Log & Level: Maintainer Document

Author: Tony Abboud
Instructor: Bill Hawkins
ENEE 440 - Fall 2013

Table of Contents

Abstract 1
Document Outline 1
1. State Machine Diagram 2
2. Program Overview 3
2.1. Startup Code 3
2.2. Main Code 3
2.3. Macros 5
2.4. Port/Switch Initialization Functions 6
2.5. Logging Voltage Meter Assembly Code 6
2.6. Accelerometer Interface Assembly Code 8
2.7. USART2 Assembly Code 10
2.8. Tasking Assembly Code 12
2.9. Display Functions 14
2.10. Rotary Encoder 16
3. Limitations of Code 16
4. Function File Listing 17
5. References 18

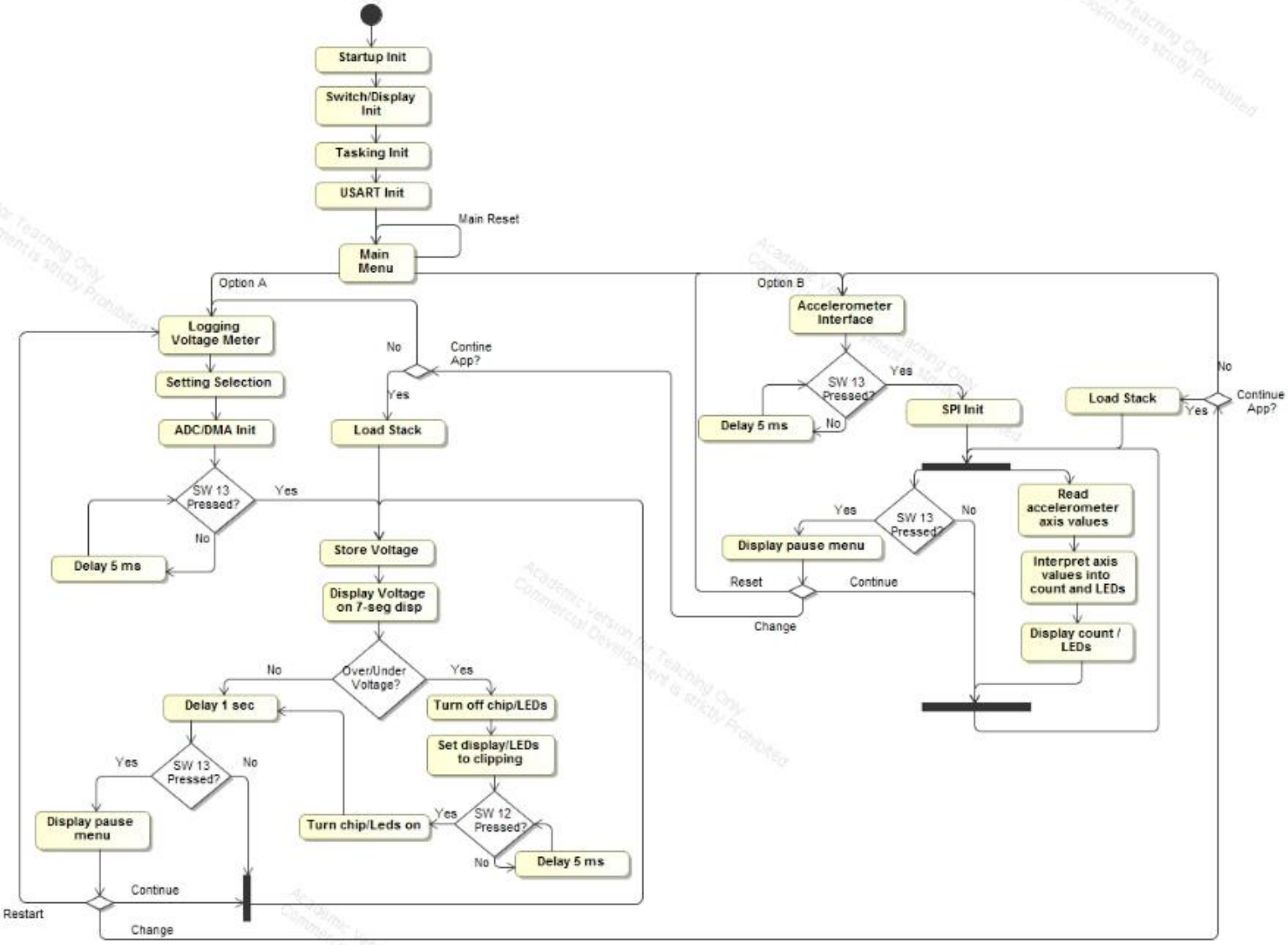
Abstract

This paper is tailored for an audience with a technical background or engineers who might modify/service the software. The goals for this project were to implement a logging voltage meter, using the onboard analog-to-digital converter (ADC), direct memory access (DMA), and universal asynchronous receiver/transmitter (USART), as well as an accelerometer interface, using the serial peripheral interface bus (SPI). In this software, the logging voltmeter and the accelerometer interface are treated as tasks or "apps", which is controlled by a very simple operating system task scheduler. The interface for the software is through serial communication (USART), which allows the user to select settings, and control which tasks are run. While an app is running, the user has the ability to pause, continue, restart or switch tasks at any time using the P24v04 expansion board. The simplified OS is capable of switching tasks and restoring the correct state of the task upon return. The logging voltmeter utilizes the STM32F4 ADC, DMA, USART, and GPIO peripherals. The ADC performs the analog conversion, while the DMA transfers the converted values to a place in memory. The USART allows the voltage to be logged on the serial terminal in real time. The user is able to enter various settings such as a sample delay time, over/under voltage, etc. The accelerometer interface utilizes the STM32F4 SPI, USART, GPIO peripherals. The SPI provides communications between the accelerometer chip, LIS302DL, and the stm32f4 board. The display for the accelerometer app is solely on the P24v04 expansion board, and does not log any values to the serial terminal. See the User manual for a complete listing of the required software/hardware components.

Document Outline

