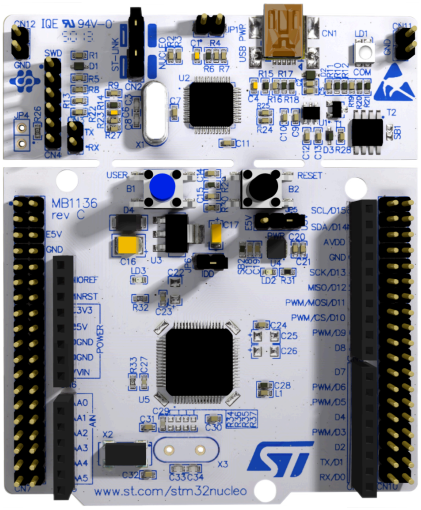
Microcontrollers and Embedded Applications Laboratory Manual

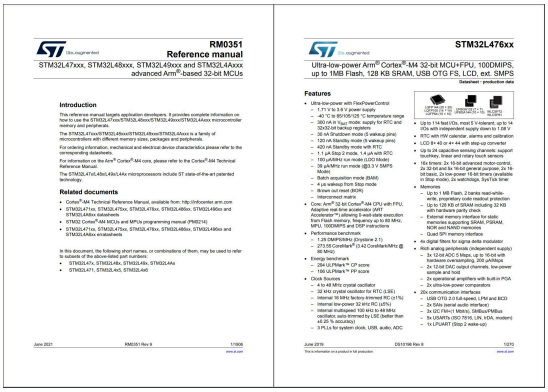
STM32L476RG / STM32L4A6ZG / STM32L496ZG



Paul Hummel and Jeff Gerfen
with contributions by Tamara Houalla
Winter 2024

Table of Contents

Lab Manual Resources


RM0351
Reference manual
STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm®-based 32-bit MCUs

Introduction

Related documents

STM32L476xx
Ultra-low-power Arm® Cortex®-M4 32-bit MCU+FPU, 100DMIPS, up to 1MB Flash, 128 KB SRAM, USB OTG FS, LCD, ext. SMPS

Datasheet - production data

Features

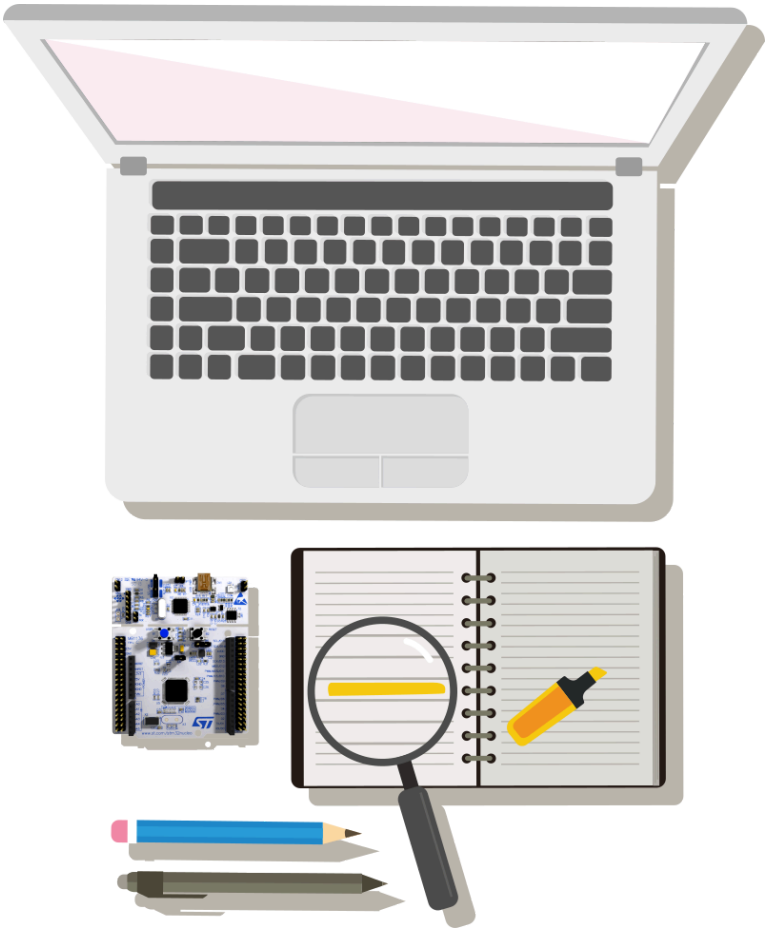
ST Reference Documentation:

- STM32L4xxxx Reference Manual — RM0351
- STM32L476xx Datasheet — DS10198
- STM32L4A6xx Datasheet — DS11584
- STM32L496xx Datasheet — DS11585
- NUCLEO-L476RG Users Manual — UM1724
- NUCLEO-L4x6ZG Users Manual — UM2179
- STM32CubeIDE Users Manual — UM2609

Lab Kit Part List:

- NUCLEO-L467RG / NUCLEO-L4A6ZG / NUCLEO-L496ZG
- 12 button matrix keypad (Recommend COM-14662)
- 12-bit SPI DAC (Recommend MCP4921)
- 256k EEPROM I2C (Recommend 24LC256)
- 16x2 LCD, parallel interface (preferably 3.3 V)
- 4 x LEDs
- 4 x 150 Ω Resistors
- 2 x 2 kΩ Resistors

Assignments



A0 – Getting to know you

Background Information

CPE 316 is a class where some of you may have some prior experience. You may have participated in robotics competitions, attended hack-a-thons, or have a good experience with Arduino. Even for the most experienced students, CPE 316 should be able to add to your understanding of microcontrollers and embedded systems development. And for those of you who have have little to no experience with embedded systems - now is your time to get that experience!

Embedded systems is not for everyone. If the frustration of debugging software is something you find unbearable, debugging *hardware and software at the same time* can be even more frustrating.

No matter where you are in embedded systems, we're all in this class together and are here to help each other learn. So might as well try to get to know a little more about one another.

Instructions

1. Post to the #chillzone channel in Slack following the prompts given (prompts are pinned in #chillzone). Then reply to at least two other student posts.

2.

A1 - Execution Timing

Reference Materials

- STM32L4xxxx Reference Manual (RM) – Clock System, GPIO
- NUCLEO-L4x6 Users Manual (UM) – Pin Diagram (L476RG/L4A6ZG)

Instructions

LED Counter

1. Connect 3 LEDs to GPIO pins PC0, PC1 and PC2. (LED and 150 Ω resistor)
2. Connect a button (or wire) to PA4. Configure the button to pull down if pressed. Configure the PUPDR bits for PA4 for pull-up (why pull-up - make sure you understand this).
3. Write a program that repeatedly counts 0 to 7, outputting the count to the 3 LEDs if PA4 is pulled low. Do nothing if PA4 is pulled high.
    a. Include a software delay between counts to make the counting visible. You may use `HAL_Delay();`

Measure Execution Timing

1. In a new project, add the code listing below (ExecutionTime.c) to main.c to replace the auto-generated main(). This code should only replace the main() function. Keep the other functions that are auto-generated with a blank project.
2. Set up an oscilloscope or logic analyzer to catch a single pulse on PC0
3. Change var_type and TestFunction(num) type to test the various variable types and operations listed in Table A1.1.
4. Run the code and fill in the table with the timing results obtained by measuring the duration of the high pulse on PC0

Timing Function	uint8_t	int32_t	float	double
Subroutine Call				
test_var = num + 1				
test_var = num * 3				
test_var = num / 3				
test_var = sqrt(num)	**NA**			
test_var = sin(num)	**NA**			

Table A1.1: Execution Timing Table

Deliverables

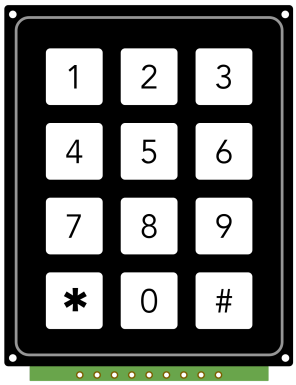
1. Take a video of your working A1 and include a link to the video on your submission
2. Submit your properly formatted C source code file for the LED counter.
3. Write a sentence or two on what you learned from the measurement of the execution timing exercise.
4. Fill out the execution timing table, submit a PDF on Canvas

**Properly formatted implies properly tabbed with sufficient comments and adjusted so code doesn't wrap, excess white space removed for a compact presentation, following accepted code formatting standards, header comments at the top of each new file to assist the reader in quickly interpreting what they are looking at, etc.*

ExecutionTime.c

```c
#include "main.h"
#include <math.h>

void SystemClock_Config(void);
var_type TestFunction(var_type num);

int main(void)
{
  var_type main_var;

  HAL_Init();
  SystemClock_Config();

  // configure PC0, PC1 for GPIO output, push-pull
  // no pull up / pull down, high speed
  RCC->AHB2ENR   |=  (RCC_AHB2ENR_GPIOCEN);
  GPIOC->MODER   &= ~(GPIO_MODER_MODE0 | GPIO_MODER_MODE1);
  GPIOC->MODER   |=  (GPIO_MODER_MODE0_0 | GPIO_MODER_MODE1_0);
  GPIOC->OTYPER  &= ~(GPIO_OTYPER_OT0 | GPIO_OTYPER_OT1);
  GPIOC->PUPDR   &= ~(GPIO_PUPDR_PUPD0 | GPIO_PUPDR_PUPD1);
  GPIOC->OSPEEDR |=  ((3 << GPIO_OSPEEDR_OSPEED0_Pos) |
                      (3 << GPIO_OSPEEDR_OSPEED1_Pos));
  GPIOC->BRR = (GPIO_PIN_0);   // preset PC0, PC1 to 0


  while (1)      // infinite loop to avoid program exit
    main_var++;  // added to eliminate not used warning
    GPIOC->BSRR = (GPIO_PIN_0);               // turn on PC0
    main_var = TestFunction(15);              // test function being timed
    GPIOC->BRR = (GPIO_PIN_0);                // turn off PC0
}

var_type TestFunction(var_type num) {

  var_type test_var;                      // local variable

  { insert_function_here (ie test_var = num;) }

   return test_var;
}
```

A2 - Keypad



Reference Materials

- STM32L4xxxx Reference Manual (RM) – GPIO
- NUCLEO-L476RG Users Manual (UM) – Pin Diagram (L476RG)
- Keypad Datasheet

Keypad

The keypad is a completely passive device made up of 12 (or 16) buttons. To reduce the number of wires and connections, the buttons are connected in a matrix of rows and columns as shown in Figure 1 below. Pressing a button on the keypad does not set or create a high or low voltage. The button press makes a connection between a row and column wire. This means that a button press can not be determined simply by reading from a single GPIO port. Determining a button is pressed requires finding which row and column is connected.

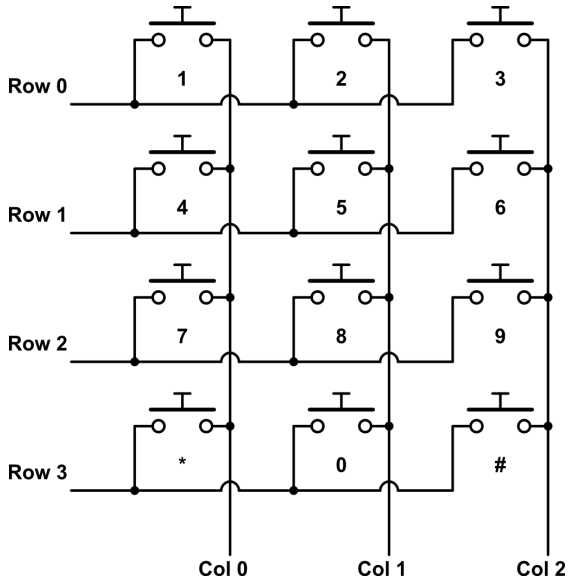


Figure A2.1: 4x3 Keypad Schematic

Detecting Key Presses

Because the STM32L4 does not have dedicated hardware to interface with the keypad, it is limited to interacting with the keypad through polling or interrupts. Polling is performed by repeatedly checking for a button press until one is detected. Determining the button press on a keypad can be performed in 2 stages. The first stage is detecting when a key is pressed while the second is determining which key is pressed. First detecting a key press can be done by setting all of the columns (or rows) high and monitoring the rows (or columns). By setting all of one dimension of the keypad matrix high (columns or rows), any key press will send one of the wires of the other dimension (rows or columns) high when the button press makes a connection. For example, if Column0 – Column2 is set high, pressing button 8 will cause Row 2 to go high because it will make a connection between Column 1 and Row 2. Notice detecting a button is being pressed does not mean it is possible to determine which key was pressed. Pressing button 7 or 9 will also cause Row 2 to go high the same way as button 8. This requires the second stage of the process to determine specifically which button was pressed. (*Hint: This could be utilized to make the keypad interrupt driven*)

In the above example, the difference between buttons 7, 8, and 9 can be made only by determining which column is being connected to Row 2 by the button press. This determination can be made by selectively cycling which columns are set high and reading the rows. If button 8 is pressed, setting Column 0 high while setting the others low will not result in Row 2 being high. This eliminates button 7 as a possibility. Only by setting Column 1 high will Row 2 go high signifying a connection at button 8. Once a button press has been detected in stage 1, the columns (or rows) can be individually cycled while reading the other.

Pull-up / Pull-down Resistors

When no key is pressed on the keypad, the connections between the rows and columns are open. When this occurs, the input signals being read from the rows (or columns) will not be connected to a low or high voltage. This means the input signal will float. Floating signals can create havoc on digital circuits because when the inputs on a digital signal float they may be read as low (0) or high (1). This will create unpredictable behavior in reading keypresses when no keys are pressed. To keep the inputs from floating, the input signals can be forced to go high or low rather than float. This is accomplished by adding resistors that pull the signal up or pull the signal down. These resistors are called pull-up or pull-down resistors. When using pull-up resistors, the button connection would need to drive the input low. When pull-down resistors are used, the button connection would need to drive the input high. In the example behavior described above, the button connections would drive the input high, so pull-down resistors will be needed. Built-in resistors are available on all of the GPIO pins of the STM32L4 and can be configured as either pull-up or pull-down.

Instructions

1. Decide which port(s) to use on the STM32L4 to connect to the keypad. Be mindful that having the rows (4 pins) and columns (3 pins) each being continuous bits on the port will make the program logic simpler to code. The NUCLEO-L476RG has a total of 49 GPIO pins available on

the ST morpho extension pin headers CN7 and CN10. All 16 bits of PA, PB, and PC are accessible. (*Protip: define specific values for items like* ROW_PINS, COLS_PINS, NUM_OF_ROWS, NUM_OF_COLS, ROW_PORT, COL_PORT, and KEYPAD_PORT)

2. Write a keypad function that can determine keypresses and return the value of the key pressed. If no key is pressed, it should return an obvious error value like -1. (*Be mindful of using -1 and unsigned variables.)*

3. Write a main program that uses the keypad function in an infinite while loop and outputs the result when a key is pressed to the LEDs used in the counter program from A1. If no key is pressed the LEDs should not change from what they were previously. Extra keys like * and # will depend on what values you assign to them.

Deliverables

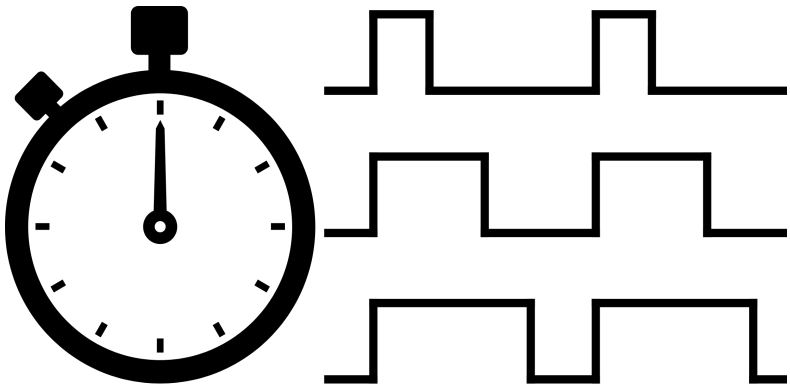
1. Take a video of your working assignment. Share a link to the video in your submission.

2. Submit all of your properly formatted C source code files.

**Properly formatted implies properly tabbed with sufficient comments and adjusted so code doesn't wrap, excess white space removed for a compact presentation, following accepted code formatting standards, header comments at the top of each new file to assist the reader in quickly interpreting what they are looking at, etc.*

A3 – LCD Display

Supports P1



Reference Materials

- STM32 Reference Manual (RM) – Clock System, GPIO
- FRDM-STM32 – Pin Diagram
- LCD Display / LCD Controller Datasheets

Software Delays

Controlling devices connected to a microprocessor often require specific timing for the control signals. For some devices the control timing can be easily achieved with software delays. In general software delays are not ideal for precise or efficient timing because the processor is kept busy doing nothing and cannot perform other operations while waiting. In some instances like device initialization or printing to a display this is less of an issue because there is nothing else the CPU could do until the necessary time has passed. Future assignments will cover better timing options available to the STM32. Software delays are created by having the processor perform a set number of operations. By controlling how many times this instruction occurs, the wait time can be configured. This is usually achieved with loops of nops (no operation assembly instruction). Adjusting the number of iterations of the loop allows the delay timing to be controlled. This method is intrinsically inaccurate with changing clock frequencies, compiler optimizations, and interrupts. More accurate timing can be achieved using hardware timers.

Delay Function

You should use your software delay functions from A4.

Instructions

1. Gather and review the datasheet for the LCD display. Be sure to look over the schematic, instruction table, initialization sequence, and write data timing diagram.

<u>Build the Circuit</u>

2. Draw a wiring diagram for connecting the LCD to the STM32 using 8-bit mode to start with. Make sure to label the pins as organized on the STM32 and LCD display. Note that most pins on the STM32 are not organized in order sequentially on the Freedom board. Refer to the

STM32 pin diagrams. Be sure to include any necessary power connections. Some LCDs include a V0 pin used to control the contrast on the screen that can be connected to ground through a potentiometer.

