();
    int tmp, i;               //Declare i for delay loop
    //Clock bits
    RCC_IOPENR |= 2;          //Bit 1 is GPIOB clock enable bit
    //I/O bits PB2
    GPIOB_OSPEEDER = 0xC;     //PB2 to high speed
    GPIOB_MODER = 0x55555;    //Set first 10 port bits to output
    GPIOB_PUPDR &= 0xFFFFF000; //No pull up and no pull down
    tmp = 0;
    while(1)
    {
        {GPIOB_ODR = tmp;        //Inverts 10 bits on Port B
            tmp = ~tmp;
            for(i=0;i<1000;i++);    //Add this delay loop
        }
    }
}
```

In this case the square wave on the Port B pins runs at about 267 KHz.
If while debugging you find an error it is tempting to change the code in the debugger to fix the problem. Such changes will not be effective. If you make any changes to the code they will not go into effect until you have again recompiled the program.

For future projects, in order not to have to run through the tedious STLink set up, you can create a new project folder on your computer by downloading STM32L0Template.zip, unzip it, and rename the folder. Also rename the main c-file (named STM32L0Template.c). Open the project by double clicking on STM32L0Template.uvprojx. An error message will indicate that it could not find the file STM32L0Template.c since you renamed it. Ignore the error message. In the Project File Window (see Figure 2.3) click on the file STM32L0Template.c and delete it from your project. Right click on Application User Group and select Add existing file to the group. Locate your renamed .c file and add it to the project. After it is added double click on it to see it in the Code Window and alter the c code as needed.
Section 3 Quirks, clocks, and details

Naming Conventions
STMicroelectronics uses the naming convention shown in Figure 3.1 for microcontrollers in this series.

Example:
STM32 L 031 K 6 T 6 D TR

Device family
STM32 = Arm-based 32-bit microcontroller

Product type
L = Low power

Device subfamily
031 = Access line

Pin count
C = 48 pins
K = 32 pins
G = 28 pins
E = 25 pins
F = 20 pins

Flash memory size
4 = 16 Kbytes
6 = 32 Kbytes

Package
T = LQFP
U = UFQFPN
Y = WLCSP
P = TSSOP

Temperature range
6 = Industrial temperature range, −40 to 85 °C
7 = Industrial temperature range, −40 to 105 °C
3 = Industrial temperature range, −40 to 125 °C

Number of UFQFPN28 power pairs
S = one power pair\(^1\)
No character = Two power pairs

Options
No character = \(V_{DD}\) range: 1.8 to 3.6 V and BOR enabled
D = \(V_{DD}\) range: 1.65 to 3.6 V and BOR disabled

Packing
TR = tape and reel
No character = tray or tube

1. This option is available only on STM32L031GxUxS part number. Contact your nearest ST sales office for availability.

Figure 3.1 Part naming conventions[5]
Architectural Features and Quirks
In order to get a microcontroller to run at 32 MHz using relatively slow flash memory it is necessary to have some architectural creativity. For the STM32 M0+ processor this includes a phase locked loop, an Advanced Peripheral Bus (APB) and an Advanced High-speed Bus (AHB). All of these things run from a rather complex system clock. The following discussion gives an overview of how this works.

Bus System
There are two main busses on the Cortex M0+: AHB Lite and the APB. The AHB is the high speed bus that services the processor and memory and the APB is the peripheral bus which services all of the peripheral items. The two busses are connected by a bus bridge as shown in Figure 3.2. Each bus has its own clock and the two can run at different speeds.

System Clock
The STM32L0 has four clock sources which can be used to provide the system clock. The sources are: the High-Speed External (HSE) oscillator, the High-Speed Internal (HSI16) Oscillator, or the Phase Locked Loop clock (PLL) and the multi-speed internal clock (MSI). The HSE clock is driven by an external oscillator which is not present on the board that we are using. This allows several processors to run from a single external clock. The HSI clock is driven by a 16 MHz internal RC oscillator. Its frequency is approximate but may be calibrated by the user. The MSI clock at 2.1 MHz is used to start the processor after a power on reset. There are also two low speed clock sources: one at 37 KHz from and RC internal oscillator and one at 32768 Hz driven by an external crystal (not on the board we are using). These two low speed clocks can drive a real time clock or the watchdog timer but they cannot drive the system clock.
The HSI clock is driven by an internal RC oscillator so it is not particularly accurate. It is low cost and a software calibration procedure is provided. The HSI clock must be selected by software after the processor starts.

Figure 3.3 The clock configuration logic for the Cortex M0+.
**Phase Locked Loop Clock**

Internally the STM version of the ARM Cortex M0+ processor can run at up to 32 MHz. This would imply that a very fast external interface bus would be needed to communicate with the outside world. For many embedded controller systems this high speed interface is unnecessary. The solution to this problem is to add a phase locked loop into the hardware which allows the CPU to run at 32 MHz and to simultaneously communicate with a synchronized system on the outside at a lower speed. The PLL is a frequency multiplier. Thus, the STM32 M0+ can use any of the clock sources along with the PLL to run at 32 MHz.

A system diagram for a phase locked loop (PLL) is shown below. An 16 MHz internal RC oscillator runs the external system busses and clocks. Internally a voltage controlled oscillator runs at about 32 MHz. Figure 3.4 shows how this is done. The 16 MHz oscillator is first divided by 16 to get a stable 1 MHz clock. The 1 MHz clock from the crystal is compared to the 1 MHz signal from the divided voltage controlled oscillator. The error signal is fed back to adjust the frequency of the VCO. Effectively, the PLL acts as a frequency multiplier.

![System diagram for a phase locked loop.](image)

**Peripheral Clock**

The peripheral clock for the GPIO ports, the A/D and D/A converters, the timers, and the high speed busses are derived from the system clock by way of a pre-scaler which can be modified by the user.

Several pre-scalers are used to configure the Advanced High-speed Bus (AHB) frequency, the Advanced Peripheral Busses: high-speed APB (APB2) and low-speed APB (APB1). The maximum frequency of the AHB is 32 MHz. The maximum allowed frequency of the high-speed APB2 is 32 MHz. The maximum allowed frequency of the low-speed APB1 is 32 MHz.

**Barrel Shifter**

A barrel shifter is a combinational logic block which uses multiplexors to shift bit positions as data is copied from one location to another. For a normal shift register it takes \( n \) clock cycles to shift \( n \) places left or right. The barrel shifter on the Cortex-M0 makes it possible to do up to a 32-bit shift left or right in a single clock cycle.
**Processor Block Diagram**

A block diagram of the STM32L031x6 processor is shown in Figure 3.5. For the Nucleo board the STM32L031x6 has 32 KB of flash memory and 8 K of RAM plus 1 K of EEPROM.

![Block diagram of STM32L031x6 processor](MSV9545EV2)

**Figure 3.5** Block diagram of STM32L031x6 processor.[5]
Section 4 Peripherals
The STM32L031 has a number of built in peripherals. In this section we take a closer look at some of these.

Timers
The STM32L0031 has three general purpose timers (TIM2, TIM21, and TIM22), one low power time (LPTIM1), an independent watchdog timer (IWDG), and a window watchdog timer (WWDG). We will not be using the watchdog timers in this class.

The low power timer 1 is a 16-bit counter that has a 3-bit prescaler and an auto-reload register. It can be clocked from the LSI clock and can run through all modes including deep sleep and standby. It is typically used as a wakeup timer.

Timers 2, 21, and 22 are general purpose timers. All are 16-bit up/down counters with auto reload. Timer 2 has four channels available for PWM and timers 21 and 22 have just 2 channels available for PWM. Timer 2 can also sponsor a DMA interrupt. Figure 4.1 shows a logic diagram for timer 2.

Figure 4.1 Timer 2 logic diagram. Timers 21 and 22 are similar but with only two output channels.
Examples 2C and 3C in C-examples show how to use the timers by polling or by generating an interrupt.

**Interrupts**
The ARM processor has a vectored interrupt system. Each interrupt is assigned a number which can be translated to an address by say multiplying the number by 4. At this address a second address is stored which is the address of the interrupt service routine. For example if a timer produces an interrupt numbered 20, the system would multiply this by 4 to get 80. At location 80 in memory we would find the actual address of the interrupt service routine. If you are writing in assembly code you need to place the vector in the table yourself. If you are writing in C the compiler will do it for you.

To set up an interrupt you need to know the interrupt number. Interrupt numbers are given in Table 52 on page 258 of the reference manual [1]. For example Timer 2 is at interrupt number 15. The interrupt must be enabled by setting a bit in the set enable register (there are two of these called NVICSER0 and NVICSER1 which together hold 64 bits to enable interrupts.)

Interrupt 15 could be enabled using the following:

```
NVICSER0 |= (1 << 15);
```

(There are also two clear enable registers NVISCER0 and NVISCER1 to disable an interrupt).

To get the interrupt vector in place in c-code you need only write the interrupt service routine using a specific name for the interrupt you want. For Timer 2 for example we would write the following:

```c
void TIM2_IRQHandler()
{
    //Do something useful
    TIM2_SR &= 0xFFFF; //Turn off the interrupt
}
```

The c-compiler seeing the IRQHandler notation would put the vector in place – in this case at location 4 x 15 = 60 in memory.

Interrupts can be generated by such things as timers, the USART, I2C as well as by error exceptions and by external events connected to pins coming off the chip.

Example 3C in C-Examples shows how to use a timer to generate an interrupt.

**UART and USART**
The term USART stands for Universal Synchronous/Asynchronous Receiver/Transmitter and it does the logic functions to make serial communications easier. If a clock signal is sent separately from the data the communication is termed synchronous. If the clock is embedded in the data the communication is termed asynchronous.

Effectively, a UART provides an 8-bit parallel interface suitable for a microcontroller bus, to a serial communications channel. The UART can take 8-bits of data from a microcontroller, convert it to a serial data stream, and send it out over a single line at a preprogrammed bit rate. It
can simultaneously, in full duplex mode, receive a serial stream, at the same or different bit rate, and convert it to 8-bit parallel for input to a microcontroller bus.

The STM32L0 has a low power UART and a separate USART (which may be programmed to be a UART). A block diagram for the STM32L0 USART is shown in Figure 4.2.

![Figure 4.2 The USART on the STM32 L0 processor.](image)

The lines at the top of the diagram are the parallel read and write bus. This is 8-bits wide. The serial lines are on the left side and are marked TX (transmit) and RX (Receive). For synchronous communication the clock line is shown on the right side. The lines on the left side
marked RTS (Ready To Send) and CTS (Clear To Send) are handshaking lines that are often ignored. The clock line marked \( f_{ck} \) at the bottom drives the system and may come from several sources (see [1] p. 654) For this class we will generally use the peripheral clock as \( f_{ck} \).

Figure 4.3 shows how serial data is transmitted asynchronously. If no data is being transmitted the TX line is held high. At the beginning of transmission the TX line goes low for one-bit time. This is called the start bit. The start bit is followed by 8 data bits and a stop bit. The stop bit is always high. Data is transmitted with the least significant bit going out first. Traditionally, a start bit, 7 ASCII data bits, a parity bit, and a stop bit make up a 10-bit character. Most UARTs allow the user to use two stop bits and ignore the parity bit but two stop bits are seldom used and the parity bit is often ignored. (The parity bit is used to detect an odd number of errors in transmission. If say even parity is used the UART sets the parity bit to either one or zero so that the number of ones transmitted is even.)

### Asynchronous Mode

![Asynchronous Mode Diagram](image)

**Figure 4.3** Transmission and reception of serial data.

The rate at which data is transmitted is called the baud rate (created by Émile Baudot in the 1920s). The baud rate is defined as the number of symbols sent per second. Since we are using binary data with only two symbols the baud rate and the bit rate are the same. For asynchronous serial communication the transmitter and the receiver must agree on what the baud rate will be and the two have independent clocks running at that rate to either send or receive the data. Normally, the receiver would check the serial line to determine when it goes low (a start bit). After half a bit time it would start its clock and clock in the data and the stop bit. If the transmission rate is not within a few percent of the receiving rate the receiver may clock in a zero for a stop bit which is called a framing error.

Baud rates have been somewhat standardized over the years. The commonly used baud rates are 1200, 2400, 4800, 9600, 4800, 19200, 38400, 57600, and 115200. If we use one start bit, one stop bit, and 8 data bits, there are 10-bits per character so the character rate is the baud rate/10 so that 9600 baud would send and receive data at 960 characters/second.

Oversampling
To improve clock accuracy between the sender and receiver the UART on the ARM uses oversampling. In this case the receiver clock may run at 8 or 16 times as fast as the bit rate so that there are multiple clock signals per bit. Some additional circuitry uses a majority vote of clocked data to get the actual data.

Setting the baud rate
The USART on the STM32L0 has a baud rate register (BRR) which must be loaded with the clock divider to get the baud rate. The clock divider number is determined in part by the amount of oversampling. There is a bit named Over8 in USART_CR1 which determines the amount of oversampling. The bit defaults to zero at reset which gives oversampling by 16. In this case the number loaded into BRR is the baud rate divisor. For example, suppose we are clocking the system at 32 MHz and we want a baud rate of 9600. We can write $32000000 / \text{BRR} = 9600$. In this case the BRR is 3333.33. Since this must be an integer we round it to 3333. Loading the BRR with 3333 will produce a baud rate of 9600 if the system clock is at 32 MHz. For using 8x oversampling see the reference manual [1] on pp. 665-666.

Example 6C in C-examples shows how to set up the USART to write characters to a serial line.

**Pulse Width Modulation**
Many of the ARM Cortex processors support an onboard D/A converter. The ARM Cortex M0+ does not. (There is an A/D but no D/A). Digital to analog data conversion can be done by filtering a signal which is pulse width modulated so that all frequencies at or above the base frequency are cutoff.

In PWM we start with a base frequency of say 1,000Hz and each cycle we output a single pulse whose width varies from 0% of the period to 100% of the period.

![Pulse width modulation](image)

**Figure 4.4** Pulse width modulation. In this case the signal is 50% of maximum.

The base frequency of the PWM is $f = 1/T$ so that the frequency content of a PWM signal will have frequencies at $f$ and higher due to the base frequency. The signal information is transmitted by the duty cycle and will be, generally, a much lower frequency signal. Thus a relatively simple low pass filter can eliminate the frequencies at the base frequency and above and recover an analog signal from the PWM.

A timer diagram which supports PWM is shown in Figure 4.1. The ARM Cortex M0+ Timers 2, 21, and 22 all support PWM. The base frequency is determined by the clock frequency and the number loaded into the ARR (Auto reload register). For example if the clock frequency is 32 MHz and the ARR is loaded with the number 32000, the base frequency will be $f_{\text{base}} = f_{\text{ck}} / \text{ARR} = 32000000 / 32000 = 1000$ Hz.
The duty cycle will be determined by the number loaded in the CCR (Capture and Control Register). For example if the ARR = 32000 and the CCR = 16000 the duty cycle will be 50%.

Example 4C shows how to generate a PWM signal from a ramp function and example 5C shows how to generate a PWM signal from the A/D converter.

**A/D Conversion**
The analog to digital converter on the STM32LM0+ is a 12-bit converter that can have up to 10 external channels of input and 3 internal channels (all passing through a 16:1 analog multiplexor to the single converter). On the board that we have ADC channel 0 to 7 connected to PA0 to PA7, channel 8 and 9 are connected to PB0 and PB1.

The ADC can run in a one-time conversion mode or, it can run in a sequence mode in which a sequence of channels is triggered sequentially and the data is sent directly to memory via DMA.

The ADC uses the successive approximation algorithm. This algorithm is akin to doing division when you are given only a multiplier. Figure 4.5 shows such a scheme.

![Figure 4.5](image)

In Figure 4.5, take the counter to be started at 0 so the output of the D to A converter is zero. The D to A output goes to a comparator which compares it to the analog input. If the comparator finds that the analog input is higher than the D to A input it outputs a 1 allowing the counter to be clocked to a higher number. A new D to A conversion is performed and this process continues until such time as the D to A output is just above the Analog input at which time the comparator output goes to zero and the counting stops.

In an SAR converter the counting logic is replaced by an SAR register. This register and hardware effectively do a binary search for the answer by guessing from the most significant bit to the least significant bit. This process guarantees that for n-bit resolution we need only guess n times at the result.

Most microcontrollers that do A/D conversion require the following steps
- Set up the A/D clock. The A/D typically runs slower than the microcontroller and require its own clock.
- Select the channel you want to convert on the analog multiplexor.
- Set up the port pins for analog input
- Start the converter
- Wait for the conversion to finish.
- Read the data from the data register.

For A/D conversion on the ARM we need to do the following:
- Enable the A/D clock
- Enable the port clock
- Set the port input bit to alternate function analog
- Turn off pull up and pull down on port pin
- Turn the ADC on
- Enable end of conversion to be set on done
- Set up channel number for conversion
- Start the A/D
- Wait for EOC bit
- Recover data from ADC Data register – 12 bits
- To repeat go back to Start the A/D

Figure 4.6 The ARM ADC

Example 5C shows how to use the ADC as input and a PWM channel as output.
Section 5 Programming Details

Assembly Language Programmer's Model

Figure 5.1 shows the internal registers accessible to the assembly language programmer. The top three registers have special functions: R13 is the stack pointer; R14 is the link register; R15 is the program counter.

The normal 32-bit ARM instruction can access all of the registers. The Thumb and most of the Thumb2 instructions are only 16-bit instructions and have access only to R0-R7 as general purpose registers. For the Thumb instructions only the MOV instruction has access to all of the general purpose registers.

The link register R14 is a special one-level stack. When a subprogram call is made from a main program the return address goes into R14. If the subprogram makes further calls the return address goes onto the stack in memory. The assembly language instruction BLX (Branch and Link) automatically uses this register.

Register R13 is the stack pointer. The lower two bits of this register is always zero since it accesses the stack on 32-bit word boundaries. Normal operations always use SP_main but for thread mode the processor can be configured to use SP_process.

Register R15 is the program counter. It is 32-bits long and there is a linear address space from 0 to $2^{32}-1$. Bit 0 of this register is always 0 so that instructions are aligned on 16-bit boundaries.
The CPU flags are stored in the program status register. There are three program status registers called Application Program Status Register (APSR), Interrupt Program Status Register (IPSR), and the Execution Program Status Register (EPSR). These three status registers have mutually exclusive bit fields and can be combined and read as a single register called the Program Status Register (PSR). Figure 5.2 shows these registers.

![Figure 5.2](image)

**Figure 5.2** The three program status registers can be read a single register called PSR with bit fields as shown.

Bits 27 to 31 are the condition codes or CPU flags. The definition for these is shown in Figure 5.3.

![Figure 5.3](image)

**Figure 5.3** Definition of the condition code bits[2].

In addition to the normal CPU flags in Figure 5.3, the Program Status Register has three other fields with status information. These are the Greater than or Equal (GE) field, the Interrupt Continuable/If Then (ICI/IT) field, and the Interrupt Service Routine (ISR) number. The GE field holds information that can be used by the SEL and SIMD assembly language instruction for certain conditions. The ICI/IT field holds information for conditional execution (if-then)
assembly language instructions. The ISR number field holds the exception number currently being executed.

**Memory Mapping**
The ARM has a 32-bit address and in hex addresses range from 0000 0000 to FFFF FFFF. This is a linear address space and there are no I/O instructions since all ports are mapped from memory. Figure 5.4 gives an overview of the memory mapping. Refer to the User's manual for the address of specific peripheral registers.

**Memory map details**
All of the I/O ports are mapped into the 4 GByte memory space. From Figure 5.4 we see that the peripherals are mapped to the address range 0x4000 0000 to 0x5FFF FFFF. Table 4 gives the addresses of the most common registers for the I/O devices. This list is incomplete. See Table 3 in the STM32 Reference Manual [1] pp. 51-54 for a more complete list of register boundary addresses and links to specific register addresses. Addresses for the interrupt system are not in the Reference Manual. You can find these in the Programmer's Manual [2].

![Figure 5.4 Memory map for the ARM M0+ processor][1]