This document will first introduce the state machine diagram for the software/hardware package. Then we will delve into the software sub-systems and functions themselves to see how they interact to create the project. I will list several limitations of the software that can potentially be fixed in later versions. There are an array of figures and tables to help convey the structure and data flow of the project. Finally at the end there will be a references section, which will provide more detail and clarification on topics that I cover here, if needed.

1. State Machine Diagram



The state machine diagram above deals strictly with the flow of the software with the two peripheral applications. The diagram is still somewhat high-level, to help understand the structure without too much complication. For more information, or to see this diagram in action, consult the User Manual.

2. Program Overview

*See section 4 for the function listing directory.

2.1. Startup Code

SimpleStartSTM32F4_01.asm

This is the first code to run upon startup of the software. This code is a simple startup file for the Cortex-M4 processor. The vector table, which includes Reset, SVC, PendSV handlers, are all defined here. The first assembly function run from this file is the Reset_Handler, which initializes the hardware and system data, then calls main(). The Reset_Handler reads the data from flash memory and stores it into RAM.

2.2. Main Code

Blinky.c

This C source file contains the main() function for the project, which is called from the startup code above. This file controls the structure and processes that occur during execution. All of the peripheral header files are #included here. The source code for the SysTick_Handler() is also found here, which was commented out from the simple start code above. The Systick_Handler runs every 1ms which enables the displays to be refreshed at 1 kHz. Systick_Handler calls the various display functions, which are again #included into blinky.c. Upon entry to main(), this file first gets the core clock frequency and sets systick to a 1mS interrupt. Then main() calls several initialization functions, including init_var() (Port Initializations) and switch_init() (Expansion board switches and RE Initializations), see **section 2.4**. Main constructs the stack frames for the two tasks, voltage meter and accelerometer interface. Since we use a task scheduler that operates off of SVC and PendSV, main sets SVC to the highest priority and PendSV to the lowest using the function NVIC_SetPriority(). Main then calls the USART2 initialization function, USART2_init(), which sets the configuration of the USART2. Main utilizes the USART2 put string function to output the main menu. Based off the input from the user, which is read using the function, get_string(), main will transfer control to one of the two tasks, ADC_DMA_GO() or ACCEL_SPI_GO(), and will not return to main().