3. Construct the circuit. When wiring devices to the STM32 it is important to check all signals connected to the pins. Be careful not to exceed the maximum voltage of the STM32

Code Design

4. Write a short program to initialize the LCD and write a single character to the display. Note that *any time* you write a command or character to the LCD screen, you must make the E signal on the LCD toggle (latching the command or character on the falling edge of the E signal). Upon successful initialization, you should see a blinking cursor on the LCD. Some debug tips:
    a. If you don't see a cursor, make sure your contrast, power and ground voltages on the LCD module are correct. *note: when debugging digital systems, always check your power/ground and clock as appropriate.*
    b. Verify that you have wired the bus up correctly
    c. You may want to put your initialization routine in a large `while()` loop for debug purposes.
5. Modularize your code, writing useful functions to make later use of the LCD display easy to integrate. Minimum functions to be written include:

```
LCD_init(void);                 // initialize LCD
LCD_command(uint8_t command);   // Send LCD a single 8-bit command
LCD_write_char(uint8_t letter); // write a character to the LCD
```

Note that some #define can be used for specific commands like CLEAR_HOME, NEW_LINE, or CURSOR_BLINK.

Also note that you can combine the LCD_command and LCD_write_char to one LCD_write(unit8_t value, unit8_t mode) where value is then the data or command to write and mode is a single bit to indicate whether you are writing data or a command.

6. Write a program using your functions to print "Hello World" on the top line and "Assignment 3" on the bottom line of the LCD.

Hints

- LCD initialization must be exact. Verify the data bus and enable pulse timing with an oscilloscope or logic analyzer if it is not working.
- Be mindful of the timing for signals going low as well as high. For example, to create 2 pulses, timing delays would need to be added when the signal is set low as well as when set high.

- You may consider writing other functions that will make future use of the LCD easier, such as `LCD_write_string` which could reuse the `LCD_write_char` function to write a string of characters to the LCD.
- The timing specifications in the LCD are minimum specifications - when first developing, you can make the delays 10x normal to help account for any other issues.

Deliverables

1. Demonstrate your working program to the instructor or lab assistant
2. Copy all of your code (lcd.c, lcd.h, and main.c) into a text file - ensure that it is properly formatted (you can copy your code to tohtml.com to help with formatting first) and submit it as a pdf.

**Properly formatted implies properly tabbed with sufficient comments and adjusted so code doesn't wrap, excess white space removed for a compact presentation, following accepted code formatting standards, header comments at the top of each new file to assist the reader in quickly interpreting what they are looking at, etc.*

A4 - Interrupts and Timers

Reference Materials

- STM32L4xxxx Reference Manual (RM) – GPIO, General Purpose Timers, NVIC
- STM32L476xx Datasheet (DS) – Pinouts
- NUCLEO-L476RG Users Manual (UM) – Pin Diagram (L476RG)

Timers

Modern MCUs include hardware timers to keep precise timing without the need of software delay loops. This allows the CPU to process other instructions while “waiting” for the specified time. The STM32L4 has several different timers.

- Advanced Control Timers - TIM1, TIM8
- General Purpose Timers (32-bit) - TIM2, TIM5
- General Purpose Timers (16-bit) - TIM3, TIM4
- General Purpose Timers (Reduced features, 16-bit) - TIM15, TIM16, TIM17
- Basic Timers - TIM6, TIM7
- Low Power Timer - LPTIM1, LPTIM2

For this assignment, the most basic functionality of the general purpose timer TIM2 will be used. The core of any timer is a hardware counter that counts clock ticks. The most basic functionality of a timer is to trigger an interrupt when the count reaches a specified value. This allows easy and precise timing with an MCU without software delay loops. Timers typically include other features for multiple interterrupt time events, creating PWM signals, and measuring the timing of input signals.

Interrupts

An interrupt is a hardware mechanism in a CPU that can interrupt executing the current sequence of instructions and jump to a specific subroutine. Modern CPUs have multiple interrupts with each able to connect to a separate subroutine to allow each interrupt to handle a specific task (82 on the STM32L4). The advanced timers (TIM1, TIM8) have 4 different interrupts (some are shared with other timers). The general purpose timers each have a single interrupt. `TIM2_IRQHandler` is the interrupt handler for TIM2.

Instructions

Part A: 5 kHz with 25% Duty Cycle Square Wave

1. Setup TIM2 to count in up mode. Calculate the necessary AAR values to create a 5 kHz square wave using a 4 MHz input clock. Set CCR1 to interrupt at a 25% duty cycle. *It is recommended to draw out the 5 kHz wave along with a 4 MHz wave for reference. Be sure to specify period, high, and low time values of the 5 kHz wave.*
2. Create a 5 kHz with a 25% duty cycle square wave by setting a pin high and low with the TIM2 ISR. The code in main should only set up TIM2 and not keep track of any timing. No software delays should be used.
3. View this output on an oscilloscope and take a screen capture with measurements verifying timing.

Hints

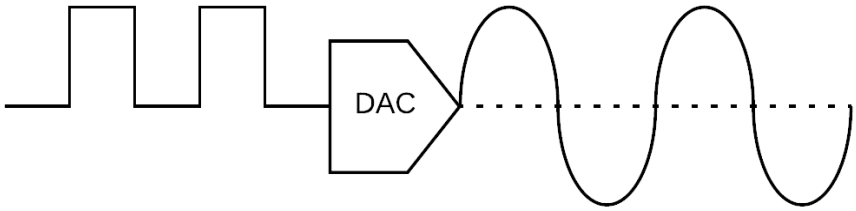
- Include counting 0 when calculating CCR / AAR values
- Be sure to clear the correct interrupt flag in the ISR.

Deliverables

1. Create a single pdf document (not a full lab report) containing the following:
    a. CCR value calculations from Part A
    b. Screen captures Part A – 5 kHz with 25% duty cycle square wave
    c. Properly formatted source code of:
        i. Part A – 5 kHz, 25 % duty cycle

**Properly formatted implies properly tabbed with sufficient comments and adjusted so code doesn't wrap, excess white space removed for a compact presentation, following accepted code formatting standards, header comments at the top of each new file to assist the reader in quickly interpreting what they are looking at, etc.*

A5 - SPI Digital Analog Converter



Reference Materials

- STM32L4xxxx Reference Manual (RM) – SPI
- STM32L476xx Datasheet (DS) – Pinouts
- NUCLEO-L476RG Users Manual (UM) – Pin Diagram (L476RG)
- Microchip MCP4921 Datasheet

SPI - Serial Peripheral Interface

Serial Peripheral Interface (SPI) is a common full-duplex, 4-wire bus used by a variety of digital devices. The STM32L4 provides multiple SPI peripherals. Configuration options can be reviewed in the corresponding RM chapter.

SPI Signal Naming Convention

SPI signals have historically been labeled using offensive and noninclusive master / slave terminology. There has been an ongoing effort to change the terminology in the electronics industry. While many peripherals have already adopted new terminology (including the DAC we use), most microcontroller documentation continues to use antiquated terminology. In this class I will be following the naming convention proposed by the Open-Source Hardware Association.

ST Terminology	Replacement Terminology
MOSI (Master Out / Slave In)	PICO (Peripheral In / Controller Out)
MISO (Master In / Slave Out)	POCI (Peripheral Out / Controller In)
SS (Slave Select)	CS (Chip Select)

DAC - Digital to Analog Converter

For this assignment you will use a Microchip DAC (digital to analog converter). The MCP4921 is a 12-bit DAC, allowing for 4096 different output voltages, including 0v. The MCP4921 interfaces with an MCU via SPI. The DAC is a write only device so only 3 of the 4-wires from SPI are necessary. Details on how the MCP4921 is controlled can be found in its datasheet. Figure 1 below shows a sample timing diagram for setting the DAC to output a voltage corresponding to the 12-bit value 0x6A7, unbuffered, and with a gain of 1.

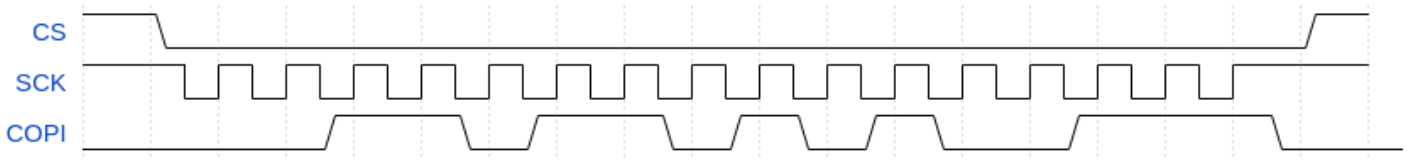


Figure A6.1 - SPI Timing Diagram for Unbuffered, 1x Gain, 0x6A7

Instructions

Interface STM32L4 and MCP4921

1. Connect the MCP4921 DAC to the STM32L4 via SPI. The STM32L4x6 datasheet and NUCLEO-L4x6 user manual can be used to find which pins are selectable for SPI operations.
2. Use 3.3 V for power and references for the MCP4921. *No 5 V signals are necessary and should be avoided to prevent any unintended overvoltage on the STM32L4 pins*.
3. Write some functions to utilize the DAC and verify they function properly
    a. DAC_init - initialize the SPI peripheral to communicate with the DAC
    b. DAC_write - write a 12-bit value to the DAC
    c. DAC_volt_conv - convert a voltage value into a 12-bit value to control the DAC
4. Verify the timing of the CS signal by using the logic analyzer to view CS, SCLK, and SDI on the DAC. CS should go low before SCLK starts and high after all SCLK pulses have stopped.
5. Use the DAC to generate a 25% duty cycle square wave with a low voltage of 1V and a high voltage of 2V. Measure this with an oscilloscope.

Hints

- Avoid the use of floats by treating voltage variables in units of mv. So 3.3 V becomes 3300 mV.

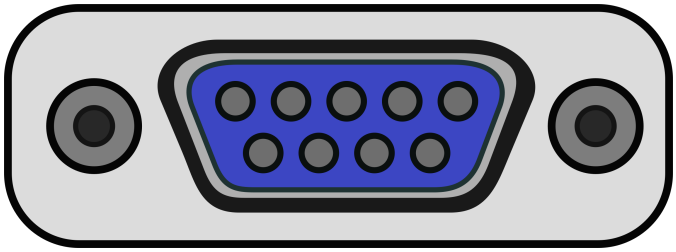
- The DAC may not output a voltage according to the ideal equation in the datasheet. So the conversion equation may need to be adjusted. This is a process called calibration. A technical note in the lab manual describes a process that can be used to calibrate a better fit.

Deliverables

1. Take a video of your working assignment. Share a link to the video in your submission.
2. Create a single pdf document (not a full lab report) containing the following:
    a. Image of single transmission to the DAC including CS, SCLK, and SDI
        i. Comment what data was being transmitted in the capture
    b. Properly formatted C source code

**Properly formatted implies properly tabbed with sufficient comments and adjusted so code doesn't wrap, excess white space removed for a compact presentation, following accepted code formatting standards, header comments at the top of each new file to assist the reader in quickly interpreting what they are looking at, etc.*

A6 - UART Communications



Reference Materials

- STM32L4xxxx Reference Manual (RM) – USART (L476) or LPUART (L4A6, L496)
- STM32L4x6xx Datasheet (DS) – Pinouts
- NUCLEO-L4x6xG Users Manual (UM) – Pin Diagram (L476RG, L4A6ZG, L496ZG)
- Technical Note - VT100 Serial Terminal

Debugger virtual COM port

The NUCLEO board routes a serial (RS-232) connection through the debugger and connectes using the same USB cable for loading a program and debugging. This creates a virtual COM port on the host computer used for programming that can be used to establish a serial connection using RS-232. On the NUCLEO-L476, the virtual com port is connected to USART2, using pins PA2 (TX) and PA3 (RX). On the NUCLEO-L4A6 and L496, the debugger is connected to LPUART1 using pins PG7 (TX) and PG8 (RX).

LPUART1 on STM32L4A6 and STM32L496

The LPUART1 on the L4A6 and L496 uses GPIO port G. The upper 14 pins of port G, PG[15:2] use a separate power supply that is isolated from the other GPIO pins, V_{DDIO2}. Before those pins can be used, a configuration setting must be set in the power control module (PWR). The IOSV bit in the CR2 register of the PWR module must be set to signify a proper voltage exists on V_{DDIO2}.

```
PWR->CR2 |= PWR_CR2_IOSV;
```

Instructions

1. Calculate the baud rate divisors for communicating via UART at 115.2 kbps with the clock running at the default speed of 4 MHz. Use oversampling if possible (Only available with USART2, not with LPUART).

Interface the STM32L4 to a Serial Terminal

2. Write some functions to utilize the LPUART1 or USART2 and verify they function properly with a terminal application. You can find a list of available options in Technical Note 4
    a. UART_init - initialize USART2 or LPUART1 device for serial communication
    b. UART_print - print a string of characters (Recall all strings of characters in C are terminated with a NULL or 0 value)
3. Verify the UART connection is working by repeatedly sending a single character. When the software terminal is properly connected, this character should be printed repeatedly across the screen.

Use VT100 Escape Codes

4. Change the program from printing a single character repeatedly to printing the following. This should only be printed once and in the order listed. (Pro tip: create a USART_ESC_Code function that is similar to the USART_print function but includes sending ESC)
    a. Escape codes to move the cursor down 3 lines and to the right 5 spaces
    b. Text “All good students read the”
    c. Escape codes to move the cursor down 1 line and to the left 21 spaces
    d. Escape code to change the text to blinking mode
    e. Text “Reference Manual”
    f. Escape code to move cursor back to the top left position
    g. Escape code to remove character attributes (disable blinking text)
    h. Text “Input: “

Echo Characters

5. Write an ISR that echoes whatever is received by transmitting it back to the software terminal. Typing a character in the terminal transmits it rather than causes it to be displayed on the screen in the terminal emulator. If you want to see what you are typing, the STM32L4 will need to echo the character it receives back to the terminal to print.

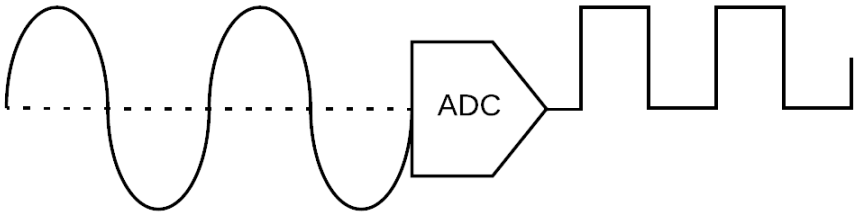
    If the character typed is “R” instead of echoing “R”, it should change the text color to red. It should do the same with “B” for blue, “G” for green, and “W” for white.
6. Verify this works by typing in the terminal and text should appear after “Input: “ in the terminal screen

Deliverables

1. Take a video of your working assignment. Share a link to the video in your submission.
2. Create a single pdf document (not a full lab report) containing the following:
    a. Baud Rate calculation
    b. Properly formatted C source code

**Properly formatted implies properly tabbed with sufficient comments and adjusted so code doesn't wrap, excess white space removed for a compact presentation, following accepted code formatting standards, header comments at the top of each new file to assist the reader in quickly interpreting what they are looking at, etc.*

A7 - Analog to Digital Converter



Reference Materials

- STM32L4xxxx Reference Manual (RM) – ADC
- STM32L4x6xx Datasheet (DS) – Pinouts
- NUCLEO-L4x6xG Users Manual (UM) – Pin Diagram (L476RG, L4A6ZG, L496ZG)

Connecting to the ADC

Connect an adjustable DC voltage source to an analog input pin on the STM32L4.

⚠️ **BEWARE OF VOLTAGES ABOVE 5 V!** ⚠️

Always be sure to double check the voltage before connecting or powering any analog voltage connected to the STM32L4. When connecting the external source, make sure to connect the ground of the source to the STM32L4 ground. Possible sources include power supplies, function generators, or wavegen outputs from the oscilloscope. Any source that you can reliably adjust a DC voltage from 0 to 3.3V will work. **To be careful, never set the voltage above 3.3V.** Also be sure to turn on the voltage source and set the voltage output *before* connecting it to the STM32L4. Many pins have been fried due to turning on a voltage source after connecting it to the STM32L4 only to realize that the output had previously been set to a value outside the acceptable range.

Instructions

1. Write some functions to utilize the ADC
    a. ADC_init()
        i. Run the ADC with a clock of at least 24 MHz.
        ii. Single conversion, not repeated, initiated with SC bit
        iii. Use sample and hold timer with a sample time of 2.5 clocks
        iv. 12-bit conversion using 3.3 V reference
        v. Configure analog input pin

b. ADC ISR

   i. Save the digital conversion to a global variable

   ii. Set a global flag

2. Write a program that initializes the ADC and then runs in an infinite loop checking for the global flag set by the ADC ISR. When the flag is set, it should save the converted value into an array, reset the flag, and start the ADC to perform another sample and conversion. After collecting 20 samples from the ADC, process the array to calculate the minimum, maximum, and average values. (*Protip: Use 16-bit variables for saving the samples and 32-bit variables for the calculations to avoid overflow errors*)

3. Connect the selected analog input pin to 1.0 V from the AD2 waveform generator (DC) or DC power supply (V+) and run your program. Adjust the voltage input from 0 to 3 V to verify the values from the ADC change accordingly.

Calibrate your ADC

4. Calibrate your ADC to get a voltage value from the digital conversion that matches the function generator or power supply. This value should be accurate (+/- 10 mV) for voltages 0.00 – 3.00 V. Approaches on how to derive a calibration equation can be found in the Technical Note on Calibrating ADC / Sensor. (*Protip: Avoid the use of floats by adjusting the calibration to result in units other than volts*)

5. Add this calibration calculation to the array processing, converting the resulting minimum, maximum, and average values into voltages. (*Protip: It is faster to only covert min, max, and average instead of converting all 20 values*)

Print to the Terminal

6. Print the calibrated voltages (Min, Max, Avg) to a terminal via USART. Remember that the USART can only transmit a single character at a time, so you will have to break down your calculated values into individual digits and transmit them sequentially. Printing to the terminal cannot make use of any stdio.h, stdlib.h, or string.h C library functions for converting numbers to strings. (No itoa(), sprintf() or similar allowed. ~~Sorry~~. Not sorry.) These values should be printed in units of volts with 2 decimal precision.

7. Using a 1.5 V input, adjust the ADC sample time to 47.5, and 640.5 clock cycles, recording the variation in the ADC values for each. Record each of these voltage values (Min, Max, Avg) into a table for submission.