Note: AHB – Advanced High-performance Bus, APB – Advanced Peripheral Bus, FSMC

**Figure 5.4** Memory map for the ARM M0+ processor.[1]
Pin Assignments

The microcontroller on the Nucleo L0 board is an STM32L031K6T6 which comes in a 32 pin Low profile Quad Flatpack (LQFP) shown in Figure 5.5.

LQFP32 pin package and board pin assignments

Figure 5.5 Pin assignments for the STM32L031K6T6 chip and board. [3][5]

The Nucleo board has two connectors each of which has 15 pins number 1 to 15 on each side in Figure 5.4 right. These pins protrude through the bottom and the top of the board at one-tenth of an inch spacing so that you can plug the board directly into a breadboard. Only 30 of the 32 pines on the chip are brought out to the side connectors. Pins 23 and 24 which correspond to PA13 and PA14 are not brought out. Table 1 gives the pins that are available on this board.

Table 1

<table>
<thead>
<tr>
<th>Port</th>
<th>Pins on Nucleo Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA</td>
<td>PA0 – PA12</td>
</tr>
<tr>
<td>PB</td>
<td>PB0, PB1, PB3-PB7</td>
</tr>
<tr>
<td>PC</td>
<td>PC14, PC15</td>
</tr>
<tr>
<td>+5</td>
<td>CN4-P4</td>
</tr>
<tr>
<td>+3.3</td>
<td>CN4-P14</td>
</tr>
<tr>
<td>Gnd</td>
<td>CN3 P4, CN4 P2</td>
</tr>
<tr>
<td>NRST</td>
<td>CN3 P3, CN4 P3</td>
</tr>
<tr>
<td>AREF</td>
<td>CN4 P14</td>
</tr>
<tr>
<td>VIN</td>
<td>CN4 P1</td>
</tr>
</tbody>
</table>

NRST is the chip reset, AREF is the Analog Reference voltage for the A/D Converter, VIN is an external power supply pin.

A green LED is connected to Pin 15 on CN5. It is active high. An active low reset switch is on the end of the board opposite the USB connector.
As is typical, many of the pins have alternate functions. You can find the list for these on the Data Sheet [5] in Table 15 pp. 39-45.

**GPIO Pins**

Each of the GPIO ports on the ARM Cortex M0 processor has ten 32-bit registers that are used to configure the port and get information in and out. Table 2 defines these registers. Figure 5.6 shows a schematic of typical GPIO Pin circuitry.

**Table 2**

<table>
<thead>
<tr>
<th>Reg name</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIOx_MODER</td>
<td>Mode register</td>
<td>Input, output, alternate function(^1), or analog</td>
</tr>
<tr>
<td>GPIOx_OTYPER</td>
<td>Type register</td>
<td>Allows push/pull or open drain</td>
</tr>
<tr>
<td>GPIOx_OSPEEDR</td>
<td>Speed register</td>
<td>Allows low, medium, fast or high speed</td>
</tr>
<tr>
<td>GPIOx_PUPDR</td>
<td>Pull up/Pull down register</td>
<td>Allows pull up, pull down, or no pull up or pull down</td>
</tr>
<tr>
<td>GPIOx_IDR</td>
<td>Input data register</td>
<td>Stores data for reading from outside</td>
</tr>
<tr>
<td>GPIOx_ODR</td>
<td>Output data register</td>
<td>Stores data for writing to outside</td>
</tr>
<tr>
<td>GPIOx_BSRR</td>
<td>Bit Set/Reset register</td>
<td>Writing a 1 to a bit in this register allows the user to set or reset a bit in the ODR</td>
</tr>
<tr>
<td>GPIOx_LCKR</td>
<td>Lock register</td>
<td>Allows the user to lock a pin so that its value cannot be changed.</td>
</tr>
<tr>
<td>GPIOx_AFRH</td>
<td>Alt function register high</td>
<td></td>
</tr>
<tr>
<td>GPIOx_AFRL</td>
<td>Alt function register low</td>
<td></td>
</tr>
</tbody>
</table>

Note 1: alternate functions include timers, UARTs, I2C, and SPI to be attached to a port pin.

![Figure 5.6. Typical GPIO Pin circuitry.](image)
Note that on input most GPIO pins default to analog – not input. GPIO pins are also synchronous and as such, have a clock which must be enable before they are used.

Notice that on the output control there is a P-MOS/N-MOS push pull circuit which can be turned on or off using the OTYPER register. There is also a pull up/pull down configuration on the I/O line at the pin. This is set up by the PUPDR register. The protection diodes allow all of the I/O pins to be 5 volt tolerant. The input has a Schmitt trigger which is bypassed for analog input.

Each GPIO pin can source or sink up to 16 ma [5]. However, care must be taken so that the total current going into or out of the chip does not exceed 105 ma. Exceeding this value may result in overheating. Some pins marked FT or FTf are five volt tolerant and can sink up to 22 ma. See the Data Sheet[5] table 15 p. 42.

Addressing ports – Include files

Assembly code
The ARM processor is mostly a RISC machine with a load and store architecture. Since all of the ports are memory mapped, reading/writing to a port is the same as reading/writing to a memory location. For example the following assembly code reads GPIOA to a register:

;From Table 1 the address of GPIOA_IDR is 0x50000010
ldr r4, =0x50000010; address of Port A
ldr r3, [r4]; Copy port to r3

To copy data to GPIOA we could use the following:

ldr r4, =0x50000014; address of Port A output data register
str r3, [r4]; Store r3 in port A

Notice that the load instruction has data flowing from right to left but the store instruction has data flowing from left to right. In both cases, the square brackets around r4 indicate that it holds a register indirect address. That is, R4 holds the memory address of the data – not the data itself.

C-Code traditional style
In traditional C-code we create an include file which allows us to define register names for the register memory addresses. Typical define statements in such an include file for the GPIOA input and output data registers might look like this:

#define GPIOA_IDR (*((volatile unsigned long *) 0x50000010)) //GPIO A Input Data reg
#define GPIOA_ODR (*((volatile unsigned long *) 0x50000014)) //GPIO A Output Data reg

In the line for GPIOA_IDR the code
(volatile unsigned long *) 0x50000010
does a cast making the hexadecimal number 0x50000010 a pointer type. The * in front of this dereferences this pointer allowing us to get the data that address. In C-code we can write:

GPIOA_ODR |= 0x00000080; //Make bit 7 a 1
OR we could write:

GPIOA_ODR |= (1 << 7); //Also makes bit 7 a 1

The file stm32L031.h is an include file which has most of the registers for the ARM Cortex defined in this manner. This file is available in the template program for the Keil V5 programming environment on the course web site. Table 3 gives an example of the items in the stm32L031.h files
Table 4  
An abbreviated list of memory addresses for many I/O ports.

<table>
<thead>
<tr>
<th>Port A Addresses</th>
<th>ADC Addresses</th>
<th>Interrupt controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIOA_MODER</td>
<td>ADC1_ISR</td>
<td>NVICISER0</td>
</tr>
<tr>
<td>0x50000000</td>
<td>0x40012400</td>
<td>0xE000E100</td>
</tr>
<tr>
<td>GPIOA_PUPDR</td>
<td>ADC1_IER</td>
<td>NVICISER1</td>
</tr>
<tr>
<td>0x50000004</td>
<td>0x40012404</td>
<td>0xE000E104</td>
</tr>
<tr>
<td>GPIOA_OSPEEDE</td>
<td>ADC1_CR v</td>
<td>NVICISER0</td>
</tr>
<tr>
<td>0x50000008</td>
<td>0x40012408</td>
<td>0xE000E180</td>
</tr>
<tr>
<td>GPIOA_OTYPER</td>
<td>ADC1_CFGR1</td>
<td>NVICISER1</td>
</tr>
<tr>
<td>0x5000000C</td>
<td>0x4001240C</td>
<td>0xE000E184</td>
</tr>
<tr>
<td>GPIOA_IDR</td>
<td>ADC1_CGR2</td>
<td>USART Registers</td>
</tr>
<tr>
<td>0x50000100</td>
<td>0x40012410</td>
<td></td>
</tr>
<tr>
<td>GPIOA_ODR</td>
<td>ADC1_DR v</td>
<td>USART1_SR</td>
</tr>
<tr>
<td>0x50000114</td>
<td>0x40012440</td>
<td>0x40011000</td>
</tr>
<tr>
<td>GPIOA_BSRR</td>
<td>ADC1_CHSELR</td>
<td>USART1_DR</td>
</tr>
<tr>
<td>0x50000118</td>
<td>0x40012428</td>
<td>0x40011004</td>
</tr>
<tr>
<td>GPIOA_AFRL</td>
<td>ADC1_SMPR</td>
<td>USART1_BRR</td>
</tr>
<tr>
<td>0x50000200</td>
<td>0x40012414</td>
<td>0x40011008</td>
</tr>
<tr>
<td>GPIOA_AFRH</td>
<td>Common ADC Addresses</td>
<td>USART1_CR1</td>
</tr>
<tr>
<td>0x50000204</td>
<td>0x40012428</td>
<td>0x4001100C</td>
</tr>
<tr>
<td>Port B Address</td>
<td>ADC_CCR v</td>
<td>USART1_CR2</td>
</tr>
<tr>
<td>0x50000400</td>
<td>0x40012708</td>
<td>0x40011010</td>
</tr>
<tr>
<td>GPIOB_MODER</td>
<td>TIM2 CR1</td>
<td>USART1_GTPR</td>
</tr>
<tr>
<td>0x50000404</td>
<td>0x40000000</td>
<td>0x40011018</td>
</tr>
<tr>
<td>GPIOB_PUPDR</td>
<td>TIM2 CR2</td>
<td>0x40000004</td>
</tr>
<tr>
<td>0x50000408</td>
<td>0x40000004</td>
<td></td>
</tr>
<tr>
<td>GPIOB_OTYPER</td>
<td>TIM2 SR</td>
<td>I2C1_CR1</td>
</tr>
<tr>
<td>0x5000040C</td>
<td>0x40000010</td>
<td>0x40005400</td>
</tr>
<tr>
<td>GPIOB_IDR</td>
<td>TIM2_DIER</td>
<td>I2C1_CR2</td>
</tr>
<tr>
<td>0x50000410</td>
<td>0x4000000C</td>
<td>0x40005404</td>
</tr>
<tr>
<td>GPIOB_ODR</td>
<td>TIM2 EGR</td>
<td>I2C1_OAR1</td>
</tr>
<tr>
<td>0x50000414</td>
<td>0x40000014</td>
<td>0x40005408</td>
</tr>
<tr>
<td>GPIOB_BSRR</td>
<td>TIM2_CCMR1</td>
<td>I2C1_OAR2</td>
</tr>
<tr>
<td>0x50000418</td>
<td>0x40000018</td>
<td>0x4000540C</td>
</tr>
<tr>
<td>GPIOB_AFRL</td>
<td>TIM2_CCR</td>
<td>I2C1 DR</td>
</tr>
<tr>
<td>0x50000420</td>
<td>0x40000020</td>
<td>0x40005410</td>
</tr>
<tr>
<td>GPIOB_AFRH</td>
<td>TIM2_CNT</td>
<td>I2C1 SR1</td>
</tr>
<tr>
<td>0x50000424</td>
<td>0x40000024</td>
<td>0x40005414</td>
</tr>
<tr>
<td>Clock addresses</td>
<td>TIM2_PSC</td>
<td>I2C1 SR2</td>
</tr>
<tr>
<td>0x40021000</td>
<td>0x40000028</td>
<td>0x40005418</td>
</tr>
<tr>
<td>RCC_CR</td>
<td>TIM2 ARR</td>
<td>I2C1_CCR</td>
</tr>
<tr>
<td>0x40021000</td>
<td>0x4000002C</td>
<td>0x4000541C</td>
</tr>
<tr>
<td>RCC_ICSCR</td>
<td>TIM2_CCR1</td>
<td>I2C1_TRISE</td>
</tr>
<tr>
<td>0x40021004</td>
<td>0x40000034</td>
<td>0x40005420</td>
</tr>
<tr>
<td>RCC_CIFGR</td>
<td>TIM3 CR1</td>
<td>I2C1_FLTR</td>
</tr>
<tr>
<td>0x40021000</td>
<td>0x40000400</td>
<td>0x40005424</td>
</tr>
<tr>
<td>RCC_CIIFR</td>
<td>TIM3 CR2</td>
<td>0x40000404</td>
</tr>
<tr>
<td>0x40021014</td>
<td>0x40000404</td>
<td></td>
</tr>
<tr>
<td>RCC_CIICR</td>
<td>TIM3 SR</td>
<td>0x40000410</td>
</tr>
<tr>
<td>0x40021018</td>
<td>0x40000410</td>
<td></td>
</tr>
<tr>
<td>RCC_IOPRSTR</td>
<td>TIM3_DIIE</td>
<td>0x4000040C</td>
</tr>
<tr>
<td>0x4002101C</td>
<td>0x40000410</td>
<td></td>
</tr>
<tr>
<td>RCC_AHBRESTR</td>
<td>TIM3 EGR</td>
<td>0x40000414</td>
</tr>
<tr>
<td>0x40021020</td>
<td>0x40000414</td>
<td></td>
</tr>
<tr>
<td>RCC_APB2RESTR</td>
<td>TIM3 CCMR1</td>
<td>0x40000418</td>
</tr>
<tr>
<td>0x40021024</td>
<td>0x40000418</td>
<td></td>
</tr>
<tr>
<td>RCC_APB1RSTR</td>
<td>TIM3 CCER</td>
<td>0x40000420</td>
</tr>
<tr>
<td>0x40021028</td>
<td>0x40000420</td>
<td></td>
</tr>
<tr>
<td>RCC_IOPENR</td>
<td>TIM3_CNTL</td>
<td>0x40000424</td>
</tr>
<tr>
<td>0x4002102C</td>
<td>0x40000424</td>
<td></td>
</tr>
<tr>
<td>RCC_AHBENR</td>
<td>TIM3 PSC</td>
<td>0x40000428</td>
</tr>
<tr>
<td>0x40021030</td>
<td>0x40000428</td>
<td></td>
</tr>
<tr>
<td>RCC_APB2ENR</td>
<td>TIM3 ARR</td>
<td>0x4000042C</td>
</tr>
<tr>
<td>0x40021034</td>
<td>0x4000042C</td>
<td></td>
</tr>
<tr>
<td>RCC_APB1ENR</td>
<td>TIM3_CCR1</td>
<td>0x40000434</td>
</tr>
<tr>
<td>0x40021038</td>
<td>0x40000434</td>
<td></td>
</tr>
<tr>
<td>RCC_IOPSMENR</td>
<td>0x4002103C</td>
<td></td>
</tr>
<tr>
<td>RCC_AHBSMENR</td>
<td>0x40021040</td>
<td></td>
</tr>
<tr>
<td>RCC_APB2SMENR</td>
<td>0x40021044</td>
<td></td>
</tr>
</tbody>
</table>
C-Code CMSIS style
The Cortex Microcontroller Software Interface System (CMSIS) is a standardized, vendor
independent software interface for the Cortex M-series processors. In addition to a number of
libraries to interface code to peripherals, it provides register definitions that allow access by way
of c-code. These register definitions make use of the typedef structure and pointers and
provides a uniform access path for the peripherals and registers on all of the ARM Cortex
processors.

In c-code the typedef statement allows the user to rename a variable type. For example I can enter:
typedef int weight; //Rename int to weight
weight w1, w2;       //Declare w1 and w2 as ints

You can extend this to arrays. For example
typedef float height[50]; //Rename float as an array named height
height men;                //Create men[50] as floats

The typedef declaration can also be used with the struct declaration. Consider this example:
typedef struct             //rename a struct as record
    {int age;
     float height;
     float weight;
    } record;
record john, mary;     //john and mary are now structs

The user has access to the variables in the struct by means of the dot operator.
john.age = 22;
If we look at the include files for the ARM processor using CMSIS we find definitions for structs
that might look like the following:
/*!< Peripheral memory map */
#define PERIPH_BASE   ((uint32_t)0x40000000)
#define APB1PERIPH_BASE  PERIPH_BASE
#define APB2PERIPH_BASE  (PERIPH_BASE + 0x00010000)
#define AHB1PERIPH_BASE  (PERIPH_BASE + 0x00020000)
#define AHB2PERIPH_BASE  (PERIPH_BASE + 0x10000000)

/*!< AHB1 peripherals */
#define GPIOA_BASE (AHB1PERIPH_BASE + 0x0000)
#define GPIOB_BASE (AHB1PERIPH_BASE + 0x0400)
#define GPIOC_BASE (AHB1PERIPH_BASE + 0x0800)
#define GIOD_BASE (AHB1PERIPH_BASE + 0x0C00)
#define GPIOE_BASE (AHB1PERIPH_BASE + 0x1000)
#define GPIOF_BASE (AHB1PERIPH_BASE + 0x1400)
#define GPIOG_BASE (AHB1PERIPH_BASE + 0x1800)
#define GIIH_BASE (AHB1PERIPH_BASE + 0x1C00)
#define GPIOI_BASE (AHB1PERIPH_BASE + 0x2000)
From these declarations we see that, for example, the GPIOA_BASE address is at 0x40020000 and GPIOG_BASE is at 0x40021800.

A typedef for a struct is defined to gain access to the port registers.