void ADC_DMA_GO();

This function is the main controller for the logging voltage meter application. Before getting input the user, this function de-initializes the ADC and DMA using the functions, ADC_DeInit() and DMA_DeInit(). This de-initialization avoids any conflicts from previous initialization processes. This function then gets user input, again through the USART_puts() and get_string() functions. The over/under voltage settings are also taken care of here by calling the function, get_Voltage(). The returned voltage is then stored into a variable and extended to 4 digits if the precision value of the rotary encoder was less than 3, upon switch 13 being pressed. The configuration of the ADC and DMA are configured with the settings gathered by the user by calling the function ADC3_DMA_Config(). Switch 13 is then polled to determine when to begin

logging the voltage. Once the switch is pressed, the ADC begins conversion by the function ADC3_START(). By DMA, the voltage value is stored into a variable, and in this function we convert the value into a character array that can be displayed on the seven segment display and terminal. The into to character transformation is executed by first taking the modulus of the voltage value by 10, then adding the ascii value of '0' to the digit. This in effect transforms the integer digit into an ascii character. For the tens, hundreds, and thousands digit we have to divide by 10, 100, and 1000 respectively before taking the modulus by 10. The voltage value received by DMA can then be compared with the over and under voltage to determine if the chip should be shutoff. If indeed the over/under voltage is exceeded, then the display is changed to "----" by setting set_negs variable to 1. The set_negs variable is tested in the refresh_VOLTdisplay() function and will output the '-' if it is set. The variable logging is set to 0, to disable LED4 from blinking. Instead LED2 and LED5 are set to ON and Red by LED_Cutoff_ON(). Switch 12 is then polled for user input. Once the user presses sw12, the logging will continue. While this function is running, the user has the ability to press sw13 to pause the application. Sw 13 is being polled every 50ms in Systick_Handler and a variable, ADC_stop, is set if it is pressed. If ADC_stop is set then this function will output the pause menu. From this menu the user can select to change apps, restart the app, or continue. If continue is chosen, nothing is done, except for resetting ADC_stop. If reset is chosen then we simple call the same function that we are in. If change is chosen, then we must initiate an SVC call to change apps by the function SVC_ACCEL_SPI(). Calling this function will in effect call SVC_Handler and PendSV_Handler, and ultimately change tasks to the accelerometer interface. If you return from the accelerometer interface, you will pick up at the line directly after the SVC_ACCEL_SPI function call, and continue processing.

void ACCEL_SPI_GO();

This function is the main controller for the accelerometer interface application. The function first polls switch 13, prompting the user to begin the application using USART_puts(). Once the switch is pressed the configuration of the SPI interface is configured using the function, my_init(). A delay of 30 ms then follows which is required to initialize the accelerometer chip. Then the function MEMS_read() is called which returns the original offsets of the accelerometer. This is used to determine the initial position of the board when the app is started. Then this function enters an infinite loop which simply reads the accelerometer axis values by the function, Poll_Acc() and then delays some time. The main processing of the accelerometer values occurs in the Poll_Acc() and display functions discussed later. Again the user has the ability to pause by pressing switch 13 which is polled in Systick_Handler, and sets the value ACCEL_stop if pressed. If the switch is pressed then the chip is powered down by the function setChip_LOW(). If continue is chosen, then the chip is powered on by setChip_HIGH(), and then the infinite loop is continued. If restart is chosen then the accelerometer values Acc_turn and Acc_val are reset and we call the ACCEL_SPI_GO(). If change is chosen then, we

initiate the task switch by the SVC call, SVC_ADC_DMA. This SVC call will transfer control back to the Voltage Meter application.