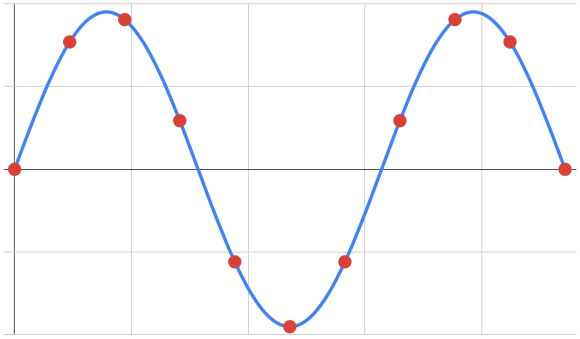
Deliverables

1. Take a video of your working assignment. Share a link to the video in your submission.

2. Create a single pdf document (not a full lab report) containing the following:
    a. Table of ADC sample time clocks and corresponding min, max, and average voltages.
    b. Properly formatted C source code

**Properly formatted implies properly tabbed with sufficient comments and adjusted so code doesn't wrap, excess white space removed for a compact presentation, following accepted code formatting standards, header comments at the top of each new file to assist the reader in quickly interpreting what they are looking at, etc.*

A8 - Frequency Measurement



Reference Materials

- STM32L4xxxx Reference Manual (RM) – ADC, General Purpose Timers, Comparator
- STM32L4x6xx Datasheet (DS) – Pinouts
- NUCLEO-L4x6RG Users Manual (UM)

Signal Measurement

Measuring input signals with an MCU typically involves more than just monitoring constant values or DC voltages. When analyzing different signals, characteristics beyond just the voltage value at a singular point are important. One of the most common aspects of a signal to characterize is its frequency. This is because properly sampling a signal is easier if the frequency is first known. If the signal has a frequency of 1kHz, sampling at a rate of 1 ms will be inadequate because only 1 data point per period is insufficient for any type of signal analysis. Sampling at 10 us will provide 100 points per period. If the signal was 10 Hz, sampling at this same rate of 10 us will require 10000 samples to capture the full period. Trying to analyze that many points would be computationally expensive, especially when similar precision results could be achieved with orders of magnitude fewer data points. The limited memory on a microcontroller will also limit the number of samples that can be saved.

This assignment will help practice and prepare for the next project by developing a strategy to measure the frequency of an input signal. Determining the frequency requires collecting multiple data points at consistent time intervals for analysis. To achieve quality measurements, calibrated use of timers in coordination with the ADC is needed.

Instructions

1. Expand on the previous ADC assignment to sample an analog input signal and determine its frequency.
2. Display the result in Hz on a terminal via UART.
    a. The value on the terminal should be updated at least every 2 seconds

3. For this assignment, the input signals will be limited to the following
    a. Input signals will have a sinusoidal waveform
    b. Input signal frequency will vary from 1 Hz to 1 kHz
    c. The maximum voltage of any input signal will be 3 V
    d. The minimum voltage of any input signal will be 0 V
    e. The minimum peak-to-peak voltage of any input signal will be 0.5 V
    f. The maximum DC offset value of any input signal will be 2.75 V

Hints

- Speed up the MCU, running at a minimum of 24 MHz.
- Run the ADC clock at the fastest clock rate to make the conversion process as quick as possible. Adjust the desired sample and hold time of the ADC accordingly.
- Use a timer to set the start conversion bit at regular intervals for uniform sampling.
- Building some circuits in hardware can be helpful and reduce the complexity of the software.
    - An analog circuit that could transform a periodic wave into a square wave of the same frequency can be used by a timer in capture mode to accurately measure the period.
    - The internal comparator peripheral in the STM32L4 can also be used with a simpler analog circuit, the DAC, or internal reference generator to achieve a similar result.
    - Adding hardware is not inherently easier or harder, it is just a different approach with different trade-offs. Adding hardware can reduce the complexity of the software that needs to be debugged and verified. However, the added hardware will require debugging and verification which a purely software approach would not.

Demonstration

Demonstrate by showing the minimum and maximum frequencies that can be measured (1 Hz to 1 kHz for full credit) with each input signal specified below

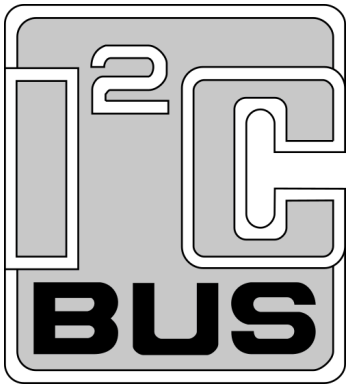
1. 3 Vpp sine wave with 1.5 V DC offset
2. 0.5 Vpp sine wave with 0.5 V DC offset
3. 1 Vpp sine wave with 2.5 V DC offset

Deliverables

1. Take a video of your working assignment. Share a link to the video in your submission.
2. Create a single pdf document (not a full lab report) containing the following:
    a. Schematic of your working device.
    b. Properly formatted C source code

**Properly formatted implies properly tabbed with sufficient comments and adjusted so code doesn't wrap, excess white space removed for a compact presentation, following accepted code formatting standards, header comments at the top of each new file to assist the reader in quickly interpreting what they are looking at, etc.*

A9 - I2C EEPROM



Reference Materials

- STM32L4xxxx Reference Manual (RM) – I2C
- STM32L4x6xx Datasheet (DS) – Pinouts
- NUCLEO-L4x6xG Users Manual (UM) – Pin Diagram (L476RG, L4A6ZG, L496ZG)
- Microchip 24LC256 EEPROM Datasheet

I2C - Inter Integrated Circuit

I2C is a common half-duplex, 2-wire bus used by a variety of digital devices. The I2C protocol is more complex than SPI or RS-232 allowing for 2-way acknowledgments and addressed devices, The STM32L4 provides multiple I2C outputs via the I2C peripherals. Configuration options can be reviewed in the corresponding TRM chapter.

EEPROM - Electronic Erasable Programmable Read-Only Memory

EEPROM is a type of nonvolatile memory that can be erased electronically. Memory is considered nonvolatile if the data is retained between power cycles. More prevalent flash memory is a type of EEPROM, but the designation EEPROM is used to distinguish memory that is byte erasable. Flash memory can only be erased in larger blocks. The 24LC256 EEPROM is a 256 Kbit (32k x 8), so it is byte addressable with 32k (2^{15}) addresses. It has a configurable device address that ranges from 0x50 to 0x57 based on how it is wired. The EEPROM communicates with the microcontroller via I2C bus with a maximum speed of 400 kHz. Details on how data can be read from and written to the EEPROM are in its datasheet.

Instructions

Interface STM32L4 and EEPROM

1. Review the EEPROM datasheet and select a device address to use in the range 0x51-0x57. (Do NOT use 0x50)

2. Draw a schematic connecting the 24LC256 EEPROM to the STM32L4 with I2C. The datasheet (DS) can be used to find which pins are selectable for I2C operations.
3. Wire the schematic on the breadboard
4. Use STM32IDE to calculate the I2C timing factor for the clock and I2C speed. ***Do NOT save the configuration, just make note of the timing factor!***

Write a program to use the EEPROM

5. Write some functions to utilize the EEPROM and verify they function properly
    a. EEPROM_init - initialize an I2C peripheral to communicate with the EEPROM
    b. EEPROM_read - read a byte of data from a given 15-bit address
    c. EEPROM_write - write a given byte of data to a given 15-bit address
6. Use the above functions in a program to write a random byte of data to the EEPROM at a random 15-bit address. (Your program does not need to generate random values, they can be hard coded into your program, but the values you select should be “random”. You can use https://www.random.org/bytes/ to select random values to use) Wait 5ms for the data to be saved then read a byte of data from the same address. Compare the data read with the data sent to be saved and If the values match, the onboard LED should turn on, and if not, the LED should stay off.

Analyze the I2C Bus

7. Take two screenshots of the SCL (I2C clock) and SDA (I2C data) using the logic analyzer. Write on the screenshots or otherwise mark the start, restart, stop, device address, R/W, ACKs, NACKs, and data being transmitted.
    a. Entire transmission for writing a byte of data
    b. Entire transmission for reading a byte of data

Hints

- If errors occur during transmission and a STOP condition is not transmitted, the EEPROM will stay in the same state, waiting for data or a stop condition to release the bus. This can cause the STM32L4 to see the bus as occupied when running or rerunning code on a successive attempt. It will then wait indefinitely before sending out a new START. This can also cause the EEPROM to not acknowledge when its address is sent by the STM32L4, which can affect how the STM32L4 handles sending data after a start condition. To avoid these issues, power to the EEPROM can be quickly cycled by removing and replacing the 3.3 V wire connected to the VCC pin.

- The EEPROM requires a 5 ms delay after writing data before any other operations can occur. Failure to keep this delay after writing will cause the EEPROM to behave unexpectedly.

Deliverables

1. Create a single pdf document (not a full lab report) containing the following:
    a. Take a video of your working assignment. Share a link to the video in your submission.
    b. Screenshots of I2C bus analysis
    c. Properly formatted C source code

A10 - Diversity, Equity and Inclusivity

1. Do a web search on “unknown heroes of engineering”, preferably someone related to computer engineering.
2. Use an AI generator tool to write you a 2-page draft report on this unknown hero.
3. Critique the AI generator output by providing a list of three places the report could be improved.
4. Re-write the AI generator tool’s output with your improvements

Deliverables:

1. A single PDF that includes:
    a. Your 2-page report
    b. A picture (if any) of the subject
    c. A list of your critiques (3) of the AI generated output
    d. A copy of the prompt used to generate your first draft
    e. The raw output from the AI generator tool

Rubric:

	Excellent (1)	Good (.75)	Fair (.5)	Poor (.25)	N/A (0)
Writing Quality					
Importance of the Subject					
Quality of your critique of the AI generated output					

A11 - Ethics

1. Browse the papers listed at: https://ethics.calpoly.edu/publications.htm
2. Choose a paper to read and critique
3. Write a one-page summary of your chosen paper
4. Prompt an AI generator tool to help expand your paper
    a. Provide a paragraph critically evaluating the output of the AI generated output (½ page)
    b. Provide the prompt and the raw output text at the end of your report
5. Write ½ page of your own opinion on the chosen subject

Deliverables: submit your 2+ page report to Canvas. For full credit, you will be assigned peer-reviews to complete (exact timing depends on when students submit their reports). The rubric in Table A11 will be will be used (copied from Canvas).

Criteria	Ratings				Pts
This criterion is linked to a Learning Outcome Identified Problem	0.5 pts Identified problem accurately	0.35 pts Mostly identifies the problem	0.15 pts somewhat identifies problem	0 pts Problem statement va or unclear	0.5 pts
This criterion is linked to a Learning Outcome Consider Stakeholders	0.75 pts Accurately identifies several stakeholders and reflects on their viewpoints/motivations	0.55 pts two stakeholders with differing viewpoints identified	0.2 pts A stakeholder is identified and reflects a viewpoint	0 pts Unsure who is involved/affected	0.75 pts
This criterion is linked to a Learning Outcome Considers Ethics Theories and Codes	0.25 pts Thoroughly links problems to one or more ethical codes		0 pts Does not or poorly links ethical codes to problems		0.25 pts
This criterion is linked to a Learning Outcome Analyzes Alternatives and Consequences					0.75 pts

	0.75 pts Many alternatives and consequences accurately considered.	0.55 pts Two alternatives or consequences considered	0.2 pts One alternative or consequence considered	0 pts No alternatives provi or consequences considered	
This criterion is linked to a Learning Outcome Chooses / Justifies and action	0.75 pts Identifies a preferred action out of a range of possibilities	0.4 pts Identifies a preferred action, but not a range of possibilities. Or identifies a range of possibility without a preferred action.		0 pts Does not clearly identify a choice action	0.75 pts

A12 - Timer Input Capture

Use the timer input capture mode to measure the width of a periodic pulse train as generated by the bench-top oscilloscope. The pulse train should be a frequency of 50 Hz with a variable duty cycle. You should display the duty cycle on your LCD screen and the displayed duty cycle should be within 5% of nominal.

Do not use the HAL for this assignment.

Deliverables:

1. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. Your code listing, properly formatted

A13 - Timer PWM

Use the timer PWM mode to generate two PWM waveforms. The waveforms must be non-overlapping and be of 25% duty cycle at 50 Hz. Verify the PWMs using the benchtop oscilloscope.

Do not use the HAL for this assignment.

Deliverables:

2. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. Your code listing, properly formatted

A14 - Direct Memory Access

Develop a project where you are using the DMA to write an array of data out the SPI, where the SPI is connected to the DAC from project 2. The DMA should transfer 128 12-b data samples out to the DAC. A sample should be sent to the DAC once every 10 mS. Use TIM2 to trigger the DMA. Use the scope to verify that the DMA is functioning correctly.

Deliverables:

3. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. Your code listing, properly formatted

A15 - Low Power Modes

Your task is to design an experiment to measure the power consumption of the STM32 board in one of three modes: 32 MHz operation, Sleep mode and Stop 2 mode. The code you run is up to you. It is suggested that you use a bench power supply to power the STM32 microcontroller (the bench power supply also measures current). See pages 21-22 of the Nucelo board manual to see how to power the board using the VIN pin.

Deliverables:

1. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. A listing of your power consumption measurements at the three different operating modes.
    g. Your code listing, properly formatted

A16 - Pin Interrupts

Using the EXTI, enable pin interrupts. When a pin transitions from high to low, create an interrupt to turn on the LED on the Nucelo board.

Deliverables:

1. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. Your code listing, properly formatted

A17 - Comparator

Design an experiment to use the STM32's on-chip comparator in window mode, where if the input voltage is between ¾ VREFINT and ½ VREFINT (VREFINT = 1.21 V), turn on the LED on the Nucleo board.

Deliverables:

1. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. Your code listing, properly formatted

A18 - Random Number Generator

Using the RNG on the STM32, design a system that displays the 32b RNG on the LCD screen.

Deliverables:

1. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. Your code listing, properly formatted

A19 - AES

Demonstrate the use of the STM32's on-chip AES encryption and decryption accelerator. You only need to encrypt a single 128b piece of data with a 128b key (ECB mode). Use the LCD screen to show the plain text, the ciphertext and the key. The block encryption should be triggered when the button on the Nucelo board is pressed. When the button is pressed again, the data should then be decrypted.

Deliverables:

1. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. Your code listing, properly formatted

A20 - Watchdog Timer

Design a system that uses the Watchdog timer (IWDG) to reboot the STM32. You can use a minimal prescalar and reload value to demonstrate the IWDG's operation. When the STM32 boots, turn on the LED on the Nucelo board to show that the system has rebooted.

Deliverables:

1. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. Your code listing, properly formatted

A21 - SDMMC

No one has done this yet - feel free to design your own experiment.
Deliverables:

1. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. Your code listing, properly formatted

A22 - USB OTG

No one has done this yet - feel free to design your own experiment.
Deliverables:

1. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. Your code listing, properly formatted

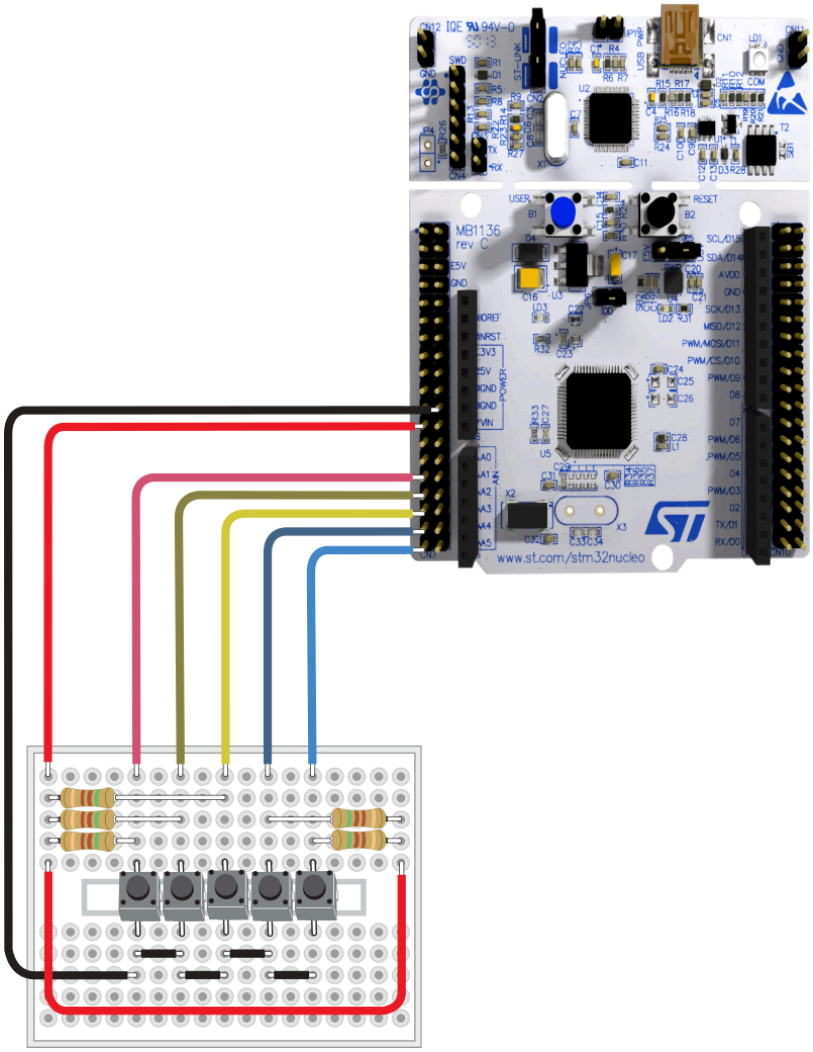
A23 - RTC

Design an alarm clock with a settable time/day for the alarm. When the alarm is triggered, turn on the LED on the Nucelo board.

Deliverables:

2. A PDF report that includes:
    a. Name of students who did the work
    b. A video link to a demo of your working assignment
    c. Block diagram of your system, including which pins on the STM32 you are using
    d. A paragraph describing the block diagram
    e. A paragraph describing your code
    f. Your code listing, properly formatted

Projects



Project Report Formats

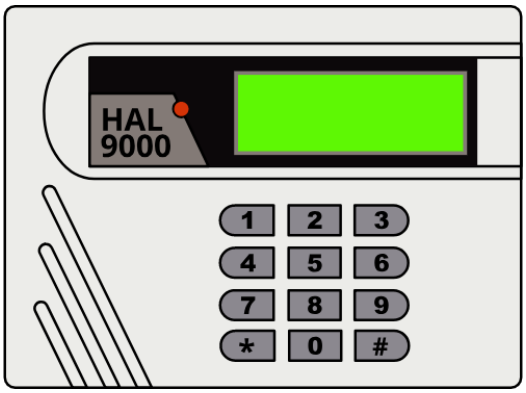
All projects must follow the project report format. The purpose of the project reports is to document your project so that if another engineer were to read your report, they would be able to rebuild your project without any guessing or interpretation.