```c
typedef struct
{
  __IO uint32_t MODER;   //Address offset: 0x00
  __IO uint32_t OTYPER;  //Address offset: 0x04
  __IO uint32_t OSPEEDR; //Address offset: 0x08
  __IO uint32_t PUPDR;   //Address offset: 0x0C
  __IO uint32_t IDR;     //Address offset: 0x10
  __IO uint32_t ODR;     //Address offset: 0x14
  __IO uint32_t BSRR;    //Address offset: 0x18
  __IO uint32_t LCKR;    //Address offset: 0x1C
  __IO uint32_t AFR[2];  //Address offset: 0x20-0x24 */
} GPIO_TypeDef;
```

Finally we can define the individual port pointers.

```c
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE ((GPIO_TypeDef *) GPIOE_BASE)
#define GPIOF ((GPIO_TypeDef *) GPIOF_BASE)
#define GPIOG ((GPIO_TypeDef *) GPIOG_BASE)
#define GPIOH ((GPIO_TypeDef *) GPIOH_BASE)
#define GPIOI ((GPIO_TypeDef *) GPIOI_BASE)
```

We see that (GPIO_TypeDef *) is defining a pointer to struct. For example, after these declaration, GPIOA will be a pointer to GPIOA_BASE which is the base address of the registers associated with GPIOA. To access one of the registers, say the output data register, we can use the following:

```c
(*GPIOA).ODR = 0;
```

In this notation the *GPIOA dereferences the pointer and dot operator allows access to a member or the struct. But this is a bit unwieldy so there is a replacement notation.

```c
GPIOA -> ODR = 0;
```

This notation uses the arrow operator to access a struct member.

To summarize the following three lines to the same thing – set bit PA7 to 0 using different notation. The first uses a define statement that makes the address of the output data register on GPIO port A a dereferenced pointer. It uses `#include "stm32f446.h"` which is an include file from the course web site. The second and third use `#include "stm32f446xx.h"` which is an include file that comes from the vendor (STM) and is part of the installation of Keil µVision 5 when you use the Nucleo M4 Board. For the second and third the GPIOA term is a pointer to a structure that defines the registers for the port with offsets from a base address.
GPIOA_ODR &= ~(1 << 7);  //PA7 to 0
GPIOA -> ODR &= ~(1 << 7); //PA7 to 0
(*GPIOA).ODR &= ~(1 << 7); //PA7 to 0

**Easy Guide to Assembly Code**

In general, for the ARM processors you can add an *s* to an instruction and it will alter the flags. Using the same instruction without an *s* will not alter the flags. For the compiler/assembler that we have this is often not an option and you are forced to use one or the other.

Note also that the assembler is *not* case sensitive so that R0 is the same as r0 and you can use whatever you want although you should be consistent for readability.

This particular assembler is sensitive to spaces. *All labels must begin in column 1 and no assembly mnemonic is allowed to begin in that column.*

**Instructions**

The ARM M0 and M0+ implements 56 instructions in assembly code – 50 of these are 16-bit thumb instructions and just 6 are 32-bit instructions. Table 4 lists the mnemonics for the instructions that are implemented. The following mnemonics are for the 6, 32-bit instructions: BL, DSB, DMB, ISB, MRS, and MSR.

The text below gives the more commonly used instructions with examples (not exhaustive) of how them may be used.

**Loading immediate data into a register:**

```assembly
ldr R3, =0x000003FF;  Loads the 32 bit hex number into R3
ldr R2, =41;          Loads 41 base 10 into R2
```

**Store a register into a memory:**

```assembly
str R2, [R1];         Stores R2 into the memory whose address is in R1
```

**Copy one register into another:**

```assembly
movs R2, R3;           Copies R3 into R2
movs R2, #0xFF;        Copies 255 into R2. Immediate limited to 0 to 255
```

**Add:**

```assembly
adds R2, R3;           Adds R2 + R3 and puts result in R2 (flags set)
adds R10, R11;         R10 = R10 + R11 but flags not set
```

**Note:** Flags are set only if registers are R0 to R7

```assembly
adcs R2, R3;           R2 = R2 + R3 + carry
```

**Note:** Can be used only with registers are R0 to R7
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Brief description</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCS</td>
<td>{Rd}, Rn, Rm</td>
<td>Add with Carry</td>
<td>N,Z,C,V</td>
</tr>
<tr>
<td>ADD{S}</td>
<td>{Rd}, Rn, &lt;Rm</td>
<td>imm&gt;</td>
<td>Add</td>
</tr>
<tr>
<td>ADR</td>
<td>Rd, Label</td>
<td>PC-relative Address to Register</td>
<td>-</td>
</tr>
<tr>
<td>ANDS</td>
<td>{Rd}, Rn, Rm</td>
<td>Bitwise AND</td>
<td>N,Z</td>
</tr>
<tr>
<td>ARS</td>
<td>{Rd}, Rn, &lt;Rs</td>
<td>imm&gt;</td>
<td>Arithmetic Shift Right</td>
</tr>
<tr>
<td>B{cc}</td>
<td>Label</td>
<td>Branch {conditionally}</td>
<td>-</td>
</tr>
<tr>
<td>BICS</td>
<td>{Rd}, Rn, &lt;#imm&gt;</td>
<td>Bit Clear</td>
<td>N,Z</td>
</tr>
<tr>
<td>BKPT</td>
<td>#imm</td>
<td>Breakpoint</td>
<td>-</td>
</tr>
<tr>
<td>BL</td>
<td>Label</td>
<td>Branch with Link</td>
<td>-</td>
</tr>
<tr>
<td>BLX</td>
<td>Rm</td>
<td>Branch indirect with Link</td>
<td>-</td>
</tr>
<tr>
<td>BX</td>
<td>Rm</td>
<td>Branch indirect</td>
<td>-</td>
</tr>
<tr>
<td>CMN</td>
<td>Rn, Rm</td>
<td>Compare Negative</td>
<td>N,Z,C,V</td>
</tr>
<tr>
<td>CMP</td>
<td>Rn, &lt;Rm</td>
<td>imm&gt;</td>
<td>Compare</td>
</tr>
<tr>
<td>CPSID</td>
<td></td>
<td>Change Processor State, Disable Interrupts</td>
<td>-</td>
</tr>
<tr>
<td>CPSIE</td>
<td></td>
<td>Change Processor State, Enable Interrupts</td>
<td>-</td>
</tr>
<tr>
<td>DMB</td>
<td></td>
<td>Data Memory Barrier</td>
<td>-</td>
</tr>
<tr>
<td>DSB</td>
<td></td>
<td>Data Synchronization Barrier</td>
<td>-</td>
</tr>
<tr>
<td>EORS</td>
<td>{Rd}, Rn, Rm</td>
<td>Exclusive OR</td>
<td>N,Z</td>
</tr>
<tr>
<td>ISB</td>
<td></td>
<td>Instruction Synchronization Barrier</td>
<td>-</td>
</tr>
<tr>
<td>LDM</td>
<td>Rn(!), reglist</td>
<td>Load Multiple registers, increment after</td>
<td>-</td>
</tr>
<tr>
<td>LDR</td>
<td>Rt, Label</td>
<td>Load Register from PC-relative address</td>
<td>-</td>
</tr>
<tr>
<td>LDR</td>
<td>Rt, [Rn, &lt;Rm</td>
<td>imm&gt;]</td>
<td>Load Register with word</td>
</tr>
<tr>
<td>LDRB</td>
<td>Rt, [Rn, &lt;Rm</td>
<td>imm&gt;]</td>
<td>Load Register with byte</td>
</tr>
<tr>
<td>LDRE</td>
<td>Rt, [Rn, &lt;Rm</td>
<td>imm&gt;]</td>
<td>Load Register with halfword</td>
</tr>
<tr>
<td>LDRSB</td>
<td>Rt, [Rn, &lt;Rm</td>
<td>imm&gt;]</td>
<td>Load Register with signed byte</td>
</tr>
<tr>
<td>LDRSH</td>
<td>Rt, [Rn, &lt;Rm</td>
<td>imm&gt;]</td>
<td>Load Register with signed halfword</td>
</tr>
<tr>
<td>LLS</td>
<td>{Rd}, Rn, &lt;Rs</td>
<td>imm&gt;</td>
<td>Logical Shift Left</td>
</tr>
<tr>
<td>LRS</td>
<td>{Rd}, Rn, &lt;Rs</td>
<td>imm&gt;</td>
<td>Logical Shift Right</td>
</tr>
<tr>
<td>MOV{S}</td>
<td>Rd, Rm</td>
<td>Move</td>
<td>N,Z</td>
</tr>
<tr>
<td>MRS</td>
<td>Rd, spec_reg</td>
<td>Move to general register from special register</td>
<td>-</td>
</tr>
<tr>
<td>MSR</td>
<td>spec_reg, Rm</td>
<td>Move to special register from general register</td>
<td>N,Z,C,V</td>
</tr>
<tr>
<td>MULS</td>
<td>Rd, Rn, Rm</td>
<td>Multiply, 32-bit result</td>
<td>N,Z</td>
</tr>
<tr>
<td>MVNS</td>
<td>Rd, Rm</td>
<td>Bitwise NOT</td>
<td>N,Z</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td>No Operation</td>
<td>-</td>
</tr>
<tr>
<td>ORRS</td>
<td>{Rd}, Rn, Rm</td>
<td>Logical OR</td>
<td>N,Z</td>
</tr>
<tr>
<td>POP</td>
<td>reglist</td>
<td>Pop registers from stack</td>
<td>-</td>
</tr>
<tr>
<td>PUSH</td>
<td>reglist</td>
<td>Push registers onto stack</td>
<td>-</td>
</tr>
<tr>
<td>REV</td>
<td>Rd, Rm</td>
<td>Byte-Reverse word</td>
<td>-</td>
</tr>
<tr>
<td>REV16</td>
<td>Rd, Rm</td>
<td>Byte-Reverse packed halfwords</td>
<td>-</td>
</tr>
<tr>
<td>REVSH</td>
<td>Rd, Rm</td>
<td>Byte-Reverse signed halfword</td>
<td>-</td>
</tr>
<tr>
<td>RORS</td>
<td>{Rd}, Rn, Rs</td>
<td>Rotate Right</td>
<td>N,Z,C</td>
</tr>
<tr>
<td>RSBS</td>
<td>{Rd}, Rn, #0</td>
<td>Reverse Subtract</td>
<td>N,Z,C,V</td>
</tr>
<tr>
<td>SBCS</td>
<td>{Rd}, Rn, Rm</td>
<td>Subtract with Carry</td>
<td>N,Z,C,V</td>
</tr>
<tr>
<td>SEV</td>
<td></td>
<td>Send Event</td>
<td>-</td>
</tr>
<tr>
<td>STM</td>
<td>Rn(!), reglist</td>
<td>Store Multiple registers, increment after</td>
<td>-</td>
</tr>
<tr>
<td>STR</td>
<td>Rt, [Rn, &lt;Rm</td>
<td>imm&gt;]</td>
<td>Store Register as word</td>
</tr>
<tr>
<td>STRB</td>
<td>Rt, [Rn, &lt;Rm</td>
<td>imm&gt;]</td>
<td>Store Register as byte</td>
</tr>
<tr>
<td>STRH</td>
<td>Rt, [Rn, &lt;Rm</td>
<td>imm&gt;]</td>
<td>Store Register as halfword</td>
</tr>
<tr>
<td>SUB{S}</td>
<td>{Rd}, Rn, &lt;Rm</td>
<td>imm</td>
<td>imm&gt;</td>
</tr>
<tr>
<td>SVC</td>
<td>#imm</td>
<td>Supervisor Call</td>
<td>-</td>
</tr>
<tr>
<td>SXTB</td>
<td>Rd, Rm</td>
<td>Sign extend byte</td>
<td>-</td>
</tr>
<tr>
<td>SXTH</td>
<td>Rd, Rm</td>
<td>Sign extend halfword</td>
<td>-</td>
</tr>
<tr>
<td>TST</td>
<td>Rn, Rm</td>
<td>Logical AND based test</td>
<td>N,Z</td>
</tr>
<tr>
<td>UXTB</td>
<td>Rd, Rm</td>
<td>Zero extend a byte</td>
<td>-</td>
</tr>
<tr>
<td>UXTH</td>
<td>Rd, Rm</td>
<td>Zero extend a halfword</td>
<td>-</td>
</tr>
<tr>
<td>WFE</td>
<td></td>
<td>Wait For Event</td>
<td>-</td>
</tr>
<tr>
<td>WFI</td>
<td></td>
<td>Wait For Interrupt</td>
<td>-</td>
</tr>
</tbody>
</table>

angle brackets, <>, enclose alternative forms of the operand
braces, {}, enclose optional operands and mnemonic parts
Subtract:
```
subs R1, R2;       R1 = R1 – R2
subs R1, R2, R3;   R1 = R2 – R3
```
Note: Can be used only with registers are R0 to R7
```
sbcs R1, R2;       R1 = R1 – R2 and accounts for carry flag (borrow)
```
Note: Can be used only with registers are R0 to R7

Multiply and Divide
```
muls R1, R2;       R1 = R1*R2
```
The `mul` instruction is for signed multiplication. You can also do unsigned multiply with `umul`.
```
sdiv R0, R1;       R0 = R0/R1
```
The `sdiv` instruction is for signed division. There is also a `udiv` for unsigned division.
Note: Can be used only with registers are R0 to R7

Compare:
```
cmp R2, R10;     Subtracts R10 from R2, discards result and sets flags
ncmp R1, #0x20;    Subtracts 0x20 from R1, discards result and set flags
```
Note: for immediate operand value must be 0 to 255.

Logical AND, OR, and Exclusive OR
```
ands R1, R2;       R1 = R1 AND R2
orrs R1, R2;       R1 = R1 OR R2
eors R1, R2;       R1 = R1 Ex Or R2
```
Note: Can be used only with registers are R0 to R7

Negation and Inverse:
```
neg R2, R2;        R2 = -R2  twos complement
neg R2, R3;        R2 = -R3  twos complement
mvn R0, R1;        R0 = inverse of R1
```

Setting a bit:
```
ORRS R0, #(1<<7);  Sets bit 7 of R0 to 1
```

Clearing a bit:
```
BIC R0,#(1<<7);    BIC is Bit Clear instruction. Clears bit 7
```

Toggling a bit:
```
eors R0, #(1<<7);  Exclusive OR to toggle bit 7
```

Left and Right Shift
```
lsls R2, R3, #5;   shift left R3 5 times and store in R2
lsls R2, R3;       shift left R2, R3 value times. If R3 is 31, R2 is cleared, if R3 > 31 R2 and carry are cleared.
lsrs R2, R3, #5;   shift right R3 5 times and store in R2
lsrs R2, R3;       shift right R2, R3 value times. If R3 is 31, R2 is cleared, if R3 > 31 R2 and carry are cleared.
asrs R2, R3, #5;   arithmetic shift right R3 5 times and store in R2
asrs R2, R3;       arithmetic shift right R2, R3 value times. If R3 is 31, R2 is cleared, if R3 > 31 R2 and carry are cleared.
```
Note that because shifting is done with the barrel shifter you can add a shift operation to the second source operand for most instructions. For example:

\[ \text{adds R1, R0, R0, lsl #3; R1 = R0 + R0 << 3} \]

**Branch instructions**

- \( B \text{ Target; } \) unconditional branch to target within ±2K (10 bits)
- \( B\{\text{Cond}\} \text{ Target; } \) conditional branch to target within -252 to +258
- \( BL\{\text{Cond}\} \text{ Target; } \) Branch and link. Stores return address in R14 and conditionally branches to target. To return from subroutine move R14(link reg) into R15(PC)
- \( BX \{\text{Cond}\} \text{ Rm; } \) Branches indirect to address in Rm.
- \( BLX \{\text{Cond}\} \text{ Rm; } \) Branches indirect to address in Rm and stores return in R14 (link reg).

**Table 5**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Flags</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Z set</td>
<td>equal</td>
</tr>
<tr>
<td>NE</td>
<td>Z clear</td>
<td>not equal</td>
</tr>
<tr>
<td>CS/HS</td>
<td>C set</td>
<td>unsigned higher or same</td>
</tr>
<tr>
<td>CC/LO</td>
<td>C clear</td>
<td>unsigned lower</td>
</tr>
<tr>
<td>MI</td>
<td>N set</td>
<td>negative</td>
</tr>
<tr>
<td>PL</td>
<td>N clear</td>
<td>positive or zero</td>
</tr>
<tr>
<td>VS</td>
<td>V set</td>
<td>overflow</td>
</tr>
<tr>
<td>VC</td>
<td>V clear</td>
<td>no overflow</td>
</tr>
<tr>
<td>HI</td>
<td>C set and Z clear</td>
<td>unsigned higher</td>
</tr>
<tr>
<td>LS</td>
<td>C clear or Z set</td>
<td>unsigned lower or same</td>
</tr>
<tr>
<td>GE</td>
<td>N equals V</td>
<td>signed greater or equal</td>
</tr>
<tr>
<td>LT</td>
<td>N not equal to V</td>
<td>signed less than</td>
</tr>
<tr>
<td>GT</td>
<td>Z clear AND (N equals V)</td>
<td>signed greater than</td>
</tr>
<tr>
<td>LE</td>
<td>Z set OR (N not equal to V)</td>
<td>signed less than or equal</td>
</tr>
<tr>
<td>AL</td>
<td>(ignored)</td>
<td>always (usually omitted)</td>
</tr>
</tbody>
</table>

**Push and Pop**

- push \( \{R1, R2\}; \) pushes \( R1 \) and \( R2 \) on the stack
- push \( \{R1-R5\}; \) pushes \( R1, R2, R3, R4, \) and \( R5 \) on the stack
- push \( \{R1, R2, LR\}; \) pushes \( R1, R2, \) and the link register on the stack
- pop \( \{R1, R2\}; \) pops \( R1 \) and \( R2 \) from the stack
- pop \( \{R1-R5\}; \) pops \( R1, R2, R3, R4, \) and \( R5 \) from the stack
- pop \( \{R1, R2, LR\}; \) pops \( R1, R2, \) and the link register from the stack

**Calling functions**

There are no call instructions in ARM assembly language. In place of an explicit call you use the BL instruction which is a Branch and Link. BL branches to the function but it also saves the return address in the link register (r14). To return from a function you use \( B\lr \) (or \( BLX \lr \))
which branches to the address in the link register. If a function calls another function it must save
the link register before doing a branch and link – otherwise the original link will be destroyed. You can do this by pushing the link register on the stack push lr. To return you can pop the link register and do a bx lr or you can simply pop the stack into the program counter. The sequence below shows how to call multiple functions in assembly code:

```
... bl Sub1
... push lr
  bl Sub2
  pop {pc}
...           Sub2
  push lr
  bl Sub3
  pop {pc}
   bx lr
                      Sub3
```

**Assemble Directives**

An assembler directive is a command that tells the assembler what to do with some portion of your program. It is not an executable statement. A complete list of assembler directives is given in Chapter 6 of ARM Assembler Reference [11]. The most commonly used directives are listed in Table 6 along with a brief description of what they do.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGN</td>
<td>Addresses need to be aligned on a 4-byte boundary. Use ALIGN 4 after a thumb instruction to ensure four-byte alignment since many thumb instructions are just 2-bytes long.</td>
</tr>
</tbody>
</table>
| AREA        | The AREA directive instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.
  
  For example, AREA Ex1, CODE, READONLY could be used for a new section named Ex1. |
| DCD         | The DCD directive allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory. & is a synonym for DCD.
  
  DCDU is the same, except that the memory alignment is arbitrary. |
| END         | This must be used at the end of all assembly code and should appear only once in a module. |
| ENTRY       | Declares the entry point of a program |
| EQU         | The EQU directive gives a symbolic name to a numeric constant, a register-relative value or a PC-relative value. * is a synonym for EQU. For example Addr1 EQU 0x12345678 allows you to use Addr1 in place of the hex constant in a program. |
| EXPORT      | The EXPORT directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. |
| FUNCTION    | The FUNCTION directive marks the start of a function. PROC is a synonym for FUNCTION. Every FUNCTION must have a matching ENDFUNCTION and every PROC must have a matching ENDPROC |
| ENDFUNCTION |            |
| IMPORT      | These directives provide the assembler with a name that is not defined in the current assembly. The linker matches import and export names. |
| EXTERN      |            |
| MACRO       | The MACRO directive marks the start of the definition of a macro. Macro expansion terminates at the MEND directive |
| MEND        |            |

Table 6

A list of frequently used ARM assembler directives
Addressing modes

The addressing modes include immediate addressing, register addressing, register indirect addressing, Pre-Indexed addressing, and PC-relative addressing. We present here an explanation of each with examples of how they are written in assembly code.

Immediate addressing

In this mode the operand is a value which gets loaded or stored to another location. Unfortunately, the immediate data field is only 12-bits long and this has been divided into an 8-bit value field and a 4-bit rotate field. The actual number in the immediate field evaluates to the value rotated right 2 times the number in the rotate field. Here are some examples:

```
movs r1, #77;           // This works since 77 < 255
movs r1, #0101B;        // Move binary data to r1
movs r2, #0xF3;         // This is OK since F316 < 25510
movs r0, #0xF3000000;   // This is OK since we can rotate right twice
movs r1, #0x0F30000;    // Rotate right F3 three times
movs r0, #0x0F030000;   // This is illegal since F0316 > 25510
```

The rotate would at first, appear to require several cycles but the ARM Cortex processors does this with a barrel shifter so a rotation can be done in a single cycle whether it is one bit or 16 bits.

Register addressing

In register addressing the operand is a direct address of a register and since there are only a few registers the direct address is much shorter than for a memory address. The ARM processor has 16 registers (r0 to r15) but since we are using the Thumb2 instruction set with 16-bit instructions instead of the full 32-bit ARM instructions, there is not room in the 16-bit format for a 4-bit register address. As a consequence most instructions can address only registers r0 to r7. Of the 56 or so instructions, only these seven have access to r8 to r12. (r12 is the stack pointer, r14 is the link register, and r15 is the program counter.)

```
adds r8, r9, r10; r8 <- r9 + r10
cmp r8, r9;         // subtract r8 - r9 and set flags
movs r8, r9;        // r8 <- r9
bx r9;              // branch to addr in r9
blx r9;             // branch to addr in r9 and save link
msr psr, r9;        // prog status register <- r9
mrs r8, psr;        // r8 <- prog status register
```

Register indirect addressing

In this addressing mode a register operand holds the address of the data. Since you cannot directly address memory, you need to put an address in a register and use register indirect addressing to get at the data.

```
ldr r0, [r1];        // r0 <- data from M at address in r1
```

You can also do register indirect addressing with an offset as in this instruction.

```
ldr r0, [r1, 1000]; r0 <- data from M at address r1 + 1000
```
In this case the data is in memory at the address which is the sum of the data in r1 and the offset of 1000. The r1 register remains unchanged. The offset can also be a register or a shifted register as in this instruction.

\[
\text{ldr } r0, [r1, r2]; \quad r0 \gets \text{data in M at address } r1 + r2 \\
\text{ldr } r0, [r1, r2, lsl #2]; r0 \gets \text{data in M at address } r1 + r2 \text{ shifted left twice.}
\]

In register indirect addressing the register which holds the address of the data is not changed.

Pre-Indexed addressing

Pre-indexed addressing the same as register indirect addressing with the exception that the register holding the address is modified with the offset. An exclamation point is used to designate this addressing mode. Here is an example of an instruction using pre-indexed addressing:

\[
\text{ldr } r0, [r1, #-255]!; r0 \gets \text{data from M at address } r1 + \text{offset} \\
\quad \text{offset ranges from } -255 \text{ to } +4095
\]

For this instruction the offset (-255) is added to r1 and r1 is modified. This sum is used as the memory address of where to find the data.

Post-Indexed addressing

Post-indexed addressing is the same as indexed addressing except that the offset is added to the index register after the load.

\[
\text{ldr } r0, [r1], #-255; r0 \gets \text{data from M at address } r1. \quad \text{Offset is added to } r1 \text{ after load}
\]

In this example, the data at the memory whose address is in r1 is loaded into r0. After this load r1 is modified to the sum of r1 and the offset.

PC-relative addressing

In PC-relative addressing the program counter plus an offset form the address of the data in memory which can be loaded into a register.

\[
\text{ldr } r0, [pc+15]; \quad r0 \gets \text{data from M at address } pc + 15
\]

While you can use this explicitly the assembler also uses it implicitly in place of direct addressing. If you write

\[
\text{ldr } r0, =0x12345678;
\]

The assembler converts this to something like this:

\[
\text{ldr } r0, [pc+23];
\]

and it stores the number 0x12345678 along with the program at the location pc+23. This makes it appear as if you can load an immediate value into a register with the \texttt{ldr} instruction.
You can do something similar by creating your own data area in memory.

AsmLoopExmp  PROC

    push  {r0-R5};             save the registers
    ...
    ldr  r0,  =Data
    ldr  r1,  [r0];
    ldr  r2,  [r0,  #4];
    ...
    pop  {r0-r5};             Restore the registers
    bx  lr;
ENDP

Data  DCD  0x87654321, 0x45362718

The term DCD is an assembler directive to create storage for a 32-bit word or multiple words. Data is a label at the location of the two words created. The instruction ldr  r0,  =Data appears to load the address of Data into r0 as an immediate operand. But in fact, the assembler converts this sequence to PC-relative addressing: ldr  r0,  [pc+24]; where the assembler has calculated that the label Data is 24 bytes ahead of its present location. You could do the same thing by hand but this notation makes it much easier.

**Programming in Assembly Code**

There are two methods that can be used to write assembly code in Keil µVision 5. The first is to write a function in assembly code and incorporate that function in a main program written in C. The second is to write the assembly code directly.

*Writing an Assembler function in a C-Code program*

To use this method you write a main program in C-code and use the C-code to call the assembler function. You will need to use a function prototype and it can have a return value and a parameter list. The first four parameters are passed in the registers R0, R1, R2, and R3. If there are more than four parameters the remainder are passed on the stack. The return value always comes back in R0.

The sample code in Figure 5.7 shows a c-program which called an assembler function called Add2 and passes it two integer parameters. The function adds the two parameters which it catches in R0 and R1 and returns the sum in R0.
When you enter the program in Figure 5.7 in μVision 5 the error checking does not recognize the assembly code as correct c-syntax and flags all of the assembly code as errors. However, you can ignore this and build the program. If there are errors in the build, these will be correctly flagged and you can correct them. Otherwise the program will compile and run correctly. Since this program has no output you can track through it using the debugger and observing the changes in the registers.

For very short assembler routines like that shown in Figure 5.7 the method of placing the assembler code inside the c-code is acceptable. The disadvantage is that the compiler flags it as erroneous since it thinks it is supposed to use c-syntax. You also cannot use the EQU assembler directive to define addresses with symbols (You can use #define).

A better way to place assembler and c-code together is to write a separate assembly module. In this case you write the c-code as you would any other. The assembly function will be written in a second file that is added to the source group with a dot-s extension.

In the c-code you will need a prototype for the function but this looks like the prototype for any c-function except that it needs to have the word extern added to the prototype declaration as in:

```c
extern void WriteBit(int x, int y);
```

The dot-s file will contain an assembly language procedure (function) that looks like that shown in Figure 5.8. Note that the name of the procedure is exported and the linker puts together the exported and extern symbols. The equate directive allows you to substitute meaningful names for strings of numbers. The name of the procedure must be declared as such with the PROC directive. Likewise the procedure must end the ENDP directive. You can put multiple procedures in one dot-s file but each must begin with the PROC directive and end with the ENDP directive. Finally at the end of all of the assembly code you must have an END directive. All code after the END directive is ignored.
#include "main.h"
#include "stm32L031.h"

void SystemClock_Config(void);

extern void WriteBit(int x, int y); //x is pos y is bit

int main()
{
    int i, j;

    //Clock bits
    RCC_IOPENR |= 1;      //Bit 0 is GPIOA clock enable bit
    GPIOA_MODER |= 0x00555555; //Bits 0-11 on PA set
        //to 01 output in MODER
    //OTYPER register resets to 0 so it is push/pull by default
    GPIOA_OSPEEDER |= 0xC000;  //Bits 15-14 = 11 for high
        //speed on PA7

    while(1)
    {
        for(i=0;i<12;i++)
        {
            j = i;
            WriteBit(j, 1);
        }

        for(i=0;i<12;i++)
            WriteBit(i, 0);
    }

    AREA MyProg, CODE, READONLY
    EXPORT WriteBit

    GPIOA_ODR EQU 0x50000014

    WriteBit PROC
        push {r0-R5};        save the registers
        movs r2, #1;         shift a 1 to the left
        lsls r2, r0;           bitPos times
                ldr r4, =GPIOA_ODR; address of Port A
                ldr r3, [r4];        Copy port A to r3
                cmp r1, #0;          If value is a 0
                        BEQ over;              branch to over
                orrs r3, r2;         OR r2 with data from port A
                nop;                add this for padding
                        B done;
        over    ldr r5, =0xFFF;      Create a 12 bit number that is
                eors r2, r5;        all 1s except for bitPos
                ands r3, r2;        AND it with data from port A
                done    str r3, [r4];    Store result in port A
                        pop {r0-r5};        Restore the registers
                bx lr;
        ENDP

    END

    Figure 5.8 This figure shows the syntax for an assembly procedure. Note that all labels must begin in column 1.
    The main C code is above and the external assembly code below.
Writing stand-alone assembly code

To write a stand-alone assembly module using µVision 5 you begin just as you would for a project in c-code but you do not need the include file. You will need the startup files that are generated. Add a new file to your project with a dot-s extension as in MyAsm.s. The s stands for source file and is used to designate assembly code. A sample assembly code file is shown in Figure 5.9.

The label Main marks the beginning of the code or the entry point. When the program runs, the first thing that runs is the start up file. This file transfer control to your program. If we look at the file startup_stm32f407xx.s we can find a section of code labeled as the "Reset Handler". For a typical c-program it is shown in Figure 5.10.

```
;AsmDirect.s              June 30, 2019
;

AREA MyProg, CODE, READONLY

ENTRY
EXPORT Main
;Must go to startup_stm32f407xx.s and change
;  IMPORT __main to IMPORT Main
;  LDR R0, =__main to LDR     R0, =Main
Main
MOVS r4, #4;
MOVS r0, #3;
MOVS r1, #3;
MULS r1, r0, r1;
ADDs r3, r0, r4;
SUBS r2, r4, r0;
CMP R0, #1
Again
BNE Over
NOP

Over
CMP r0, #3;
BEQ Again
END
```

Figure 5.9 A sample stand-alone assembly code file.
Figure 5.10 The reset handler code in a typical c-program startup_stm32f407xx.s file.

This code imports the symbol __main and the two instructions:

LDR R0, =__main
BX R0

Do a branch to this label. In the c-code the compiler adds in a statement to export __main where __main marks the start of the c-code. The linker puts the imported and exported addresses together.

To get the startup file to branch to our assembly code we change __main in the startup code to Main since we have used that label as the start of our assembler code and we have a statement to export that label. (Alternatively, instead of changing the startup code we could just name the entry point in our assembly code to __main.)

Once these changes are made we can compile our program, load it on the Discovery Board and run it and/or execute it in the debugger. Note that the last statement in the assembly code is:

Again B Again

If you leave this out the assembly code continues executing through memory and eventually creates an error. In most other programs that run on a microcontroller there is an infinite loop somewhere that keeps running albeit, not necessarily at the end of the code.

To create a c-code program for the ARM Nucleo Board you can follow the directions in Section 2 on setting up a Keil µVision 5 project, or, much more simply, you can copy the STM32L0Template file from the course website. This is a zipped file. Unzip it and copy the directory where you want to place your program. You can rename the directory and rename the .c source file within the directory to your project name. Double click on the project file, STM32L0Template.uvprojx to open the project in µVision 5. If you renamed the source code you will have to remove STM32L0Template.c from the project and add your own c file.

The file STM32L0Template.c is shown in Figure 5.11. This file uses "stm32L031.h" which is the standard include file (not the CMSIS file). The program toggles the bits on GPIOB.

Note that the program includes a function called InitializeClock(). In Keil µVision 5 the default clock runs at 16 MHz from the internal RC oscillator. If you comment out the call to SystemClock_Config() the program will run at 2 MHz. The SystemClock_Config() function
resets the clock to 32 MHz and runs via the phase locked loop from the internal 16 MHz rc oscillator.

/* STM32L0Template June 20, 2019
   This is a test program for the STM32NUCLEO-L031K6 Board
   It toggles the first 10 bits on port B. Using an oscilloscope
   you can see a square wave at 5.003 MHz on pin PB0, PB1,
   PB3, PB4, PB5, PB6, and PB7 which are the only port B pins on
   the chip.

   The system clock configuration sets the clock to 32 MHz.
   This uses the internal 16 MHz rc oscillator and the PLL

   To run this program on the Nucleo L0 board you need
   startup_sm32L031xx.s
   main.c this file
   system_stm32L0xx.c
*/
#include "main.h"
#include "stm32L031.h"

void SystemClock_Config(void);

int main(void)
{
    SystemClock_Config(); //Declare i for delay loop
    //Clock bits
    RCC_IOPENR |= 2; //Bit 1 is GPIOB clock enable bit
    //I/O bits PB2
    GPIOB_OSPEEDER = 0xC; //PB2 to high speed
    GPIOB_MODER = 0x55555; //Set first 10 port bits to output
    GPIOB_PUPDR &= 0xFFFFF000; //No pull up and no pull down
    tmp = 0;
    while(1)
    {
        GPIOB_ODR = tmp; //Inverts 10 bits on Port B
        tmp = ~tmp;
        for(i=0;i<1000;i++); //Add this delay loop
    }
}

//Sets clock to 32 MHz
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
    // Configure the main internal regulator output voltage
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
    // Initializes the CPU, AHB and APB busses clocks
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
    RCC_OscInitStruct.PLL.PLLMUL = RCC_PLLMUL_4;
    RCC_OscInitStruct.PLL.PLLDIV = RCC_PLLDIV_2;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        Error_Handler();
    }
    // Initializes the CPU, AHB and APB busses clocks
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
    |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
    {
        Error_Handler();
    }
}

void Error_Handler(void)
{
}

Figure 5.11 The source file in the STM32L0Template.
Section 6 Programming Examples
In this section we provide example programs in both C-code and assembly language which illustrate the use of the various peripherals.

Assembly Language Examples

Example 1A Writing bits
This example, called WriteBit is written as a procedure which may be called from c-code using WriteBit(int bitNum, int value) where bitNum is the bit number 0 to 11 and value is a one or a zero to be written. All of the bits are on GPIOA. The program assumes that GPIOA has been setup for output on the first 12 bits by the calling c-code. R0 holds the bitNum and R1 has the bit value. The program works by putting a 1 in R2 and shifting it left bitNum times using lsls instruction. For a 1 R2 is logically ORed with the port or, for a 0, R2 is inverted and ANDed with the port.

The c-code and the corresponding WriteBit assembly code are given in Figure 5.8 for this example.
**Example 2A Loops**

These program fragments illustrate ways to do loops.

This loop runs 5 times. See immediate addressing mode for limitations on size of loop counter. In this example we check r1 for when it gets to zero.

```
mov r1, #5;  r1 = 5 = loop counter
Lp1 subs r1, #1;  r1 = r1 – 1
;Put loop body here
BNE Lp1;  if(r1 not zero) branch to Lp1
```

This loop runs 5 times. See immediate addressing mode for limitations on size of loop counter. In this example we use the compare instruction to see when r1 = r2

```
movs r1, #0;  initialize
movs r2, #5;
Lp2 adds r1, #1;  r1 = r1 + 1
;Put loop body here
cmp r1, r2;  compare r1 to r2
BNE Lp2;  if(r1 not equal r2) branch to Lp2
```

This loop runs 10000 times and an integer up to 32-bit long can be placed in r1 as a loop counter.

```
ldr r1, =10000;
Lp3 subs r1, #1;
;Put loop body here
BNE Lp3
```

This is a loop within a loop. The inner loop runs 200 times but it is within an outer loop that runs 100 times so the loop body runs 100 x 200 = 20,000 times.