2.3. Macros

.macro SET_bit addr, bit

Performs a logical OR with 1 at the given bit position on the given address. Sets the given bit at the address.

.macro CLR_bit addr, bit

Performs a logical AND with 0 at the given bit position on the given address. Clears the given bit at the address

.macro TST_bit addr, bit

Reads the bit at the given address and returns the value (1 or 0) in R3.

.macro PORTBIT_init mode, base, pin

Initializes the GPIO_BASE and Pin to the mode specified. The modes can be 0->Output pin, 1->Input pin, or 2->Pullup Input pin. This macro sets the GPIO->MODER, GPIO->OTYPER, GPIO->OSPEEDR, and GPIO->PUPDR, based on the mode

.macro PORTBIT_config bit, GPIOx_BASE, MODE, OTYPE, OSPEED, PUPD, AF

Similar to the macro above, but this macro also configures the GPIO->AFR[0] and GPIO->AFR[1] (Alternate Function) registers.

.macro PORTBIT_write base, pin

Sets the 'bit' of the GPIO_base port to set or clear. Writes to GPIO->BSRRL if setting. Writes to GPIO->BSRRH if clearing.

.macro PORTBIT_read base, pin

Reads the value of the 'pin' at the GPIO_base->IDR register and returns it in R0

.macro CATHODE_write a,b,c,d,e,f,g,dp

Places a 0/1 pattern on the P24 Expansion Board display cathodes. The a,b,c,…dp correspond to the seven segment display cathodes (see figure 2 & 3).

.macro ANODE_write a,R,G,D4,D3,P,D2,D1

Places a 0/1 pattern on the P24 Expansion Board display anodes, (see figure 3).

R==Red	D4==Digit4	P==colon	D1==Digit1
G==Green	D3==Digit3	D2==Digit2	a==N/ A

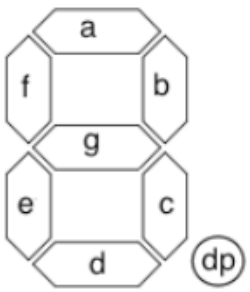


Figure 2: Seven-Segment Display

Cathodes	Ports	Anodes	Ports	LEDs	Ports
A	PC5	DIGIT1	PC2	LED1	PA1
B	PB1	DIGIT2	PA1	LED2	PC4
C	PA1	DIGIT3	PC4	LED3	PB0
D	PB5	DIGIT4	PB1	LED4	PC2
E	PB11	Green LEDs	PB0	LED5	PC5
F	PC2	Red LEDs	PB11	LED6	PB1
G	PC4	Colon	PC5		
DP	PB0				

Figure 3: Port bit Values for Cathodes, Anodes, and LEDs

.macro SWITCH_read base1, pin1, base2, pin2

This macro tests whether a switch is pressed or not. It uses two other macros, PORTBIT_write and PORTBIT_read. We first write a 0 to pin1 of base1 (switch we want to test) in order to test if the switch is pressed. We then perform a PORTBIT_read on pin2 of base 2 to determine if the returned value is 0 or 1. If the switch was pressed then the returned value of PORTBIT_read will be a 0. We then write a 1 back to pin1 of base1, in order to not test that switch again. See the figure 4 for the switch layout.

.macro MOV_imm32 reg, val

Moves a 32 bit value, 'val', into the register, 'reg', specified.

2.4. Port/Switch Initialization Functions

void init_var();

This function enables the clocks for GPIOA-D and performs the port bit initializations for the cathodes, anodes, and LEDs on the P24 expansion board. The initializations are performed using the macro PORTBIT_init

void switch_init();

This function initializes the two pull-up input pins, PA15 and PC8, that are used to read the switch values. The output pins, the switches, are all initialized to 1 (not pressed). See figure 4 below for the switch layout.