1. Overview: 1-2 paragraphs giving an executive summary of the project.
2. Top-level system diagram: A high-level block diagram of your system, and 1-2 paragraphs describing the system
3. Hardware description: Detailed breakdown of the system. Don't forget to describe wiring, and all necessary components. This section should have detailed diagrams that show pin numbers. Each diagram must have 1-2 paragraphs describing each diagram.
4. Software description: Multiple flowcharts, call diagrams or UML diagrams. Start high-level (likely documenting your main function), then create more diagrams for each of the functions. Each diagram must have 1-2 paragraphs describing the diagram.
5. Operation: Description of how your device operates.
6. Reflection: One paragraph for each of the following questions.
    a. If you were to do the design again, what would you have done differently
    b. What was something that was surprisingly difficult? What was something that was surprisingly easy? Why were they difficult/easy?
    c. Ethics - what are ethical concerns about your project? (e-waste, privacy, security, etc.)
7. Appendix:
    a. Code (all code you wrote) - please make sure you syntax highlight your code and avoid line-wrap (80 characters max width)

Make sure you label and reference all figures in your report. Example:

Figure 1. My fat dog

Every Christmas, we dress up our dog as shown in Fig. 1. He looks like a sausage and looks unhappy in Fig. 1, but he actually loves the attention.



P1 – Digital Lock

Reference Materials

- STM32 Reference Manual (RM) – GPIO
- STM32476xx Datasheet (DS) – Port / Pin Function Tables
- Keypad Datasheet
- LCD / LCD Controller Datasheets

Introduction

The purpose of this project is to integrate the LCD module and the keypad to create a virtual electronic lock. Instead of connecting a physical lock mechanism to the microcontroller, the onboard LED will function as the locking mechanism. When the lockbox is locked, the LED will be on, and when the box is unlocked, the LED will be off.

Objectives

1. Integrate the LCD and keypad to run together on the Nucelo board (STM32)
2. Develop an application which allows the user to “open” or “lock” the lockbox using the keypad
3. Create library files (pair of .h and .c source files) for the LCD and keypad functions.

System Requirements

- The digital lock will utilize a pin number of at least 4 digits.
- The digital lock device will power up in a locked state.
- The LCD will display the status of the lock.
- The LCD will display the numbers as the pin is being entered.

- When entering the pin, pressing the * key will clear the numbers entered and restart the entering process.
- When the lockbox is locked, the onboard LED will be on and when the lockbox is unlocked, the LED will be off.
- After unlocking, the lockbox should be able to be relocked without requiring a new pin.
- The pin should be reprogrammable when the box is unlocked.
- The LCD will utilize both rows of the display.
- The LCD and Keypad will be implemented as separate library source files. (Header / Source files as shown in Technical Note 2)
- Example LCD screens (*these do not have to be recreated exactly, they are just examples*)

LOCKED ENTER KEY:	LOCKED ENTER KEY 2345	UNLOCKED PRESS KEY TO LOCK

Suggested Design Steps

1. Draw a single schematic diagram of the keypad and LCD connected to the LaunchPad development board. Be sure that the LCD and keypad do not utilize the same pins.
2. Create library files for the LCD and keypad and verify they each work before writing code for the lockbox.
3. Design the lockbox as a state machine.
4. Implement (aka code) the project in small segments that can be tested individually. For example, display a specific screen on the LCD, enter a multidigit number on the keypad, and change the pin number combination.

Suggested Approach

Successful completion of this project requires you to assimilate a variety of information from different sources and implement a system solution. Although this process may seem daunting at the start, a methodical step-by-step approach can be helpful. Here are suggested steps for your project implementation.

1. Give all datasheets and reference manuals a read to verify your understanding of both devices/components and the reference material available to you. There is no substitute for reading the datasheets to get started. Aside from understanding key aspects of the device required for successful interfacing and operation, reading the datasheets will potentially

illuminate extended or extra features of the device which may be advantageous to the designer. While I was able to cover the highlights of each component in class, we did not have sufficient time to cover all of the details. Most questions can be answered by reading the documentation.

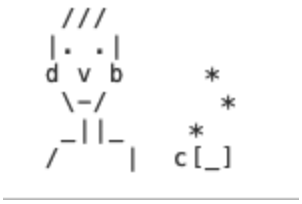


Figure P1.1: Dr Hummel drinking coffee and answering student's questions

2. The LCD Module requires 7 GPIO pins when running in nibble mode, the keypad requires 7 (or 8) additional GPIO pins. The Freedom has a total of 49 GPIO pins available on the ST morpho extension pin headers CN7 and CN10. You should consider the best method for wiring and writing your program interfacing with each peripheral before deciding how to connect them to the STM32L476.
3. Utilize accurate schematic diagrams to guide your work. The final schematic for this project should include the STM32L476, the LCD display, the keypad, and power connections. Common schematic practices include:
    a. Labeling each IC, component, etc. with a reference designator, e.g. U1, R1, C21, etc. For ICs, this label can be either inside the IC or directly above or below the IC.
    b. Label each chip/component with the value/part number or full model number for the component, e.g. 10K, .01uf, STM32, etc.
    c. Numbering all pins above the trace outside of the IC. Refer to the IC diagram in the CubeIDE for their number.
    d. Labeling the IC pin name, e.g. P1.0, CS, VCC, inside the chip where the wire attaches to the chip.
    e. Labeling all wires with a signal name, e.g. A21, D7, CLK, etc.

    The process of drawing an initial schematic does not have to be long or involved. It can be a quick effort using pencil and paper in which key information such as pin names and numbers are given to aid in wiring and troubleshooting. A formal schematic can be constructed at a later time once the design is finalized. (Refer to Technical Note 3)
4. **Do not skip designing your program before coding!** Sketch out what the software will do. Aids such as flow charts, state diagrams, pseudo-code, etc. may be used to do this.

5. ***After*** you have your complete program logic designed, you can start the actual code writing in IDECube. You can start by creating #define constants that will be useful. (For example #define DEFAULT_PIN, LOCKED)

6. Write your program in modules with separate functions that perform repeated tasks. Write a function to write to the LCD display and verify it works properly. Then write a function to read the keypad. The keypad functionality can be verified by writing values to the LCD with the functions you already have working. Then build your application as you designed in your flowcharts. If you run into an issue with the logic of your original design, go back and redesign your program with a new flowchart. Do not try to change aspects of your program without considering how it functions as a whole. This is hard to do when only looking at the code!

7. Comment and document your code as you write it. Comments are doing your future self a favor now.

8. Don't forget to make plenty of backups along the way to preserve the good work that has been accomplished. It is not a good feeling to get a major part of the system working, continue working on the code, break the code, and then have to spend time fixing what was broken. Making frequent backups will allow the developer to roll back to a known development state when something goes wrong. Although there are various comprehensive version control products available for this purpose, a simple approach such as maintaining version directories of a development project on Dropbox, Google Drive, or OneDrive or similar are satisfactory for projects such as this.

Hints

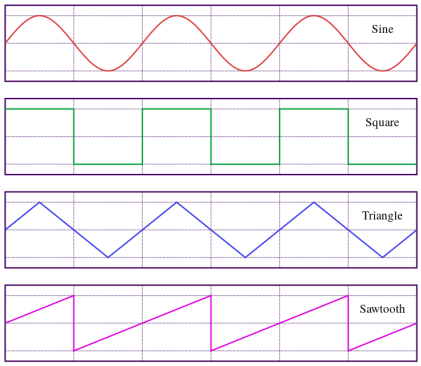
- When interfacing the keypad and LCD, note the ASCII value for '0' is 0x30, so all single digit numbers can be converted from their value (0-9) to their equivalent ASCII character by adding 0x30. (*Pro tip: define values for * and # using #define for STAR and POUND (ok, ok HASHTAG* 🙄*))*

- A while loop can be used to continuously read the keypad until a key is detected. (*Pro tip: a while loop can also be used to detect when a button has been released)*

Questions:

1. How fast (in bytes-per-second) can you theoretically write to the LCD? Use the datasheet to help you.
2. How much power does your system consume? Use estimates based on the datasheets of the STM32 and LCD for typical operation.

P2 - Function Generator



Reference Materials

- STM32L4xxxx Reference Manual (RM) – General Purpose Timers, SPI, GPIO
- Keypad Datasheet
- Microchip MCP4921 Datasheet

Introduction

The purpose of this project is to design a function generator using a microcontroller. The microcontroller should be connected with an external DAC to generate the analog waveforms required. This DAC will have an SPI interface. The waveforms that the function generator must generate include a saw tooth waveform, a square wave with a variable duty cycle, and a sinusoidal waveform. The frequency of the waveforms will also be variable. The keypad will be used to select the output waveform type, set the frequency of the waveform and set the duty cycle of the square wave.

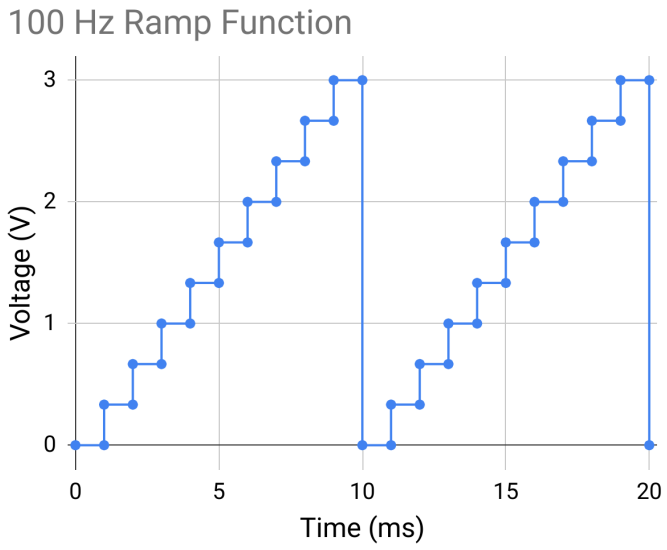
Objectives

1. Understand how to use timers and interrupts to generate timing events
2. Understand how to use a serial peripheral interface (SPI)
3. Understand how to use digital-to-analog converters (DAC) to generate analog signals from a microcontroller

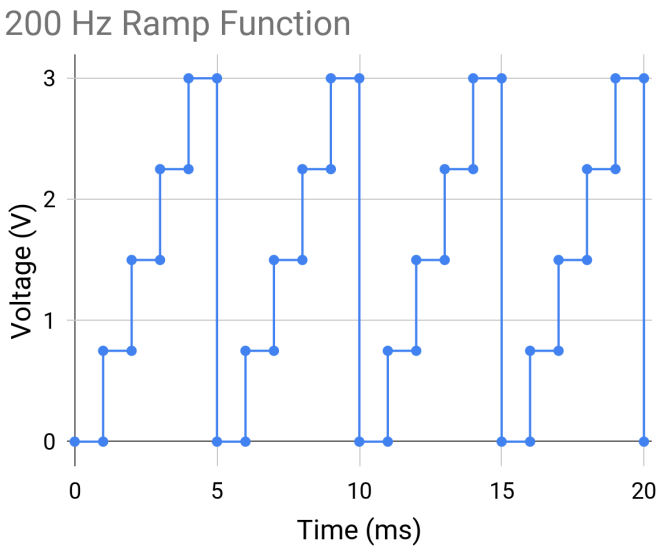
Creating Waveforms

Waveforms can be created by the DAC by changing the output voltage at consistent time intervals. This would be like plotting points on a graph. By plotting points at equal intervals, the resulting graph will look uniform and be easy to control. Instead of changing the frequency of the graph by changing the interval between points on the graph, the frequency can also be adjusted by changing how much each point is increased or decreased. Look at Graph P2.1 and P2.2 below. Graph P2.1 is a 100 Hz ramp function. Graph P2.2 is a 200 Hz ramp function. Notice both graphs adjust values every 5 ms. The 100 Hz ramp function increases voltage by 0.3 V for each point while the 200 Hz ramp increases

by 0.6 V. This means that lower frequency waveforms will look better than higher frequency ones. *This is preferable to having all of the waveforms look equally poor as the highest frequency ones.*



Graph P2.1 - 100 Hz Ramp



Graph P2.2 - 200 Hz Ramp

Timing Calculations

The maximum resolution possible with the DAC is based on the time to output a single value to the DAC. You could find this by setting a bit high before calling your previously written DAC_write() function to set the DAC and then setting the bit low after the function call. This would define the maximum resolution if driving the DAC repeatedly without any timing. While this would be fast, it would make precise timing control difficult if not impossible.

To precisely control the timing of output waves, the DAC will need to be set at controlled intervals using a timer. This will add some overhead and reduce the maximum resolution. The time to run the timer ISR can be found in a similar manner with toggling a bit. Create an empty TIM2 ISR that only clears the interrupt flag and calls the DAC_write() function. At the beginning of the ISR set a bit high and then set it low at the end. Measuring this time on an oscilloscope will give you the minimal time for plotting a point of the waveform. This will not take into account the time to get into and out of an ISR. However, from the previous assignment using the timer you have already found this overhead timing. In the previous assignment you counted the number of MCO clock pulses to execute the ISR but found that the minimal CCR1 value you could use did not match. The difference between those two values was the overhead of getting into and out of the ISR. If you add the time for those clock pulses to the pulse time measured above, you will have the minimum time or maximum resolution possible.

System Requirements

1. Document the maximum resolution (samples per second) using the timing measurements and accompanying calculations. (Add to the Software Architecture section of the report.)
2. The function generator shall use a microcontroller with an external DAC, and the DAC will be implemented as a separate library (Header / Source files as shown in Technical Note 3)
3. The function generator will be capable of producing
    a. A square wave with a variable duty cycle
    b. A sinusoidal waveform
    c. A sawtooth (ramp) waveform
    d. A triangle waveform
4. All waveforms will have a Vpp of 3.0 V and be DC-biased at 1.5 V.
5. All waveforms will have adjustable frequencies:
    a. 100 Hz, 200 Hz, 300 Hz, 400 Hz, 500 Hz
    b. The frequency of the waveforms should be within 2.5 Hz of these frequencies for full-credit.
    c. The function generator shall have an output rate (points / sec) of at least 60% of the maximum determined from (1) above.
        i. If the minimum time from (1) was 50 us, the maximum resolution is 20,000 samples per second. The requirement is to be within 60% of that resolution, or 12,000 samples per second. This is equivalent to 83.33 us between samples.
    d. The output rate or samples per second will not change with waveform frequency. The time between outputs from the DAC (time between samples or points on the graph) should be the same for all frequencies. This means the 100 Hz waveform will output 5x the number of points in one period as the 500 Hz waveform.
6. Upon power-up, the function generator will display a 100 Hz square wave with 50% duty cycle.
7. The keypad buttons 1-5 will set the waveform frequency in 100 Hz increments (1 for 100 Hz, 2 for 200 Hz, etc).
8. The keypad buttons 6, 7, 8, and 9 will change the output waveform.
    a. The 6 key will change the output to a sine waveform
    b. The 7 key will change the output to a triangle waveform
    c. The 8 key will change the output to a sawtooth waveform

    d. The 9 key will change the output to a square waveform

9. The keypad buttons ✱, 0, and # will change the duty cycle of the square wave.
    a. The ✱ key will decrease the duty cycle by 10% down to a minimum of 10%.
    b. The # key will increase the duty cycle by 10% up to a maximum of 90%
    c. The 0 key will reset the duty cycle to 50%
    d. The keys ✱, 0, and # will not affect the sine, sawtooth, or triangle waveforms.
10. You may not use software delays (ex: delay_us) to generate timing events or control when voltages are changed by the DAC. (*It would actually be more difficult to use software delays.*) You may use software delays to help debounce buttons (if necessary).

Suggested Approach

Successful completion of this project requires you to assimilate a variety of information from different sources and implement a system solution. Although this process may seem daunting at the start, a methodical step-by-step approach can be helpful. Here are suggested steps for your project implementation.

1. Give all datasheets and reference manuals a read to verify your understanding of both devices/components and the reference material available to you. There is no substitute for reading the datasheets to get started. Aside from understanding key aspects of the device required for successful interfacing and operation, reading the datasheets will potentially illuminate extended or extra features of the device which may be advantageous to the designer. While I was able to cover the highlights of each component in class, we did not have sufficient time to cover all of the details. Most questions can be answered by reading the documentation.

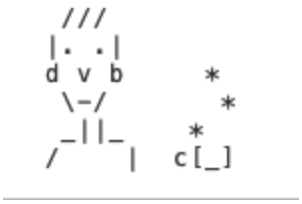


Figure P1.1: Dr Hummel drinking coffee and answering student's questions

2. Utilize accurate schematic diagrams to guide your work. The final schematic for this project should include the STM32L476, the DAC, the keypad, and power connections. Common schematic practices include:

a. Labeling each IC, component, etc. with a reference designator, e.g. U1, R1, C21, etc. For ICs, this label can be either inside the IC or directly above or below the IC.

b. Label each chip/component with the value/part number or full model number for the component, e.g. 10K, .01uf, STM32L476RG, etc.

c. Numbering all pins above the trace outside of the IC. Refer to the IC diagram in the CubeIDE for their number.

d. Labeling the IC pin name, e.g. P1.0, CS, VCC, inside the chip where the wire attaches to the chip.

e. Labeling all wires with a signal name, e.g. A21, D7, CLK, etc.

The process of drawing an initial schematic does not have to be long or involved. It can be a quick effort using pencil and paper in which key information such as pin names and numbers are given to aid in wiring and troubleshooting. A formal schematic can be constructed at a later time once the design is finalized. (Refer to Technical Note 3)

3. **Do not skip designing your program before coding!** Sketch out what the software will do. Aids such as flow charts, state diagrams, pseudo-code, etc. may be used to do this.

4. ***After*** you have your complete program logic designed, you can start the actual code writing. You can start by creating #define constants that will be useful. (For example #define DEFAULT_PIN, CPU_FREQ, LOCKED)

    a. Write your program in modules with separate functions that perform repeated tasks. Many of these were probably done durig the previous assignments, but you may find that a function could be extended with helpful features or a new function could be useful. Verify each function works individually before using it in your project. For example, a keypad functionality can be verified by writing values to a set of LEDs.

    b. After getting all of the individual functions working fully, build the project application as you designed in your flowcharts. If you run into an issue with the logic of your original design, go back and redesign your program with a new flowchart. Do not try to change aspects of your program without considering how it functions as a whole. This is hard to do when only looking at the code!