```
ldr r1, =100;
Lp4 ldr r2, =200;
Lp5 ;Loop body goes here
    subs r2, #1;
    BNE Lp5;
subs r1, #1;
BNE Lp4;
```
Example 3A Logic
This example uses assembly language to perform some logical operations. In the figure below we can write  \( z = AB + CD \)

```
AREA MyProg, CODE, READONLY
EXPORT Logic

Logic PROC
    push {r0-R5};       save the registers
    pop {r0-r5};        Restore the registers
    ANDS r1,r2       ;r1 = r1 & r2
    ANDS r3,r4       ;r3 = r3 & r4
    MVNS r3,r3       ;r3 = (not)r3
    ORRS r1,r3       ;r0 = A&B + (not)(C&D)
    MVNS r1,r1       ;r1 = not r0

ENDP
END
```

The MVNS instruction is the movs not instruction. It moves the logical inverse of one register into another.
Example 4A GPIO
This example does output to PB0-PB11 on GPIOB. It is arranged as two functions. BitsIO is called by a C-program since it is exported. SetUp is called by BitsIO to set up GPIOA for output. The SetUp function should be entered immediately after BitsIO – its name is not exported so it is not an external module. SetUp turns on the clock for GPIOB, sets the mode for bits 0-11 as output at high speed, and returns. BitsIO outputs a 1 to PB0 to PB11 in succession setting the present bit back to zero before setting the next one to one.

; Sets up bits 0-11 on GPIOB as output at high speed. Triggers each bit to 1 and back to 0 before going to the next. ;Runs forever.
AREA MyProg, CODE, READONLY
EXPORT BitsIO

;GPIO Port A Addresses
gpiob_moder   equ 0x50000400
gpiob_pupdr   equ 0x50000404
gpiob_ospeede equ 0x50000408
gpiob_otyper  equ 0x5000040c
gpiob_idr     equ 0x50000410
gpiob_odr     equ 0x50000414
RCC_IOPENR    equ   0x4002102C

BitsIO PROC
    push {r0-R5}; save the registers
    b1 SetUp;  ;Set up clock and GPIOB
    ;Output bits on GPIOB 0-11
    ldr r1, =gpiob_odr;
    Lp0   movs r2, #12; lp cntr for 12 bits
          movs r0, #1;  ;Start with bit 0
    Lp1   str r0, [r1];
          lsls r0, #1;  ;shift to next bit
          subs r2, #1;
          BNE Lp1;  ;Finish all 12 bits
          B Lp0;  ;Do again forever
          pop {r0-r5}; Restore the registers
          bx lr;
ENDP

SetUp PROC
;Enable clock for GPIOA
    ldr r1, =RCC_IOPENR;
    ldr r0, [r1];
    movs r5, #2;
    orrs r0, r5; Bit 1 enables clock ; for port b
    str r0, [r1];
;Mode to output on GPIOA bits 0-11
    ldr r1, =gpiob_moder;
    ldr r0, [r1];  moder resets to
    orrs r0, r3;    all 1's -analog
    str r3, =0xFF555555; Bits 0-11 on
    ;PB set to 01 output in MODER
    movs r0, r3;
    str r0, [r1];
;Speed to high on GPIOB bits 0-11
    ldr r1, =gpiob_ospeede;
    ldr r0, [r1];
    ldr r3, =0x00FFFFFF;
    orrs r0, r3
    str r0, [r1];
    bx lr;
    ALIGN
    END
END
C-Code Examples

Example 1C GPIO

/* GPIOInOut.c                  July 1, 2019
   This program toggles pin PA7. It also inputs from pin PB1 and
   outputs its inverse to PA9 as fast as possible */

#include "main.h"
#include "stm32L031.h"

void SystemClock_Config(void);
int main(void)
{
    SystemClock_Config();
    int tmp, i;
    RCC_IOPENR |= 1;       //Bit 0 is GPIOA clock enable bit
    RCC_IOPENR |= 2;       //Bit 1 is GPIOB clock enable bit
    //I/O bits PA7
    GPIOA_MODER &= ~(1 << 15); //Bits 15-14 = 01 for digital out on PA7
    GPIOA_MODER |= (1 << 14);
    //OTyper register resets to 0 so it is push/pull by default
    GPIOA_OSPEEDER |= 0xC000;  //Bits 15-14 = 11 for high speed on PA7
    //PUPDR defaults to no pull up no pull down
    //I/O bits PA9
    GPIOA_MODER &= ~(1 << 19); //Bits 18-19 = 01 for digital out on PA9
    GPIOA_MODER |= (1 << 18);
    //OTyper register resets to 0 so it is push/pull by default
    GPIOA_OSPEEDER |= 0xC0000;  //Bits 18-19 = 11 for high speed on PA9
    //PUPDR defaults to no pull up no pull down
    //I/O bits PB2
    GPIOB_MODER &= ~0xC;       //PB1 to input
    //All bits are analog by default.
    tmp = 0;
    while(1)
    {
        if(tmp == 0)
            GPIOA_ODR &= ~(1 << 7); //Only PA7 is set up for output
        else
            GPIOA_ODR |= (1 << 7);
        tmp = ~tmp;
        //Input bit PB1 output 1 or 0 to PA9
        if((GPIOB_IDR & 0x2) == 0)
            GPIOA_ODR |= (1 << 9);
        else
            GPIOA_ODR &= ~(1 << 9);
        for(i=0;i<10000;i++); //Put this in to slow down toggle
    }
}
Example 2C Polled Timer

/* //TimerPolled.c
This program uses Timer 2 in a polled mode to toggle bit PA5
every msec. The timer is uses autoreloading. Timer 21/22 can
be used in a similar fashion. */

#include "main.h"
#include "stm32L031.h"

void SystemClock_Config(void);

int main(void)
{SystemClock_Config();
 int flag, tmp;
 RCC_IOPENR |= 1;   //Bit 0 is GPIOA clock enable bit
 RCC_APB1ENR |= 1;  //Enable peripheral timer for timer 2 (bit 2)
 //Bits 10-11 = 01 for digital output on PA5 in GPIOA_MODER
 GPIOA_MODER |= (1 << 10);  //Bits 10-11 to 01 for output on PA5
 GPIOA_MODER &= ~(1 << 11);
 //OTYPER register resets to 0 so it is push/pull by default
 GPIOA_OSPEEDER |= 0x0C00; //Bits 10-11 = 11 for high speed on PA7
 TIM2_CR1 |= (1 << 7); //Auto reload is buffered
 TIM2_CR1 |= (1 << 4); //count down
 TIM2_CR1 |= (1 << 3); //One pulse mode is on. ADD THIS LINE
 TIM2_PSC = 0; //Don't use prescaling
 TIM2_ARR = 32000; //32 MHz/32000 = 1msec (2 msec period)
 TIM2_EGR |= 1;
 TIM2_CR1 |= 1;  //Enable Timer 3
 //Main program loop
 tmp = 0;
 while(1)
 {GPIOA_ODR = tmp; //Only PA5 is set up for output so other bits
   tmp = ~tmp; // have no effect.
   while((TIM2_CR1 & 1) != 0); //Wait here until timer runs out
   TIM2_CR1 |= 1; //Restart timer
 }
}
Example 3C Timer Interrupt

/* //TimerInterrupt.c
This program uses Timer 21 in an interrupt mode to toggle bit PA5
every 500 usec to give a 1 msec period. The timer is autoreloading.
Note that the flag variable MUST be declared volatile. This tells the
compiler that the variable may be altered outside the main
program. If it is not declared as volatile the compiler
optimization treats flag in the main program different from
flag in the interrupt routine.
*/
#include "main.h"
#include "stm32L031.h"
volatile int flag;            //Must be declared volatile
void SystemClock_Config(void);
int main(void)
{SystemClock_Config();
  int tmp;
  RCC_IOPENR |= 1;      //Bit 0 is GPIOA clock enable bit
  RCC_APB1ENR |= 1;     //Enable peripheral timer for timer 2 (bit 2)
  RCC_APB2ENR |= 4;     //Peripheral timer for timer 21
  //Bits 10-11 = 01 for digital output on PA5 in GPIOA_MODER
  GPIOA_MODER |= (1 << 10);    //Bits 10-11 to 01 for output on PA5
  GPIOA_MODER &= ~(1 << 11);
  //OTYPER register resets to 0 so it is push/pull by default
  GPIOA_OSPEEDER |= 0x0C00;  //Bits 10-11 = 11 for high speed on PA7
  TIM21_CR1 |= (1 << 7);   //Auto reload is buffered
  TIM21_CR1 |= (1 << 4);   //count down
  TIM21_PSC = 0;          //Don't use prescaling
  TIM21_ARR = 16000;      //32 MHz/16000 = 500 usec
  TIM21_EGR |= 1;
  TIM21_CR1 |= 1;         //Enable Timer 21
  //Set up interrupt
  NVICISER0 |= (1 << 20); //Bit 20 in ISER0 for int 20 (TIM 21)
  TIM21_DIER |= 1;       //Enable Timer 21 update interrupt enable
  TIM21_DIER |= (1 << 6); //Enable Timer 21 trigger interrupt enable
  //Main program loop
  tmp = 0;
  while(1)
  {GPIOA_ODR = tmp;  //Only PA5 is set up for output so other bits
    tmp = ~tmp;       // have no effect.
    flag = 1;
    while(flag == 1);  //Wait here until timer interrupt
  }
}
void TIM21_IRQHandler()
{flag = 0;
  TIM21_SR &= 0xFFFFE; //Turn off the interrupt
}
Example 4C PWM Ramp function

/* PWMRamp.c
This program produces a PWM signal of a ramp function on
PA8. The system clock configuration sets the clock to 32 MHz.
This is the internal rc oscillator.
*/

#include "main.h"
#include "stm32L031.h"

void SystemClock_Config(void);

int main()
{
    int tmp, i;
    //Clock bits
    SystemClock_Config();
    RCC_IOPENR |= 1;  //Bit 1 is GPIOA clock enable bit
    RCC_APB1ENR |= 1;     //Enable peripheral timer for timer 2 (bit 0)
    //I/O bits
    GPIOA_MODER = 0xEBFEFCFF; //Bits 17-16 = 02 for Alternate function
                             //on PA8
    //OTYPE register resets to 0 so it is push/pull by default
    GPIOA_OSPEEDER |= 0x30000;  //Bits 17-16 = 11 for high speed
    //PUPDR defaults to no pull up no pull down
    //Timer 2 bits
    GPIOA_AFRH = 0x00000000; //Sets PA8 to Timer 2
    TIM2_CCMR1 |= 0x60;      //Timer 2 in PWM Mode bits 6,5,4 = 110
    TIM2_CCMR1 |= 0x0C;      //Timer Preload enable and fast enable
    TIM2_CR1 |= (1 << 7);    //Auto reload is buffered
    TIM2_PSC = 1;            //Prescale by factor of 2
    TIM2_ARR = 10000;        //((32 MHz)/10000)/2 = 1600 Hz
    TIM2_CCR1 = 0;           //Duty cycle starts at 0
    TIM2_CCR2 |= 1;          //Compare and capture output enable
    TIM2_EGR |= 1;           //Enable event
    TIM2_CR1 |= 1;           //Enable Timer 2
    //Main program loop
    tmp = 0;
    while(1)
    {
        TIM2_CCR1 = tmp;
        for(i=0;i<1000;i++); //Delay to slow down how fast
                             //PWM is updated
        tmp++;
        if(tmp >= TIM2_ARR)
        {
            tmp = 0;
        }
    }
}
Example 5C A/D to PWM

/* AtoDPWM June 24, 2019
   PA1 is the analog input for ADC IN1. PA5 is the PWM output. Attach a 1K
   potentiometer to input a variable voltage 0 to 3 volts from PA8.
   The voltage is used to adjust the duty cycle of the PWM signal on PA1.
   The System Clock is running at 32 MHz. The base frequency of the PWM
   signal is 1600 Hz. Adjust the pot to change the duty cycle.
*/

#include "main.h"
#include "stm32L031.h"

void SystemClock_Config(void);
int main(void)
{
    SystemClock_Config();
    int i, tmp;
    RCC_IOPENR |= 1; //Bit 1 is GPIOA clock enable bit
    RCC_APB2ENR |= 0x200; //Bit 9 is ADC 1 clock enable bit
    RCC_APB1ENR |= 1; //Enable peripheral timer for timer 2 (bit 0)
    //PWM set up for output bit
    GPIOA_MODER = 0xEBFEFCFF; //Bits 17-16 = 02  Alternate function on PA8
    //OTYPER register resets to 0 so it is push/pull by default
    GPIOA_OSPEEDER |= 0x30000; //Bits 17-16 = 11 for high speed
    //PUPDR defaults to no pull up no pull down
    //Timer 2 bits
    GPIOA_AFRH = 0x00000005; //Sets PA8 to Timer 2
    GPIOA_MODER |= 0x0C; //PA1 is analog
    GPIOA_PUPDR &= 0xFFFFFFCF; //Pin PA1 has no pull up and no pull down
    TIM2_CCMR1 |= 0x60; //Timer 2 in PWM Mode bits 6,5,4 = 110
    TIM2_CCMR1 |= 0x0C; //Timer Preload enable and fast enable
    TIM2_CR1 |= (1 << 7); //Auto reload is buffered
    TIM2_PSC = 9; //Prescaling by 10
    TIM2_ARR = 2000; //(((32 MHz)/2000)/10 = 1600 Hz
    TIM2_CCR1 = (ADC1_DR & 0xFFF)/2; //12-bits of data reg to PWM
    ADC1_ISR |= 1;
    ADC1_CR |= 1; //Clear ADRDY bit in ADC_ISR
    ADC1_CR |= 1; //Set ADEN = 1 in ADC_CR to enable ADC
    ADC1_CHSELR = 2; //Select Channel 1 only
    ADC1_SMPR = 7; //Sample time register
    while(1)
    {
        ADC1_CR |= 0x4; // Bit 2 does software start of A/D conversion
        while((ADC1_ISR & 0x4) == 0); //Bit 2 is End of Conversion
        TIM2_CCR1 = (ADC1_CR & 0xFFF)/2; //12-bits of data reg to PWM
        ADC1_ISR |= 1;
        ADC1_CR |= 1;
    }
}
Example 6C USART

//USART2Test.c
/* Transmits the character 'U' at 4800 baud on USART2 PA9 forever.
Since 'U' in ASCII is 0x55 = 0101 0101 the bit time on an oscilloscope
Will be 1/4800 = 0.2083333 msec
*/

void ConfigureUSART(unsigned int baudDivisor);
void USARTPutChar(char ch);
#include "main.h"
#include "stm32L031.h"

void SystemClock_Config(void);

int main(void)
{
  SystemClock_Config();
  //Clock bits
  RCC_IOPENR |= 1;         //Bit 0 is GPIOA clock enable bit
  RCC_APB1ENR |= (1 << 17);  //Enable USART2
  //UART PIN Bits
  GPIOA_AFRH = 0x0440;  //Alternate Func PA9 PA10 to USART2
  GPIOA_MODER |= (1 << 21);GPIOA_MODER &= ~(1 << 20);
  GPIOA_MODER |= (1 << 19);GPIOA_MODER &= ~(1 << 18); //Bits 18-21
    // = 1010 for Alt Func PA9, PA10
  //OTYPER register resets to 0 so it is push/pull by default
  GPIOA_OSPEEDER |= 0x3C0000;  //Bits 21-19 = 11 high speed on PA9-10
  //PUPDR defaults to no pull up no pull down
  //The over8 bit in CR1 = 0 so BRR = fclk/TxBaud
  //If fclk = 32MHz and TxBaud = 4800
  //BRR = 32000000/4800 = 6667
  ConfigureUSART(6667);
  while(1)
  {USARTPutChar('U');
  }
}

// void ConfigureUSART(unsigned int baudDivisor)
{USART2_CR1 = 0;       //Disable during set up. Wd len = 8, Parity = off
  USART2_BRR = baudDivisor;  //Set up baud rate
  USART2_CR2 = 0;        //1 stop bit
  USART2_CR1 = 0x0D;
  USART2_CR3 = 0;        //Disable interrupts and DMA
}

// void USARTPutChar(char ch)
{//Wait for empty flag
  while((USART2_ISR & 0x80) == 0);
  USART2_TDR = ch;
}
Example 7C UART Message

//USART2Msg.c
/* Transmits the message "Hello Mom!" at 9600 baud to PA9 periodically forever. Uses USART2. */

void ConfigureUSART(unsigned int baudDivisor);
void UARTPutChar(char ch);
void SendMsg(const char msg[]);
char msg[] = "/ Hello Mom!";
#include "main.h"
#include "stm32L031.h"
void SystemClock_Config(void);

int main(void)
{
    int i, j;
    SystemClock_Config();
    RCC_IOPENR |= 1; // Bit 0 is GPIOA clock enable bit
    RCC_APB1ENR |= (1 << 17); // Enable USART2
    GPIOA_AFRH = 0x0440; // Alternate Func PA9 PA10 to USART2
    GPIOA_MODER |= (1 << 21); GPIOA_MODER &= ~(1 << 20);
    GPIOA_MODER |= (1 << 19); GPIOA_MODER &= ~(1 << 18); // Bits 21, 20, 19, 18 = 1010 for Alt Func PA9, PA10
    GPIOA_OSPEEDER |= 0x3C0000; // Bits 21 to 19 = 11 for high speed on PA9 PA10
    ConfigureUSART(3333);
    while(1)
    {
        SendMsg(msg);
        for(i=0;i<50000;i++)
            for(j=0;j<5000;j++)
    }
}

void ConfigureUSART(unsigned int baudDivisor)
{
    USART2_CR1 = 0; // Disable during set up. Wd len = 8, Parity = off
    USART2_BRR = baudDivisor; // Set up baud rate
    USART2_CR2 = 0; // 1 stop bit
    USART2_CR1 = 0x0D;
    USART2_CR3 = 0; // Disable interrupts and DMA
}

void UARTPutChar(char ch)
{
    // Wait for empty flag
    while((USART2_ISR & 0x80) == 0);
    USART2_TDR = ch;
}

void SendMsg(const char msg[])
{
    int i = 0;
    while(msg[i] != 0)
    {
        UARTPutChar(msg[i]);
        i++;
    }
}
Example 8C UART Message to Virtual Com Port

//LPUART1Msg.c
/* This program uses the low power UART to transmit a message to PA2 at 9600 baud. PA2 is the virtual com port TX. */

void ConfigureUSART(unsigned int baudDivisor);
void UARTPutChar(char ch);
void SendMsg(const char msg[]);
char msg[] = "/ Hello Mom!";
#include "main.h"
#include "stm32L031.h"
void SystemClock_Config(void);
int main(void)
{
  int i, j;
  SystemClock_Config();
  //Clock bits
  RCC_IOPENR |= 1;  //Bit 0 is GPIOA clock enable bit
  //Select 16 MHz HSI clock for LP UART
  RCC_CCIPR |= (1 << 11);RCC_CCIPR &= ~(1 << 10);
  RCC_APB1ENR |= (1 << 18);  //Enable LPUART1
  //UART PIN Bits
  GPIOA_AFRL = 0x06600;  //Alternate Func PA2 PA3 to LPUART1
  //Bits 7, 6, 5, 4 = 1010 for Alt Function PA2, PA3
  GPIOA_MODER |= (1 << 7);GPIOA_MODER &= ~(1 << 6);
  GPIOA_MODER |= (1 << 5);GPIOA_MODER &= ~(1 << 6);
  GPIOA_OSPEEDER |= 0xF0;  //Bits 4 to 7 = 11 for high speed on PA2 PA3
  //For LPUART Baud Rate Divisor = 256*fclk/Baud rate
  //BRR = 256*16000000/9600 = 426667
  ConfigureUSART(426667);
  while(1)
  {
    SendMsg(msg);
    for(i=0;i<50000;i++);
    for(j=0;j<5000;j++);
  }
}

void ConfigureUSART(unsigned int baudDivisor)
{
  LPUART1_CR1 = 0;  //Disable during set up. Wd len = 8, Parity = off
  LPUART1_BRR = baudDivisor;  //Set up baud rate
  LPUART1_CR2 = 0;  //1 stop bit
  LPUART1_CR1 = 0x0D;  //Enable UART, TX, and RX
  LPUART1_CR3 = 0;  //Disable interrupts and DMA
}

void UARTPutChar(char ch)
{
  //Wait for empty flag
  while((LPUART1_ISR & 0x80) == 0);
  LPUART1_TDR = ch;
}

void SendMsg(const char msg[])
{
  int i = 0;
  while(msg[i] != 0)
  {
    UARTPutChar(msg[i]);
    i++;
  }
}
Appendix A - STLink
The instructions below tell how to set up the debugger to load a program onto the board using STLink.

1. With your board plugged in we will need to change the project options to use STLink. Click once on Target in the project pane and then click on Project→Options for Target 1 or click or Alt → F7. Under the Target tab set the crystal speed to 32.0 MHz (or whatever you are using).

2. Under the Debug tab click on the pull down menu on the right and choose STLink Debugger. Also click on the radio button labeled Use so that the Use Simulator button is not chosen. See Figure A.1.

![Figure A.1 The Debug options.](image)

3. After you have selected the STLink Debugger click on the Settings button next to it. The setting should look like that shown in Figure 2.12. If not, be sure your board is plugged in and that you have the Debug options set up as shown in Figure A.2.
4. Click on OK for the target options and get back to the window with the c-code in it.

At this point you are ready to upload your code to the board and run your program. To do this click on the Debug button or, click on Debug→ Start/Stop Debug Session. The program will be downloaded to your board. Click on Debug→Run to start the program running.

For future projects, in order not to have to run through the tedious STLink set up, you can create a new project folder on your computer and download STM32L0Template.zip to that folder. Unzip the project and delete the main c-file (named STM32L0Template.c). Open the project by double clicking on STM32L0Template.uvproj, add you own c-code to the file, and run it as usual.
Appendix B - Clock Frequencies

If you create a program without a clock configuration function it will, by default, run at about 2 MHz. To turn at a different frequency you need to adjust the clock. The clock can run at various frequencies between 2.097 MHz and 32 MHz. The clock configuration function for 2.097 MHz, 16 Mhz, and 32 MHz are shown below.