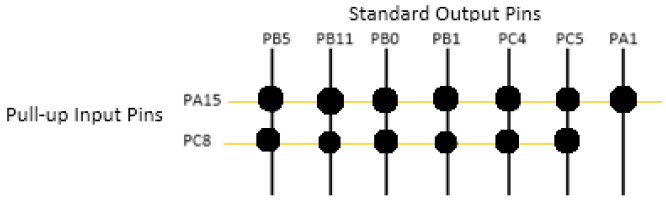


Figure 4: P24v04 Switch Layout

2.5. Logging Voltage Meter Assembly Code

void DMA_config();

-Enables the ADC3, DMA2, and GPIOC clocks

- Restarts DMA2

- Configures DMA2 Channel 2 Stream 0 (see Fig 5)

- Sets the Peripheral base address

- Sets the Memory base address

- Enables DMA2 Ch2 Stream0

- Configures PC0 as analog input

DMA2 Ch2 Stream 0 Config.		
Register	**Value**	**Description**
CHSEL	2	Channel 2
MBURST	0	Single Mem transfer
PBURST	0	Single transfer to periph.
CT	0	First buffer
DBM	0	Disable double buffer
PL	2	High priority
PINCOS	0	No increment
MSIZE	1	halfword mem. data
PSIZE	1	halfword periph. data
MINC	0	Disable
PINC	0	No increment
CIRC	1	Enable circular mode
DIR	0	Peripheral to memory transfer
PFCTRL	0	DMA
EN	0	Keep at 0 for protection

Figure 5: DMA2 Stream 0 Bit Configurations

void ADC_config(int SampleDelay);

- Initializes the common ADC registers
- Takes as input the user entered sample delay
- Initializes the ADC3 Control Registers
- Sets the number of conversions

ADC3 Initializations			
Register	**Bit_name**	**Value**	**Description**
Common Register (ADC_CCR)	MULTI	0	Independent mode
	ADCPRE	0	Div2
	DELAY	USER	User entered delay
ADC3_CR1	RES	0	12 bit resolution
	SCAN	0	Disable
ADC3_CR2	CONT	1	Enable Continuous conversion
	EXTEN	0	No external trigger
	EXTSEL	0	None
	ALIGN	0	Right Align
ADC3_SQR1	L	0	1 conversion

Figure 6: ADC3 Configurations

void ADC_Ch10_Enab(int SampleTime);

- Takes as input the user entered sample time
- Configures the ADC3 Channel 10 (ADC_SQR3)
- Enables DMA request after last transfer (Set ADC3->CR2 [DDS] = 1)
- Enables ADC3_DMA (Set ADC3->CR2 [DMA] = 1)
- Enables ADC3 (Set ADC3->CR2 [ADON] = 1)

void ADC3_START();

- Starts the software conversion by setting the bit SWSTART to 1 in the ADC3->CR2 register

void ADC_DeInit();

- Resets the ADC peripheral by setting bit 8 of RCC_APB2RSTR (clock reset register for the ADC3) and then clearing the bit to release the reset

void DMA_DeInit();

- Sets the DMA2->S0CR [EN] bit to 0, disabling the DMA2 Stream0
- Sets S0NDTR, S0PAR, S0M0AR, and S0M1AR registers all to 0 (Reset)
- Applies the reset mask to DMA->S0FCR (0x00000021) and DMA->LIFCR (0x3D)