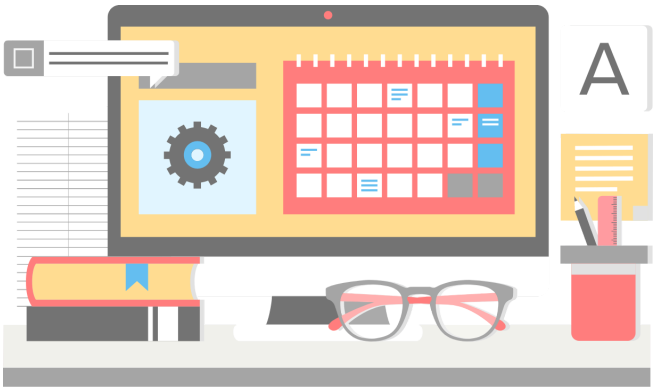
    c. Comment and document your code as you write it. Comments are doing your future self a favor now.

5. Don't forget to make plenty of backups along the way to preserve the good work that has been accomplished. Making frequent backups will allow the developer to roll back to a known development state when something goes wrong or breaks. Although there are various comprehensive version control products available for this purpose (git, svn), a simple approach such as maintaining version directories of a development project on Dropbox, Google Drive, or OneDrive may be satisfactory for projects such as this.

Hints

- Speed up the MCU clock to improve performance and resolution.
- A while loop can be used to continuously read the keypad until a key is detected. (*Pro tip: a while loop can also be used to detect when a button has been released)*
- Measure the time needed to set a single point with a Timer ISR as described and calculate the maximum resolution.
- Determine how many points you want to write to the DAC per period of each of the waveforms. *(Pro Tip: Pick a value that meets spec and is divisible by 5, 4, and 3 if possible)*
- Develop a plan for creating the sine wave function
    - It is tempting to use the C programming language's sin() function, however this function uses floating point numbers and should not be used while "plotting points" or creating the waveforms with the DAC.
    - One common microcontroller "trick" is to use pre-computed values stored in a look-up table or array so that complex computations do not have to be done by the microcontroller. These tables can also be computed once when the device is initialized or the program starts. Please ensure that any use of floating point math, e.g. calculation with a sine function, does not degrade system performance or slow the resulting device in a noticeable way.
- The method used for creating a sine wave can be used to create other the waveforms. While some waveforms may be easier to implement, creating a new method for each waveform separately is typically more work than adapting the same method multiple times.
- The output from the DAC may not be ideal due to non-linearity errors. It may take some extra time to fine tune your function generator to make it meet specifications fully.

P3 - Design Project



Project Description

The design project should be a culmination of embedded systems topics learned and experience gained in this course. This project should utilize a sensor, actuator, or another new peripheral as a central theme to the system you develop. The final system developed needs to be a stand-alone device which provides useful or entertaining functionality. Good examples of stand-alone embedded systems are devices like a digital alarm clock, hand-held medical thermometer, GPS navigation unit, and interactive LED light board. Your design project system may have other interesting components or subsystems besides the sensor, actuator, or key peripheral. You may work in a group of your choosing for this project. The expectations and grading will scale with the group size, so a group of 2 will be expected to do more than an individual working alone.

Rule of thumb: please choose one new sensor, actuator or new peripheral per person.

Selecting a Peripheral

Table P3.1 below provides links to a variety of peripherals which may be suitable for this project. Some are simply sensors and others are packaged devices. Please note that this list is only a suggestion and you are free to find other peripherals. You will purchase all components required for this project. Don't forget to include delivery time into your project schedule.

Company	Website
Sparkfun	https://www.sparkfun.com
DigiKey	https://www.digikey.com
Digilent	https://store.digilentinc.com/pmod-modules/
Adafruit	https://www.adafruit.com

ST Nucleo Expansion Boards	https://www.st.com/en/evaluation-tools/stm32-nucleo-expansion-boards.html

Table P3.1: Peripheral Sources

Peripheral Interface and Calibration

The first step in building a complete system after selecting a peripheral, is to interface to it, and calibrate it with real world data. To show this is done correctly, you should build a simple proof of functionality system. For a sensor, the calibrated data could be monitored and displayed on an LCD or terminal. For an actuator, motion could be controlled with the keypad or terminal. The specific proof of functionality will be dependent upon the peripheral selected. Your choice of peripheral will largely impact what you learn with this project. Most sensors produce an analog output that must be sampled and converted using the ADC. Other sensors interface serially via SPI, I2C, or USART. View Technical Note 5 for guidance on calibrating your peripheral.

It is required to perform some sort of verification or calibration and to report on the steps you took for verification.

~~Battery Power Calculation~~

~~The final report for the completed system must include a power section that shows calculations to predict power use and battery life. Current or power measurements can also be collected and documented in this section. Any optimization of your code to reduce power use or run in lower power states should be documented in this section as well.~~

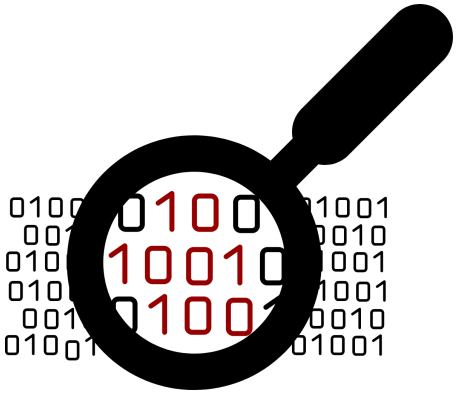
Demonstration

Each person or group will be required to give a short demonstration presentation of the final project. No slide deck is needed for the presentation. The presentation should last no more than 5 minutes and include the following:

1. Description of the project – what does it do
2. Overview of technical details
    a. What peripherals were used
    b. Basic overview of the software architecture
    c. Functionality specifications - battery life, etc.
3. Live demonstration of the device

Technical Notes

TN1 - Hardware Debugger

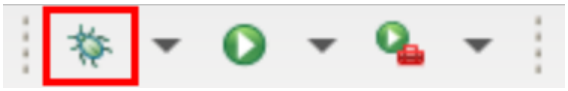


Background

A hardware debugger (HDB) is an essential piece of equipment for microcontroller development. Virtually all modern microcontrollers include the capability to interface with a hardware debugger. Learning how to use the extensive capabilities of the hardware debugger will enable you to test, debug, and verify your code quickly and effectively. This technical note will only cover a small portion of the capabilities of the hardware debugger on the Nucleo-STM32.

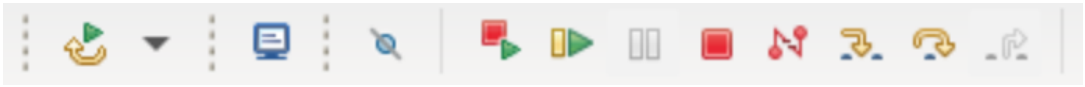
Debug Mode

Debug mode can be used every time programming the development board by clicking on the debug icon. This requires the program to compile and build correctly without errors. If there are any build errors, STM32CubeIDE will show an error message and not enter debug mode.



Run / Step / Pause

The hardware debugger allows you to step through your code, executing each line of code step by step live on the hardware. This is not a simulation, this is the actual program running on the actual hardware.



1. Restart - Will move the program counter to the beginning of the program or restart execution from the first line of code.
2. Add Breakpoint - Allows adding or creating a breakpoint manually

3. Run / Resume - Will run the program after launching debug mode or resume running the program after pausing or encountering a halting breakpoint.
4. Halt / Pause - Will cause a currenting running program to pause after finishing the currently executing instruction
5. Terminate - Will stop the current program execution and exit the current debugging session
6. Step Into - Will execute a single instruction or line of C code. If the instruction is a function call, it will step into the function and pause at the first instruction inside the function allowing the code inside the function to be stepped through line by line.
7. Step Over - Will execute the current instruction. If the instruction is a function call, the function will be executed in its entirety as a single “step” and pause at the next instruction after the function call
8. Step Out Of - Will continue execution of the program until reaching the caller function (returning from the current function being executed)

Variables / Expressions View

During the debugging process the status of all available variables can be watched. This allows the programmer to watch what the program is doing at each line of code. There are 2 tabs that can be used to monitor the status of variables in debug mode, the variables and expressions tabs. Not only can the values be monitored as each instruction is executed, but their values can also be manually changed. The values of any variable can be changed every time the debugger halts. The updated value will be used starting with the next instruction that is executed. If the variables or expressions tab is not visible in debug mode, they can be added using the view menu. The entire window can be reset using Window ➔ Perspective ➔ Reset Perspective.

Variables Tab
The variables tab will automatically populate a list of all variables that are active in memory. The list of variables in this view cannot be edited. *The values in the full list are only updated when the debugger halts.*

Expressions
The expressions list in the Debugger view can be populated with any variable in the program. When adding a variable to the list, STM32CubeIDE will help by providing autocomplete suggestions. The predefined memory structures for STM32 peripherals with references to the registers can also be added to the expressions view. Beyond just memory values, any arithmetic or logic expression can be entered. *Like the variable list, all values in the expression view are only updated when the debugger halts.*

Breakpoints

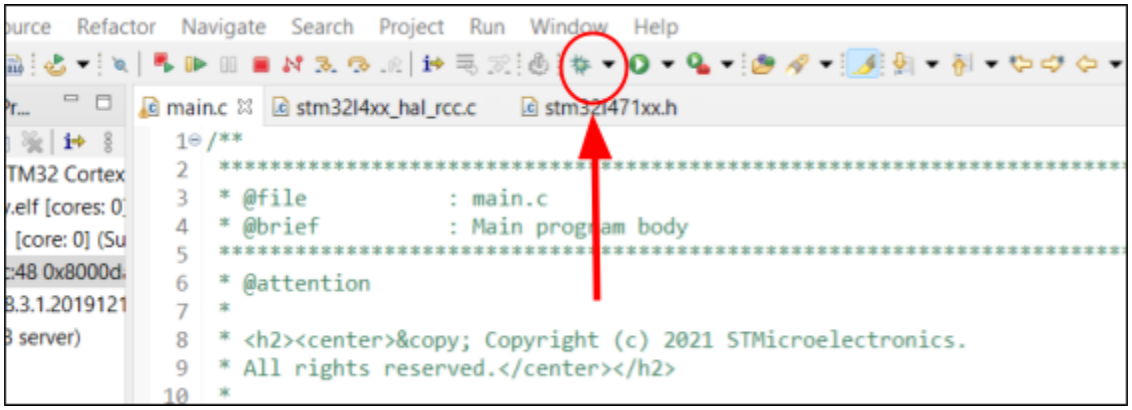
Breakpoints are markers defined at a specific instruction. Their default behavior is to cause the debugger to halt whenever they are reached. This allows the programmer to run code without stepping through it line by line and still halting at a specific section or instruction. When the Run or Resume button is used, if the program reaches a breakpoint during execution, it will halt before executing the instruction where the breakpoint was defined. Breakpoints can be added by double clicking in the empty column space to the left of the line number in the code editor. A blue circle will appear signifying a breakpoint was added on that line. Breakpoints can perform more advanced options besides simply halting execution. They are an incredibly powerful tool and critical to debugging code effectively and efficiently.

Using printf()*

In C, printf() sends data to be printed to a default stream STDOUT. On a MCU like the STM32, there is no default output for printf to send the printed text to. The hardware debugger can provide a connection for printf() and a window to view the output.

Steps to configure the debugger and enable printf()

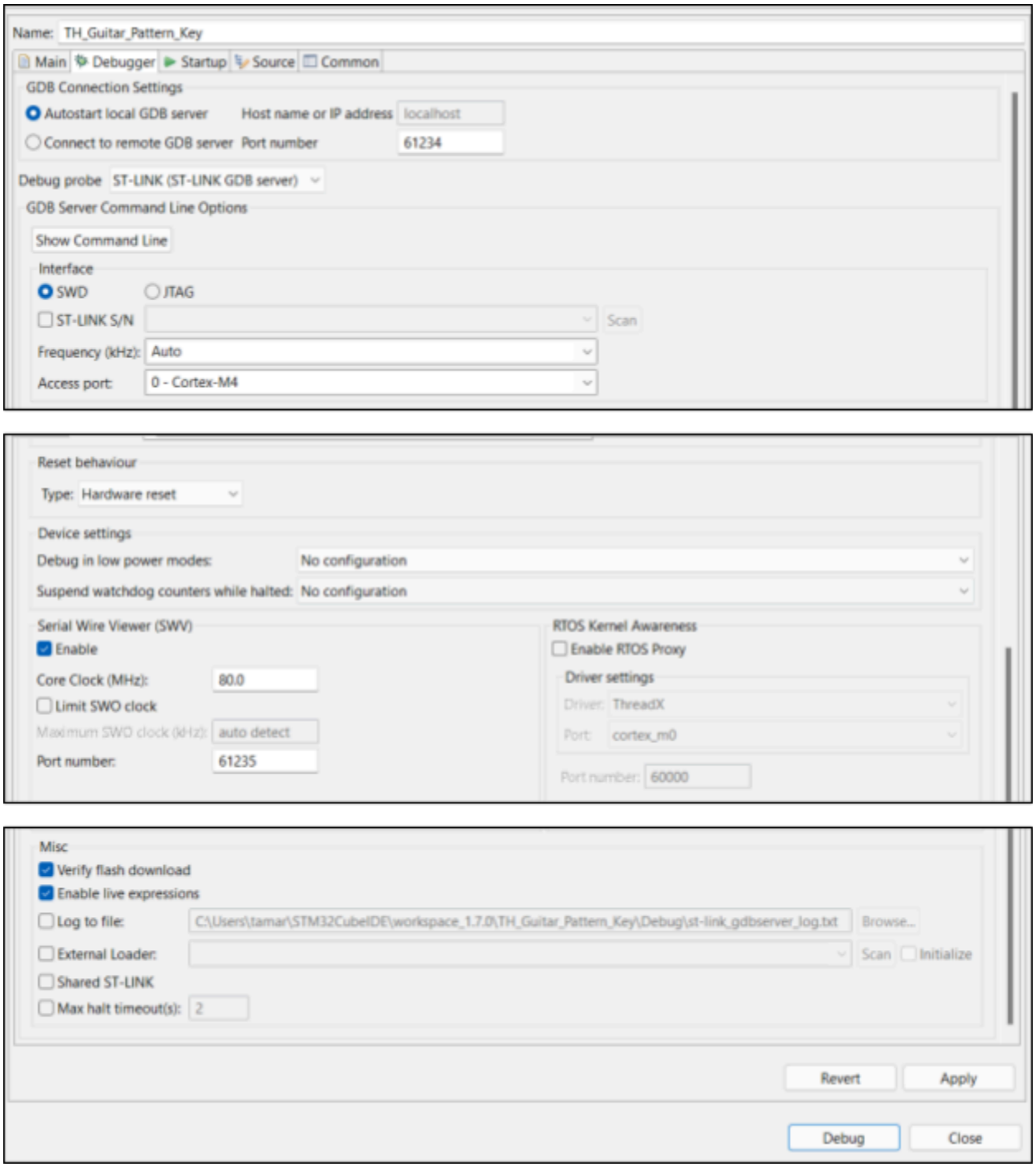
1. Select the drop down menu on the debugger icon



2. Select the “Debugger Configurations” option and select the debugger tab. Ensure that the fields below are set as follows:

   Debug Probe: ST-LINK (ST-LINK GDB server)
   Interface: SWD
   Serial Wire Viewer (SWV): Enable
   Core Clock (MHz): Set To Speed Of HCLK
   Enable Live Expressions: Check Enable

   Select apply and now the debugger is configured properly



3. Include _write Function

   A _write function must be defined to replace the typical standard library functions that printf() uses. Add the following function in main.c

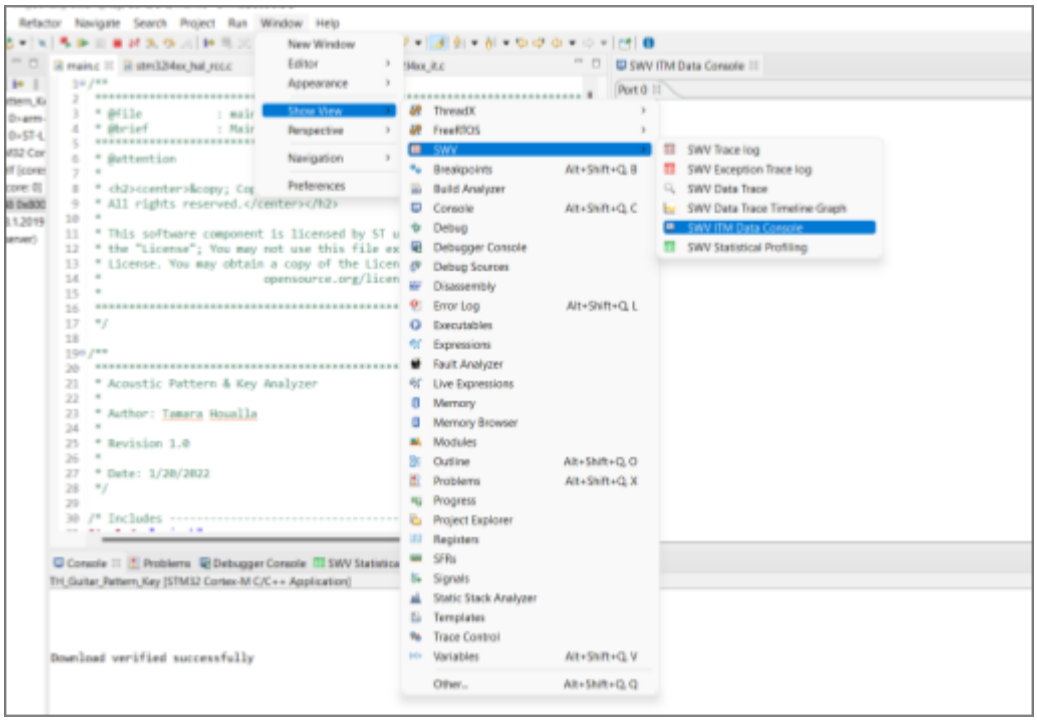
```
int _write(int file, char *ptr, int len){
   for(int i=0; i<len; i++)
      ITM_SendChar((*ptr++));
   return len;
```

4. Set Up SWV ITM Data Console

   The SWV ITM Data Console is the window where printf() will send its output to for viewing.