```c
//Sets clock to 32 MHz
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /** Configure the main internal regulator output voltage */
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
    /** Initializes the CPU, AHB and APB busses clocks */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
    RCC_OscInitStruct.PLL.PLLMUL = RCC_PLLMUL_4;
    RCC_OscInitStruct.PLL.PLLDIV = RCC_PLLDIV_2;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        Error_Handler();
    }
    /** Initializes the CPU, AHB and APB busses clocks */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK |
    RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
    {
        Error_Handler();
    }
}
```

**Figure B1** This is the clock configuration function for a 32 MHz clock. Change the line:

```c
RCC_OscInitStruct.PLL.PLLDIV = RCC_PLLDIV_2;
```

To

```c
RCC_OscInitStruct.PLL.PLLDIV = RCC_PLLDIV_4;
```

For a 16 MHz clock.
{  
  RCC_OscInitStruct.OscillatorType = RCC_OscInitTypeDef.RCC_OSCILLATORTYPE_MSI;
  RCC_OscInitStruct.MSIState = RCC_OscInitTypeDef.RCC_MSI_ON;
  RCC_OscInitStruct.MSICalibrationValue = 0;
  RCC_OscInitStruct.MSIClockRange = RCC_OscInitTypeDef.RCC_MSIRANGE_5;
  RCC_OscInitStruct.PLL.PLLState = RCC_OscInitTypeDef.RCC_PLL_NONE;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {  
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB busses clocks */
  RCC_ClkInitStruct.ClockType = RCC_ClkInitTypeDef.RCC_CLOCKTYPE_HCLK | RCC_ClkInitTypeDef.RCC_CLOCKTYPE_SYSCLK  
                             | RCC_ClkInitTypeDef.RCC_CLOCKTYPE_PCLK1 | RCC_ClkInitTypeDef.RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SysClockSourceTypeDef.RCC_SYSCLKSOURCE_MSI;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SysClockSourceTypeDef.RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_SysClockSourceTypeDef.RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_SysClockSourceTypeDef.RCC_HCLK_DIV1;
  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
  {  
    Error_Handler();
  }
}

Figure B2 Clock configuration function for 2.097 MHz.
Appendix C - Virtual Com Port

The USB port that is used to program the CPU on the STM32L031K6 board sets up a virtual com port on the PC which can be used to receive and transmit data on a serial line. On the board PA2 is the virtual com port transmit (to the computer) and PA3 is the virtual com port receiver (from the computer). Thus you can set up either the LPUART or USART2 to transmit and receive at 9600 baud on PA2 and PA3 and you board can talk to the console.

Program example 8C sets up the LPUART to send a message at 9600 baud continuously to PA2. To see this message on the console you need to set up a serial communications channel. There are several serial connections that can be used such as Putty. Putty is available free from here: https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

Before you get Putty set up you need to determine which COM port your system is using. (This may change from one run to the next.) Click on the Windows button or click on the windows icon in the bottom left corner. Type "Device Manager" and select it when it comes up. In the Device Manager click on Ports & LPT. You should see something like that shown in Figure C-1. Yours will likely be different from that shown but you should see The STMicroelectronics STLink Virtual COM Port followed by a port number. In the figure this is COM8. If you don't see this you may not have STLink installed or you may not have the board connected by way of the USB cable.

Figure C-1  The Device Manager showing the COM ports.

Once you know the COM port number you can open Putty so see a screen like that in Figure C-2. In the menu on the left click on Serial (at the bottom) to see the screen in Figure C-3. Set the parameters to match those shown except enter your own COM port number.
Figure C-2 The Putty opening screen.

Figure C-3 Click on Serial to set up the serial parameters. Click on Session to go back to the main screen.
In the main screen select the Serial radio button to see the session you created. Use the Save button to name and save the session. Double click on the saved session name to open the port for communication.

![PuTTY Configuration](image)

**Figure C-4** Click on the Serial radio button to see the serial session. Use the Save button to name and save this session. Once it is saved you can click on the name to open the session. You will get an error if the USB is not connected and it cannot find the COM port.
References
1. STM32L0x1 Reference Manual RM0377.

2. STM32L0 Programming Manual PM0214


5. STM32L031x6 Data sheet

https://static.docs.arm.com/ddi0484/c/DDI0484C_cortex_m0p_r0p1_trm.pdf


9. STM32L0 Architecture

10. This link to the ARM Help pages gives a list of the assembly instructions for the M0 and M0+

11. ARM Compiler toolchain Assembler Reference, ARM DUI 0489C Copyright © 2010-2011 ARM. ID080411