2.6. Accelerometer Interface Assembly Code

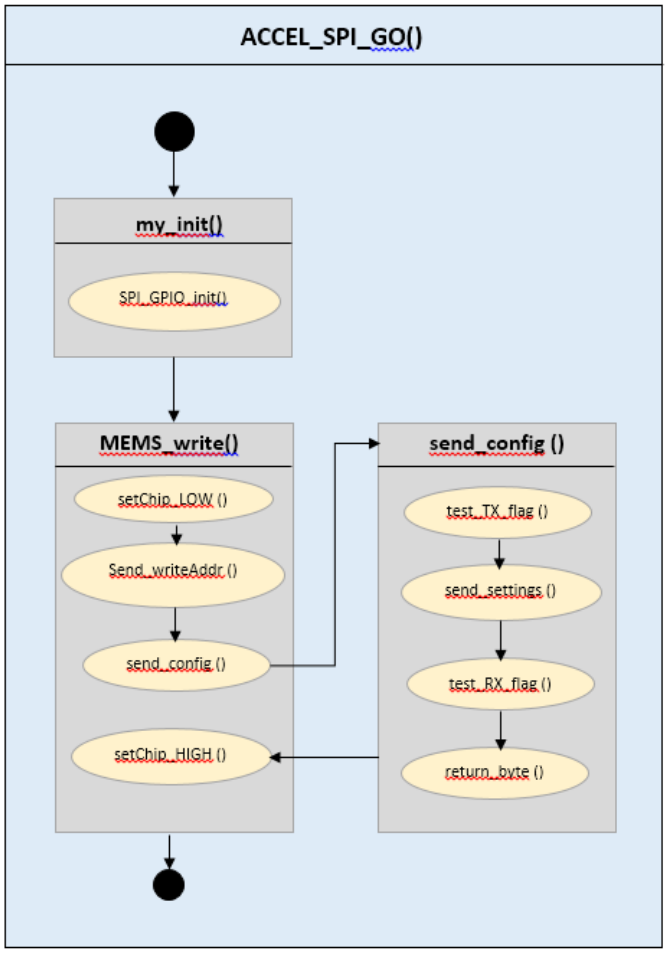


Figure 7: Accelerometer initialization functions. Gray boxes are C-Functions, and yellow ovals are the assembly functions

The diagram above illustrates the accelerometer initialization functions. The gray boxes are the C-functions and the yellow ovals are the assembly functions called by each C-function.

void SPI_GPIO_init();

- Initializes the SPI1 peripheral and GPIO
- Configures ports PA5 (SCK), PA6 (MISO), and PA7 (MOSI) as SPI1 alternate functions, using the PORTBIT_config macro
- Configures SPI1 according to the configurations in figure 8
- Enables SPI mode by resetting the I2SMOD bit in the SPI->I2SCFGR register
- Enables the SPI1 peripheral (SPI->CR1 [SPE] = 1)

SPI1 Configuration			
Register	**Bit_name**	**Value**	**Description**
SPI->CR1	BIDIMODE	0	Direction: 2 Lines Full Duplex
	BIDIOE	0	
	RXONLY	0	
	SSM	1	SPI_Mode: Master / NSS Soft
	SSR	1	
	MSTR	1	
	LSBFIRST	0	Most Significant Bit First
	BR	1	Baud Rate Prescaler 4
	CPOL	0	LOW
	CPHA	0	1 - Edge
SPI->CRCPR	CRCPOLY	7	CRC_Polynomial

Figure 8: SPI1 Register Settings

- Configures GPIO PE3 for the LIS302DL accelerometer chip

void setChip_LOW();

This function turns to accelerometer chip off by using the PORTBIT_write macro and resetting GPIOE pin 3, (set bit GPIOE->BSRRH[3]).

char send_writeAddr();

This function sends the write address through the SPI interface (puts write address into SPI->DR) and returns the byte received from the SPI bus. This function first tests the TXE flag to determine if transmission is complete, then puts the write address onto the SPI data register. Then the function waits again for the RXNE flag, which signifies that data has been received. The data is then returned.

void test_TX_flag();

This function tests the TX flag when sending data over the SPI bus. It is simply an infinite loop that keeps testing SPI->SR[TXE] for a 1. This function utilizes the TST_bit macro.

void send_settings();

- Loads the accelerometer settings into the SPI->DR register, which will be sent to configure the accelerometer (see figure 9 for settings).