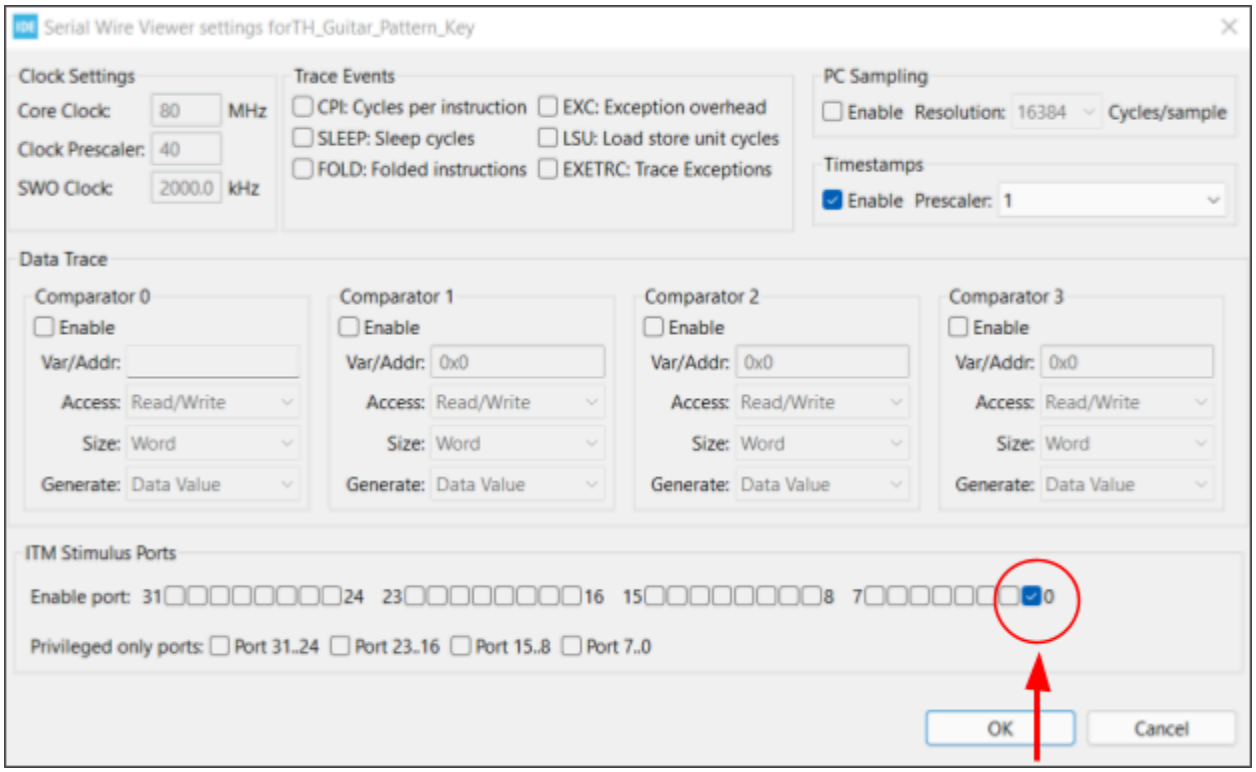
   To access that window start the debugger by clicking the debug icon (After the debugger was configured properly in earlier steps)

   On the top menu bar select Window→Show View→SWV→SWV ITM Data Console



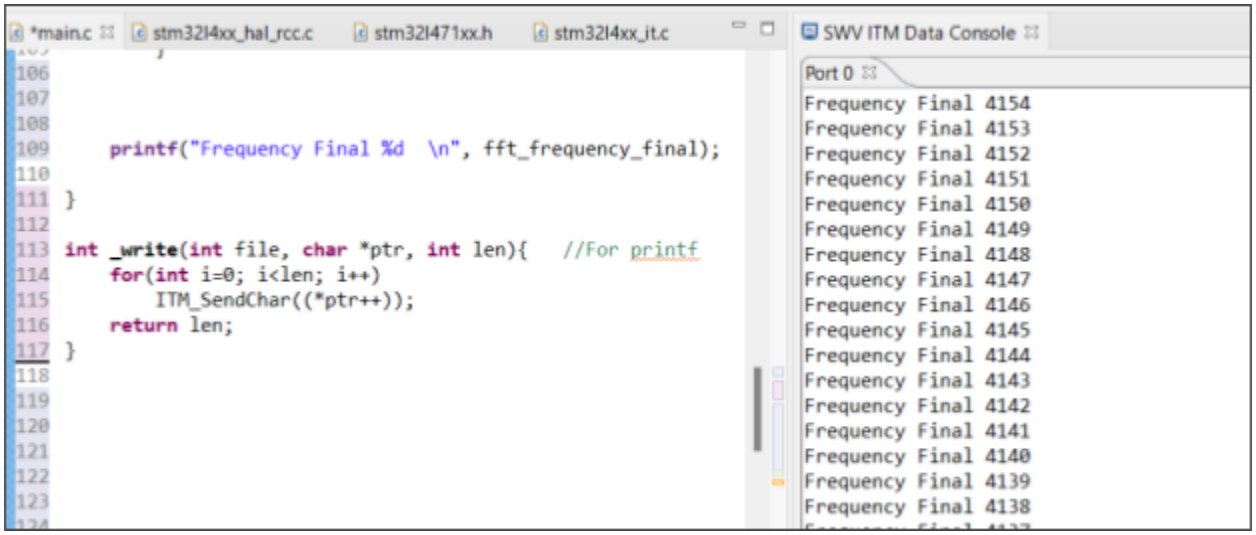
   When the window appears select the configure trace button shown below

Enable **Port 0** to view data from the printf() function in the serial window



Select the start trace button shown below

Press the play icon to run the code and view the output on the console
Note: may need to pause then run code again before output is seen



Live Expressions*

Live expressions is a separate tab in the debugger that can watch variables in “real time” as the program is running on the MCU. This can allow you to watch the data in variables without having to pause execution. There are a few caveats with Live Expressions:

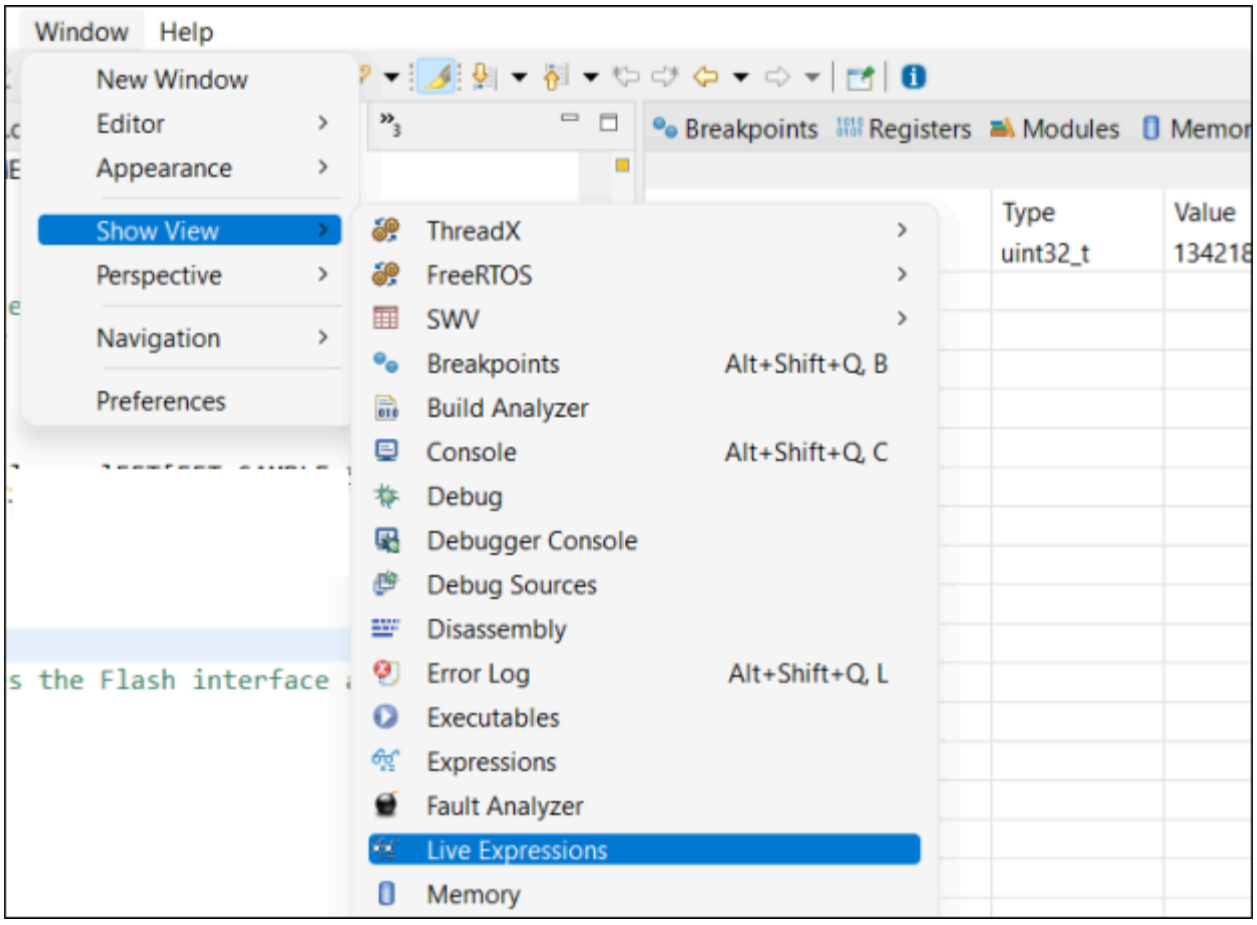
- Watching variables will have some impact on the run time of the program on the MCU
- There is some lag as the data of variable changes is sent from the MCU to the debugger
- Only global variables can be watched with Live Expressions

If you want to view non-global variables in “real time”, you can use the SWV Trace or Data Trace features of the debugger. Instructions for setting up SWV Trace can be found at https://www.youtube.com/watch?v=Eg_GLvLHM1o

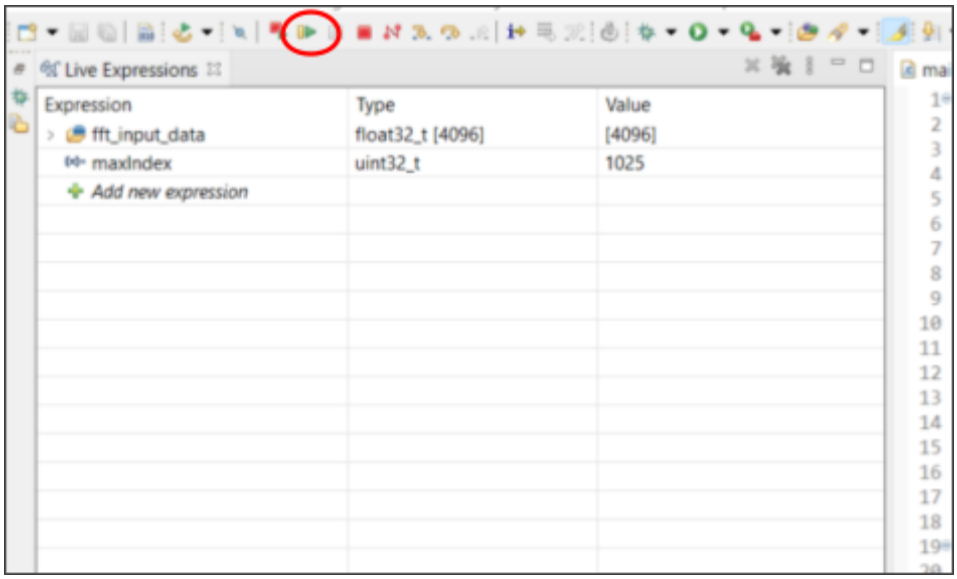
Steps to configure the debugger for Live Expressions

1. Follow the steps 1 and 2 for configuring the debugger for printf() above.
2. Open the Live Expressions tab

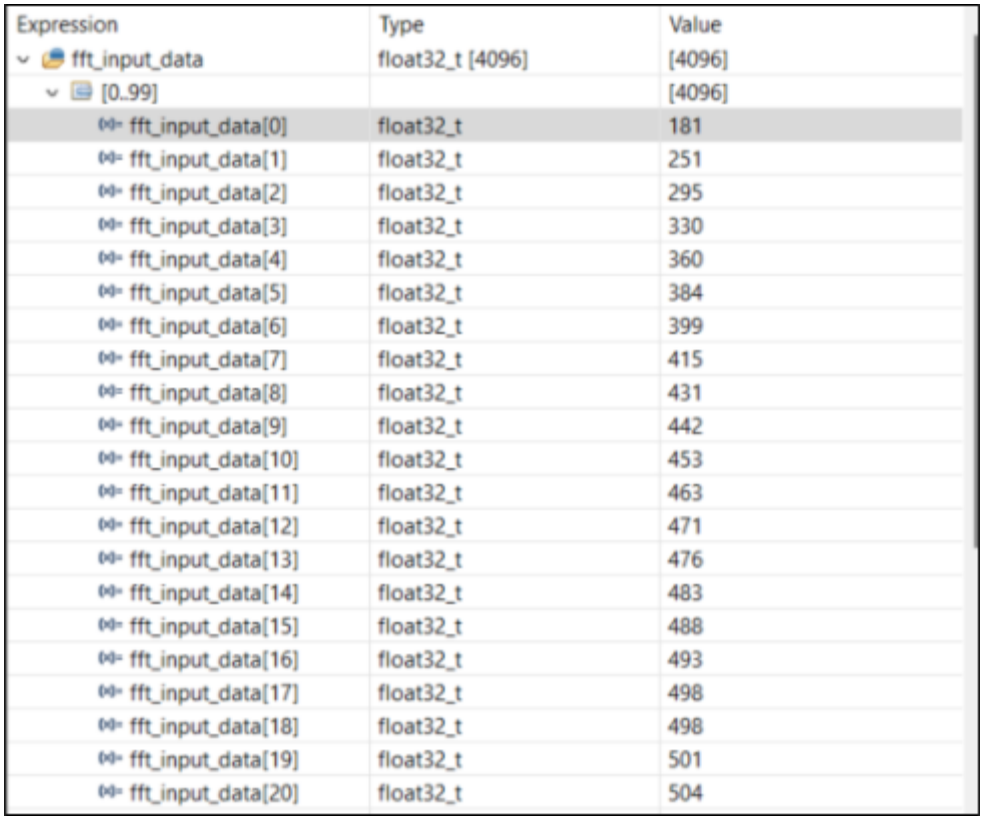
   Enter Debug mode and go to Window→Show View→Live Expressions



Enter in the variables you want to observe, and press run



Note: Arrays can be expanded and each value in the array can be observed in real time



Expression	Type	Value
fft_input_data	float32_t [4096]	[4096]
[0..99]		[4096]
fft_input_data[0]	float32_t	181
fft_input_data[1]	float32_t	251
fft_input_data[2]	float32_t	295
fft_input_data[3]	float32_t	330
fft_input_data[4]	float32_t	360
fft_input_data[5]	float32_t	384
fft_input_data[6]	float32_t	399
fft_input_data[7]	float32_t	415
fft_input_data[8]	float32_t	431
fft_input_data[9]	float32_t	442
fft_input_data[10]	float32_t	453
fft_input_data[11]	float32_t	463
fft_input_data[12]	float32_t	471
fft_input_data[13]	float32_t	476
fft_input_data[14]	float32_t	483
fft_input_data[15]	float32_t	488
fft_input_data[16]	float32_t	493
fft_input_data[17]	float32_t	498
fft_input_data[18]	float32_t	498
fft_input_data[19]	float32_t	501
fft_input_data[20]	float32_t	504

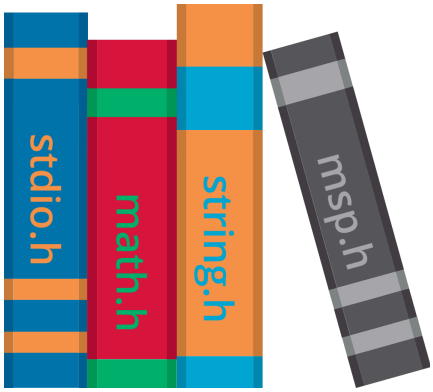
Advanced Features

The hardware debugger is capable of other advanced features that can make debugging code easier or more efficient. ST has created a set of videos covering some of them.

https://www.youtube.com/playlist?list=PLnMKNibPkDnEDEsV7IBXNvg7oNn3MfRd6

*** This section was provided by Tamara Houalla**

TN2 - Create Custom Libraries with #include Header files



Background

A header file may be used to collect function declarations within a specified file available to the C compiler, allowing specific functions to be available for reuse at a later time. Consolidating functions in this fashion is beneficial from a code reuse standpoint; code written for one module can be reused within other modules. Additional benefits are the reduction of clutter within C modules and reduced software maintenance due to the fact that code can be written and tested once, and then used repeatedly in other functions.

Example Implementation

When writing code for a specific device or peripheral, all of the functions that would typically be written to utilize it can be grouped together in a library of code (.c) and definitions and declarations (.h). For example, if working with a GPS sensor, functions to use the device could include the following: GPS_config, GPS_read_status, and GPS_read_location. These functions can be separated into a set of files that can be easily included in future projects to reuse the code and easily implement the same GPS module in a new project.

Header file

Create a header file to include all defined constants and function declarations. This allows for easy editing of constants. An example for the GPS sensor is shown below.

GPS.h

```
#define GPS_FREQ 120000
#define GPS_POCI_PIN 4
#define GPS_PICO_PIN 7

uint8_t  GPS_config (uint32_t GPS_options);
uint32_t GPS_read_status (void);
uint32_t GPS_read_location (void);
```

Issues can arise if the same set of functions is included multiple times in a single project. For example, if main.c includes GPS.h and also includes another library file that itself includes GPS.h the

compiler assumes the second or later includes are conflicting definitions of the same function. To avoid this issue, libraries can tell the compiler to only include them if they have not already been included. This is done with some precompiler directives. Each header file encloses inside a block of `#ifndef … #endif`. The `#ifndef header_text` tests if `header_text` has already been defined. If the `header_text` has been defined, the compiler skips to the `#endif`. These precompiler directives are automatically added when creating a header file with Code Composer Studio. (Using New ➔ Header File) The updated GPS.h below shows the typical template provided by Code Composer Studio

GPS.h

```
#ifndef GPS_H_
#define GPS_H_

#define GPS_FREQ  120000
#define GPS_POCI_PIN 4
#define GPS_PICO_PIN 7

uint8_t  GPS_Config (uint32_t GPS_options);
uint32_t GPS_ReadStatus (void);
uint32_t GPS_ReadLocation (void);

#endif /* GPS_h_ */
```

Code file

Create a matching .c source code file (GPS.c in this case) that includes the code for all of the functions along with the required memory structures or variables. An example for the GPS sensor is shown below.

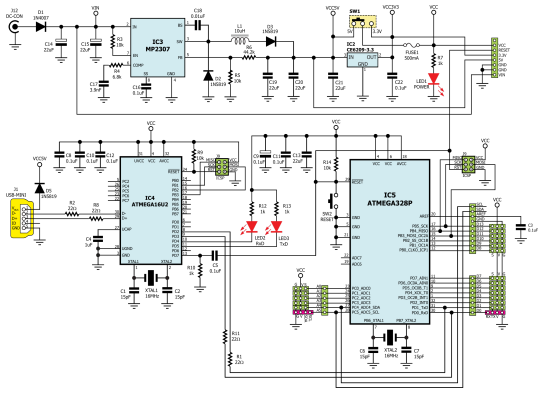
GPS.c

```
uint8_t GPS_config(uint32_t GPS_options) {
      // code implementing configuration of GPS module
}

uint32_t GPS_read_status(void) {
      // code to read GPS status
}

uint32_t GPS_read_location(void) {
      // code to read GPS location
}
```

Adding the library to a project

Both of the created GPS.h and GPS.c files can be added to any project folder and then referenced inside the main.c file by adding `#include “GPS.h”`. This will tell the compiler where to find the functions when called. Any of the defined constants or functions can be called and used from the main program.

TN3 - Schematic Best Practices



Background

Creating schematics that are clear and follow standard formatting practices is necessary for all technical documentation of electronic devices. Schematics are critical for diagnosing and repairing malfunctioning electronics, so providing all of the necessary information in a clear and concise format is important.

Component Labels

All components in the schematic require a unique identifying label, typically in the form of a letter and number combination. The letter is used to identify what type of component is being referenced. Common identifiers include:

- R – Resistor
- C – Capacitor
- L – Inductor
- D – Diode
- Q – Transistor
- U / IC – Integrated Circuit
- Y – Crystal
- S – Switch
- J / JP – Junction, Connector, Jumper Pin
- T – Transformer
- K – Relay
- M – Motor
- F – Fuse

The letter identifier is followed by a number with each component of the same type incrementing the value. The numbering will typically start at one corner of the finished schematic and increment across the schematic. This creates that result of numerical values reflecting proximity, so R2 should be close to R1 and R3, and farther away from R20. The numbering between different types of components do not necessarily relate proximity, so R20 does not have to be close to C20.

Passive components should also be labeled with their component size or value using appropriate units. For example 10 kΩ rather than 10000 Ω. Units should use appropriate symbols and prefixes. Use 10 uΩ instead of 10 micro-ohms. This value label should be located above or below the component identifier.

IC and Chip Labeling

ICs should include their specific model number with the IC identifier. The outline of the IC should contain lines for every connected pin. Smaller ICs will typically have the pin locations correspond to their physical location on the chip. As the number of pins increases, this can create messy and hard to follow wire traces. To make the schematic readable, the pin locations can be moved to different sides on the schematic. Schematics can connect pins to all 4 sides of the IC or just 2 (left and right). A schematic should let the reader's eye follow the flow of signals when possible. Pin locations should be equally spaced apart on the schematic IC.

Each connected pin of the IC requires 2 identifying labels. Inside the IC the signal or port name connected to the pin should be labeled like P1.0, VCC, SCLK, or PICO. This label should be centered where the pin connects to the chip. Immediately outside the IC, the pin number should be printed above the pin wire. The pin number corresponds to the pin location based on the specific package of the IC used in building the circuit. Refer to the example in Figure TN5.1 below. *In this example, color is used solely for distinguishing the different labeling components.*

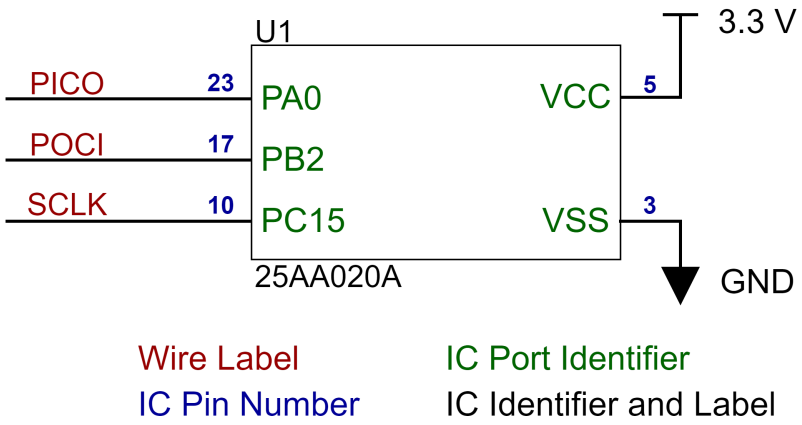


Figure TN5.1: Example IC Schematic

Drawing and Labeling Wires

Wires should be spaced uniformly on a schematic. Groups of wires can be joined to form a bus for ease of reading, but the individual wires must be labeled to show how their order in the bus makeup. This is necessary to allow individual wires to be traced through the bus. While most wires require no label, some signal names can be labeled above their specific wires to help in ease of reading the schematic. These labels are required when wires connect off the current page to another schematic.

When drawing wires, crossing wires in a schematic do not signify they are connected. Jumping wires to show they are unconnected is valid, but not used in professional schematics. The jumps make the wiring appear busy and harder to follow. To show that wires are connected, a dot should be added at the connection point. For T junctions of 3 wires connecting, adding a dot is valid but unnecessary. See Figure TN5.2 below for reference.

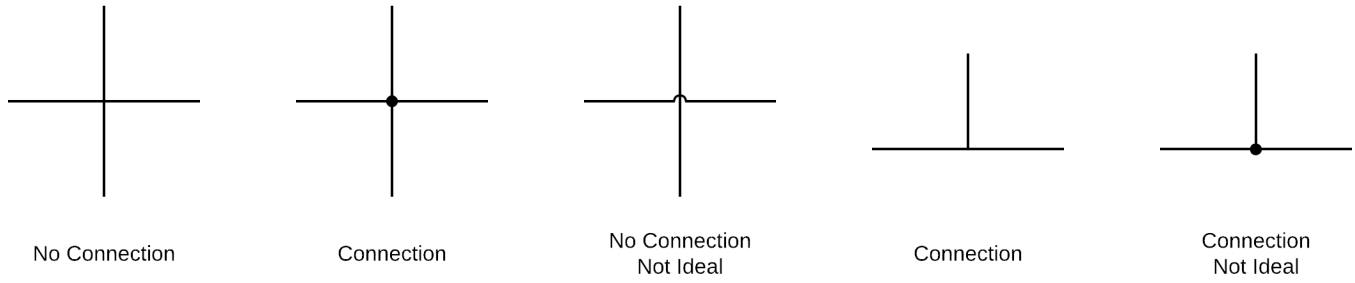


Figure TN5.2: Example Wire Connections

General Layout Considerations

- Power connections should generally point upward on the sheet while ground connections should point downward
- IC shapes should have a consistent size. While a 14 pin chip should be larger than an 8 pin chip, all 14 pin chips should have the same size rectangle.
- Be mindful of color use. All schematics must also be understandable in black and white.

Example Schematic

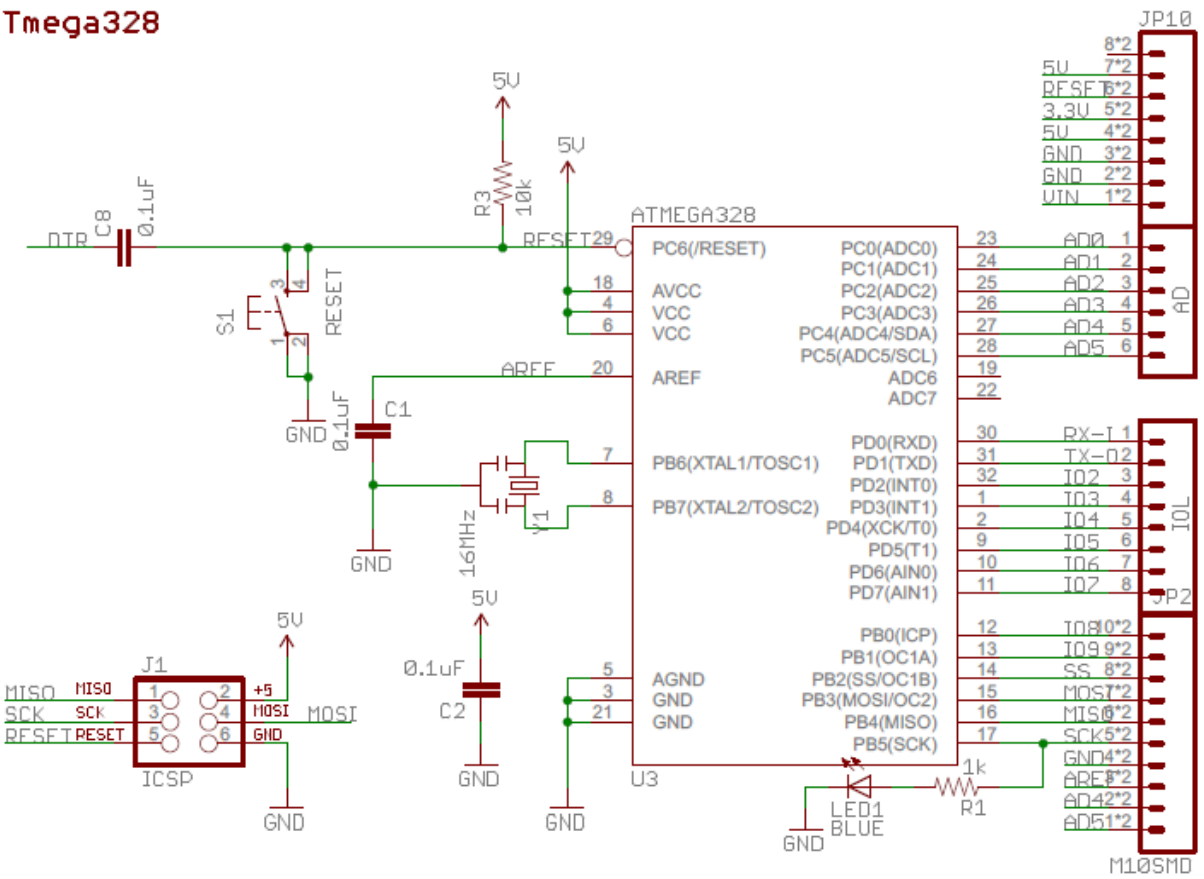


Figure TN5.3: Example Schematic (Source: sparkfun link below)

Additional Resources

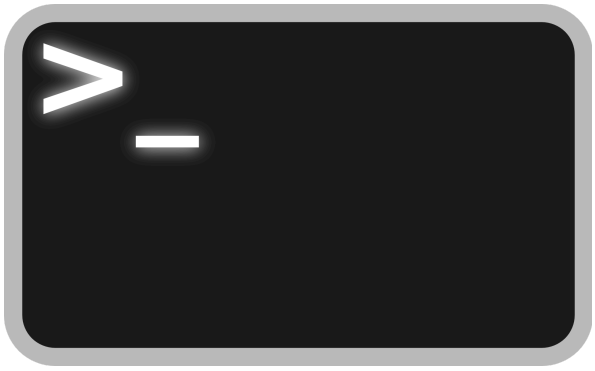
https://learn.sparkfun.com/tutorials/how-to-read-a-schematic

https://electronics.stackexchange.com/questions/28251/rules-and-guidelines-for-drawing-good-schematics

Software Tools for Creating Schematics

- LTSpice LTspice | Design Center
- CircuitLab https://www.circuitlab.com
- Diagrams.net https://www.diagrams.net (Formerly Draw.io)
- Lucidchart https://www.lucidchart.com
- Microsoft Visio https://products.office.com/en-us/visio/flowchart-software

TN4 - VT100 Serial Terminal



Background

Serial terminals have a long history in computers. Serial terminals originally started out as the primary method of gaining access to a time-shared computer system. Desktop computers as we know today didn't exist. Instead, desktops were just shells with a keyboard and mouse that connected to a local server, a room sized computer elsewhere in the building. These shells were called terminals. A user would sit down at a terminal, login at the provided prompt, and perform all work at the command line. Although this sounds fairly primitive, it was a basis for what we call computer networks today. It was also a big leap in expanding the use of computers beyond the academic and research domain.

The VT100 video terminal was the first terminal to provide ANSI escape codes for doing more than just printing text. Codes exist for controlling the screen like moving the cursor or clearing the screen. Codes were also created to add formatting to text like bold, underline, and blinking text. (Don't laugh, blinking text was high tech!) The VT100 was introduced in 1978 by Digital Equipment Corporation (DEC) and helped them become a leader in computer terminals. The codes introduced by the VT100 quickly became a standard used by others. There have been many compatible terminals used since the original VT100 but the VT100 is somewhat synonymous with a terminal that interprets escape codes for a variety of cursor and keyboard functions.

Even though networked computer systems today have outgrown the simple abilities of the VT100 terminal, the codes have continued to find uses. Serial interfaces like RS-232 are still common in embedded systems and electronic equipment for debug and diagnostic interfaces. This is largely due to the ease of implementation and previously widespread adoption. Even into the early 2000s desktop computers came with at least 1 COM port that utilized RS-232. To connect to these diagnostic interfaces, software terminals were made to replicate the functionality of terminals like the VT100. Using these software terminals allow us to take advantage of the same high tech escape codes. Below is a list of available software terminals that support VT100 escape codes

Terminal Emulator Applications

Windows

- Real Term https://realterm.i2cchip.com/

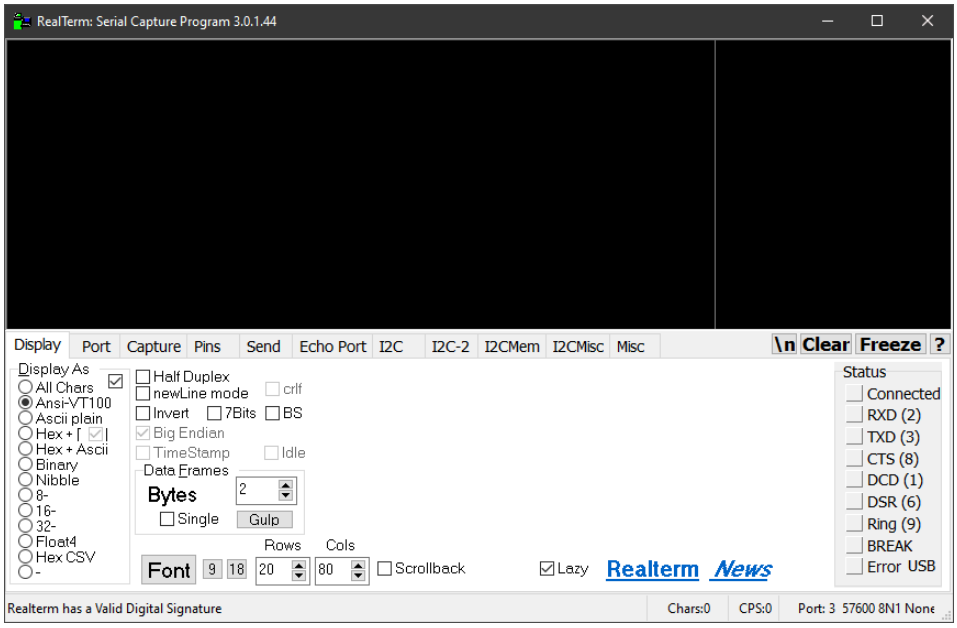


Figure TN4.1: RealTerm VT100 supported with option Ansi-VT100

OSX

The authors have not been able to verify any OSX applications, but the built-in terminal in Mac OSX should include the terminal app screen.

1. Find the right TTY device, in the terminal type ls /dev/cu.*
   With the NUCLEO development board plugged in, you will get a list

```
$ ls /dev/cu.*
/dev/cu.Bluetooth-modem          /dev/cu.iPhone-WirelessiAP
/dev/cu.Bluetooth-PDA-Sync       /dev/cu.usbserial
```

2. To run screen to connect to the usbserial device (just an example) with a baud rate of 9600, 8 bits data, no parity, and 1 stop bit type: **screen /dev/cu.usbserial 9600**

3. To quit screen, type CTRL+A, then CTRL+\

More information on using screen can be found in the manual pages for screen. They can be read by typing **`man screen`** in the terminal app. Use ‘enter’ or ‘space’ to scroll the pages and ‘q’ to quit.

Linux

Linux distributions come with a variety of terminal shells that natively support ANSI escape codes. Below is a list of command line tools that can connect the terminal with an RS-232 serial interface.

- screen

```
screen /dev/<device_port> <baud rate>
CTRL+A,?            // bring up help menu
CTRL+A,k            // disconnect and exit
```

- minicom

```
minicom -s          // setup port configuration
CTRL+A, Z           // bring up help menu
```

- picocom

```
picocom --b <baud rate> /dev/<device_port>
CTRL+A, CTRL+X      // disconnect and exit
```

Escape Codes

As described above, escape codes (or sequences) are simply use of specific sequences of ASCII characters to instruct the terminal to do things like set the cursor to a specific location, blink text, etc. The general process is to use an escape code to place the cursor at a given location and then write the desired text that will show up at that location.

The example below shows a sequence of characters that includes escape codes. If each of these characters was sent in order to a supporting VT100 terminal, the codes would be interpreted to move the cursor down 3 lines, then move the cursor over 5 spaces to the right, and then print “Hello World”. Note the ESC is a single character with a hex value of 0x1B. Also that the numbers are sent as characters, not numerical values. So 3 below is the character ‘3’ not the numeric value 3

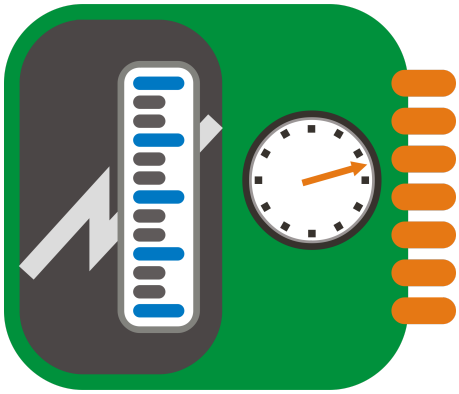
```
“ESC[3B”
“ESC[5C”
“Hello World”
```

A table of common escape codes is given below. A full listing can be found at: https://www2.ccs.neu.edu/research/gpc/VonaUtils/vona/terminal/vtansi.htm (Note that not all escape codes are supported by terminal applications or emulators)

Escape Code	Function
ESC[**Value**A	Move cursor up Value lines
ESC[**Value**B	Move cursor down Value lines
ESC[**Value**C	Move cursor right Value spaces
ESC[**Value**D	Move cursor left Value spaces
ESC[H	Move cursor to upper left corner
ESC[**Line**;**Column**H	Move cursor to screen location Line, Column
ESC7	Save the cursor position and attributes
ESC8	Restore the cursor position and attributes
ESC[0K	Clear the line from the cursor to the right
ESC[1K	Clear the line from the cursor to the left
ESC[2K	Clear the entire line
ESC[0J	Clear the screen from the cursor down
ESC[1J	Clear the screen from the cursor up
ESC[2J	Clear the entire screen
ESC[0m	Turn off character attributes
ESC[1m	Turn on bold mode
ESC[2m	Turn on low intensity mode (non-bold)
ESC[4m	Turn on underline mode
ESC[5m	Turn on blinking mode
ESC[3**Value**m	Change text color (Value is RGB color 0 to 7)
ESC[4**Value**m	Change text background color (Value is RGB color 0 to 7)

Table TN4.1: VT100 Escape Codes

TN5 - Calibrating ADC / Sensor



Background

The analog input on the STM32L4 will ideally follow the formula

$$N_{ADC} = 2^{ADC\_BITS} \times \frac{V_{in} - V_{R-}}{V_{R+} - V_{R-}}$$

The resolution of your input, the value of 1 LSB (least significant bit) is

$$1\ LSB = \frac{V_{R+} - V_{R-}}{2^{ADC\_BITS}}$$

Unfortunately most analog inputs will not perfectly match the ideal formulas above and will need a further calibration formula to get an accurate value from the ADC value. The simplest calibration is to provide a linear approximation across the range of wanted values. If your application only needs to record voltages between 1.2 V and 1.8 V then being calibrated to accurately measure 2.5 V is unimportant, so the calibration should be focused on the desired range of 1.2 to 1.8. Calibration requires you to have a reliable method to measure whatever the sensor is measuring, and the calibration can only be as accurate as the measurement equipment it is being calibrated with. Depending on the sensor, this can sometimes be the most difficult aspect of creating a calibration.

Some sensors will provide a more robust calibration formula to map the digital reading to accurate values within an acceptable error tolerance without need for outside verification. When no reliable equipment can be found for corroborating accurate measurements against, this may be an acceptable alternative. Whenever possible, all sensor data should be verified against known reliable equipment. Many sensors will have a linear relationship between what it is sensing or measuring and its output signal. Sensors that do not follow a linear relationship will typically detail a more appropriate curve fitting equation in their documentation. The type of curve will affect how many data points are needed for creating an accurate calibration.

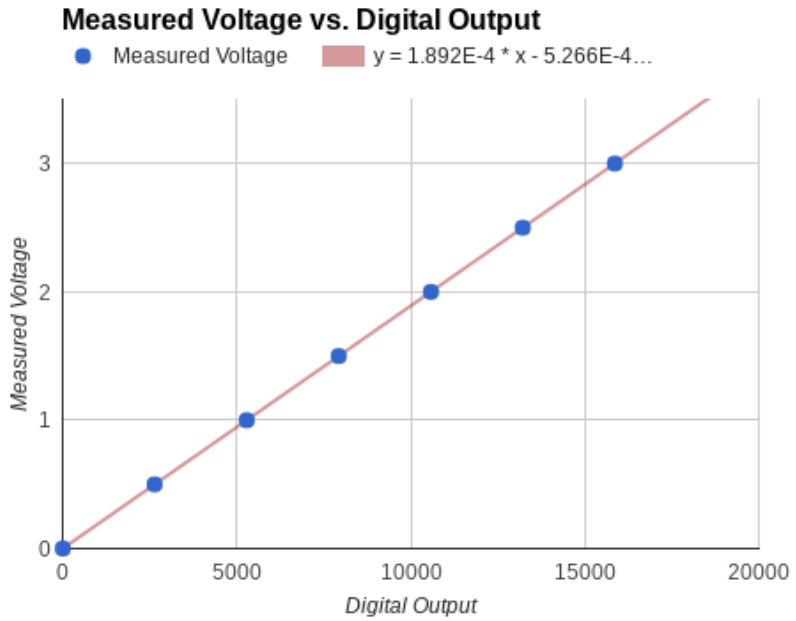
Linear Approximation Calibration Methodology

Sensors that follow a linear shape can be easily calibrated by taking a few measurements and creating a linear regression.

$$Measurement = Digital\_Value \times m + b$$

The first step is to take recordings of several known points over the range of interest. As way of example, the below data points were used in calibrating the ADC14 with a Vref of 3.1 V.

Digital Output	Measured Voltage (V)
6	0
2642	0.5
5288	1
7930	1.5
10579	2
13215	2.5
15861	3



The linear approximation creates a calibration equation for transforming the digital output from the ADC to measured voltage values.

$$Measurement = 1.892 \times 10^{-4} \times Digital\_Value - 5.266 \times 10^{-4}$$

Digital Output	Measured Voltage (V)	Calibrated Value (V)
6	0	0.000609
2642	0.5	0.499340
5288	1	0.999963
7930	1.5	1.499829
10579	2	2.001020
13215	2.5	2.499751
15861	3	3.000375

Linear Approximation without Floats

The above equation unfortunately creates a need for floating point variables. This need for floats can be removed if the value is scaled by multiple orders of magnitude, essentially changing the magnitude of the units. For the above data, the measurement could be saved as μV rather than volts. The new calibration equation becomes

$$Measurement = 189 \times Digital\_Value - 527$$

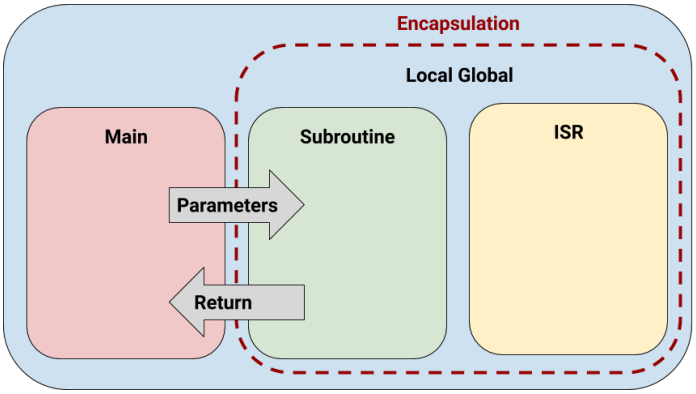
Digital Output	Measured Voltage (V)	Calibrated Value (μV)
6	0	607
2642	0.5	498811
5288	1	998905
7930	1.5	1498243
10579	2	1998904
13215	2.5	2497108
15861	3	2997202

When scaling the measurements be sure to not scale to values beyond the size of your variable. If only positive values will be measured, unsigned variables will double the available size.

Nonlinear Sensor Response

Some sensors may have a nonlinear response. This can be especially pronounced at the extremes of the measurement range. This can be corrected for by having multiple calibration equations that get applied for different ranges of output values. Ultimately it is up to the engineer to verify the sensor device is accurate for the full range specified.

TN6 - ISR Communication and Variable Encapsulation



Background

Interrupt service routines (ISR) pose a unique challenge for safely passing data into and out of. Because ISRs are not called, data cannot be passed via parameters or returned on exiting. An ISR can also interrupt at a critical time when processing or copying data. This is more probable when using data sizes larger than the base register size of the processor. Each of these issues are addressed individually below.

Variable Scope and Interrupts

Scope in a computer program refers to the visibility of variables. Variables defined inside of main can't be seen (or accessed) inside of interrupt service routines and vice versa. Given this, the main loop of a program and ISRs which it may interact with require a method to pass data between with each other. Global variables are an easy way to accomplish this. Here is a quick example of how a global variable may be used to let main know that an interrupt has occurred.

```
static uint32_t interrupt_flag = 0; // an 8-byte flag!

void main(void) {
    while(1){
        if (interrupt_flag)
            // do something
        else
            // do something else
     }
}

void ISR(void) {
    interrupt_flag = 1;
    return
}
```

Encapsulation

Encapsulation may be used to hide (or protect) variables within specific functions which are used to check, set, and clear variables. Encapsulation effectively means creating a driver for a given peripheral that handles all of the details associated with communicating data two and from the peripheral. Here are steps to implement encapsulation in C.

1. Create a separate C file for the driver. Define the variables/flags that you want to use as static globals within this C file. Note that defining these variables as static makes them visible only within this file.
2. Write access functions such as check_flag(), clear_flag(), set_flag(), set_data(), get_data() to allow access to this data.
3. Create a .h file with headers for all of the access functions.
4. Include the .h file in your main C file.
5. Use the access functions both in main and in your ISR anywhere you want to check, use, or change any of your protected variables

`main.c`

```c
#include "mydriver.h"

void main(void) {
    uint32_t localValue;
    while(1){
        if (check_flag()) {
            // something important happened!
            localValue = get_data();
        }
        // nothing happened yet
    }
}
```

`mydriver.h`

```c
#ifndef MYDRIVER_H_
#define MYDRIVER_H_

uint32_t check_flag(void);
uint32_t get_data(void);
void set_data(uint32_t data);
void driver_ISR(void);

#endif /* MYDRIVER_h_ */
```

mydriver.c

```c
#include "mydriver.h"

static uint32_t driverValue = 0;
static uint32_t driverFlag = 0;

uint32_t check_flag(void) {
    return driverFlag;
}

uint32_t get_data(void) {
    return driverValue;
}

void set_data(uint32_t data) {
    driverValue = data;
}

void driver_ISR(void) { // interrupt on peripheral device
    ...
    driverValue = device_register; // read data from device
    driverFlag = 1; // set flag value
    ...
}
```

Data Protection

Consider if a variable being written to in an ISR or read in another part of the program is large enough so that multiple assembly instructions are required to modify it, making it not an atomic data element. Such elements can be corrupted if the read or write of them is interrupted and the element modified prior completion of a read or write. Such actions could result in one half of the element being from the previous value and the other half of the element from a more recent value. Think of a sentence interrupted half way through with one half of the sentence arising from one part of a conversation and the other half from a bit later in the conversation. The end result is a corrupt and unintelligible sentence.

Non-atomic data elements can be protected by disabling interrupts prior to accessing the element and then re-enabled at the completion of reading or writing the element. Here is the example from above with this sort of protection added.

```
static uint32_t interrupt_flag = 0; // an 8-byte flag!

void main(void) {
    __enable_irq();
    while(1){
        if (interrupt_flag)
            // do something
        else
            // do something else
    }
}

void ISR(void) {
    __disable_irq();
    interrupt_flag = 1;
    __enable_irq();
    return
}
```

TN7 - Clock System

Background

The clock system for the STM32L4 can be configured using the Clock Configuration tab in the CubeIDE after creating a new project. This graphical interface can auto calculate settings after manually changing the individual clock signal frequencies.

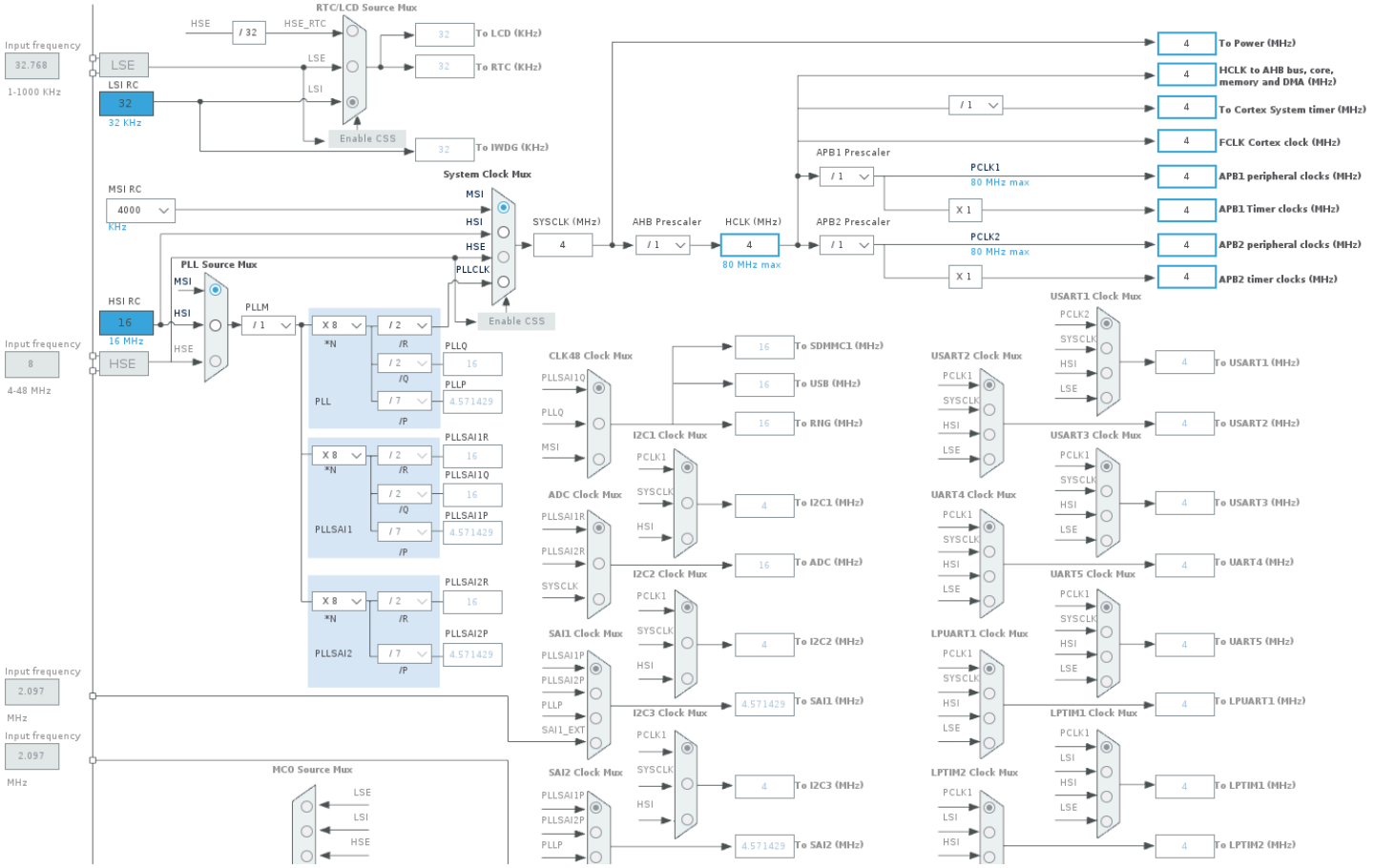


Figure TN7.1 STM32L4 Graphical Clock Configuration

After the auto configuration completes successfully, values for all of the enabled clock signals will update. Saving the configuration will trigger a popup asking to auto generate code for the clock configuration. Auto generating code will create a new main.c with a SystemClock_Config function that will configure the clock system with the selected values.

⚠ **A new blank main.c will be created anytime the automatic code generation process is run. Any code that was previously added or edited in main.c will be deleted.**

TN8 - Nucleo Development Board Unusable Pins

Background

Some pins on STM32 microcontroller are utilized by hardware on the Nucleo development board and are not available for use because they are not connected to the pin headers without adding or removing solder bridges. Avoid using these pins or changing their default configuration. The list of pins are listed in the table below.

Nucleo-L476RG

Port	Pin	Default Use
A	PA2	USART2 TX (ST-LINK)
A	PA3	USART2 RX (ST-LINK)
A	PA13	TMS (ST-LINK)
A	PA14	TCK (ST-LINK)
B	PB3	SWO (ST-LINK)
C	PC13	Onboard Push Button
C	PC14	LSE(+) Onboard oscillator (32.768 kHz)
C	PC15	LSE(-) Onboard oscillator (32.768 kHz)
H	PH0	HSE(+) ST-LINK MCO (8 MHz)
H	PH1	HSE(-) ST-LINK MCO (8 MHz)

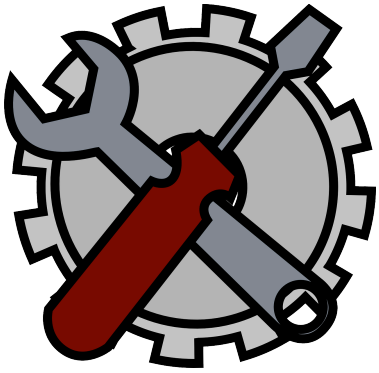
Table TN8.1 Pin table for Nucleo-L476RG

Nucleo-L496ZG / Nucleo-L4A6ZG

Port	Pin	Default Use
G	PG7	LPUART1 TX (ST-LINK)
G	PG8	LPUART2 RX (ST-LINK)
A	PA11	USB OTG (D+)
A	PA12	USB OTG (D-)
A	PA13	TMS (ST-LINK)
A	PA14	TCK (ST-LINK)
B	PB3	SWO (ST-LINK)
C	PC13	Onboard Push Button
C	PC14	LSE(+) Onboard oscillator (32.768 kHz)
C	PC15	LSE(-) Onboard oscillator (32.768 kHz)

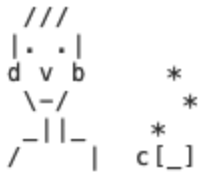
Table TN8.2 Pin table for Nucleo-L496ZG / L4A6ZG

TN9 - Troubleshooting Guide



Background

Sometimes the STM32L4 does not appear to perform properly or correctly. The following steps have been devised to help ~~quickly~~ *accurately* diagnose most common issues.



1. _____ Read (and re-read) the *fine* manual (Reference Manual and Datasheet)
2. Refer to step 1.
3. Use the hardware debugger to verify all configuration registers are set properly
4. Refer to step 1
5. Double (triple… quadruple?) check that all components are wired correctly. (*Don't forget the orientation of ICs and where pin 1 is!*)
6. Refer to step 1
7. Review the logic flow of your state diagram and flow charts (You did create them before coding right? … RIGHT?). If you realized changes you had to make after originally creating them, update them and make sure the logic flow / process still makes sense. Be especially careful with interrupts because they do not occur in sync with your main code, they can run at any point in your program!
8. Refer to step 1