Accelerometer Settings		
Bit_name	**Value**	**Description**
ODRATE	1	Output Data Rate: 400 Hz
PMODE	1	Low Power Mode: Active
FULL_SC	0	Full Scale
STP	0	Self-Test Normal
STM	0	Self-Test Normal
ZAXISEN	1	Z-Axis Enable
YAXISEN	1	Y-Axis Enable
XAXISEN	1	X-Axis Enable

Figure 9: Accelerometer Configurations

void test_RX_flag();

This function tests the RXNE flag when receiving data over the SPI bus. It is simply an infinite loop that keeps testing SPI->SR [RXNE] for a 1. This function utilizes the TST_bit macro.

char return_byte();

This function simply returns the data that is stored on the SPI data register. This function is used after testing the RXNE flag, so that data can be read off and stored.

void setChip_HIGH();

This function turns to accelerometer chip on by using the PORTBIT_write macro and setting GPIOE pin 3, (set bit GPIOE->BSRRL[3]).

void MEMS_read(uint8_t *pBuffer, uint8_t ReadAddr, int numBytesToRead); (*C-Function)

This function handles the reading of data from the accelerometer. The following assembly functions are all called from within this function. This function takes as input a character array, one 8 bit readAddress, and the number of bytes to read. The values read from the accelerometer are stored into the character array pBuffer.

void send_readAddr(uint8_t addr);

This function sends the read address, 0xE9, through the SPI->DR register. Once this data is sent to the DR register, MEMS_read calls get_FlagStatus. This function is also used to send the dummy_byte (0x00) which generates the SPI clock to the accelerometer (slave device).

int get_FlagStatus(volatile unsigned int *); (*C-Function)

This function tests the RXNE flag and returns to MEMS whether the SPI->DR register has data to be read or not. It takes as input the address of the SPI->DR register.

char receive_byte();

This function simply gets data from the SPI->DR register and returns the 8 bits through R0**.** In MEMS_read, the data returned is stored into the Buffer array and used as the accelerometer values.

2.7. USART2 Assembly Code

void USART2_init();

- Initializes the USART2 peripheral and GPIOA

- Configures PA2 and PA3 as the Tx and Rx USART2 alternate functions

- Configures the USART2 settings according to figure 10 on the right

USART2 Configuration			
Register	**Bit_name**	**Value**	**Description**
USART->CR1	M	0	Wordlength: 8 bits
	PCE	0	No Parity
	PS	0	No Parity
	TE	1	Transmitter Enable
	RE	1	Receiver Enable
USART2->CR2	STOP	0	1 Stop bit
USART2->CR3	CTSE	0	No hardware flow control
	RTSE	0	No hardware flow control
USART2->BRR	BRR	0x683	BaudRate: 9600

Figure 10: USART2 Register Settings

- Enables the USART2 peripheral (USART->CR1 [UE] =1)

void USART2_SendData(volatile char data);

- Checks the TC (Transmission Complete) flag in USART->SR register. When the flag is set the ‘data’ will be put into the USART->DR register for transmission. This function is called by the C-function, USART_puts(volatile char *). The USART_puts function passes in one character at a time, from the volatile char *, to this function. This function then handles the sending of the characters over USART.

int RxFlag_Status();

This function simply checks the value of the USART->SR [RXNE] and returns the value, either 1 or 0. If the RXNE flag is set, then this function will return a 1, and will return a 0 if the flag is not set. This function is used by the C-function, get_string(). The get_string() function calls this function to test when to read from the USART->DR register.

int TcFlag_Status();

This function simply checks the value of the USART->SR [TC] and returns the value, either 1 if set or 0 if not set. This function is used by the C-function USART_puts(volatile char *). The USAR_puts() function calls this function to test if the transmission of a byte was complete, in order to send the next byte.

char get_Char();

This function returns a character read from the user via USART. This function reads the USART->DR register and returns the value to the get_string() function.

void get_string(); (*C-function)

This function is used to get input from the user via USART. This function first calls the RxFlag_Status() function to determine if any characters have been written. Then this function calls the get_Char() function to get one character. Immediately this function checks to make sure the character read in is not the '\n' or carriage return character. It also checks to make sure the max amount of characters has not been entered, 12 characters. If both situations succeed, then the character is stored into an array, received_string[] and the function continues reading characters. Once one of the conditions is false, the receieved_string[] array stores the '\0' and the function ends.

void USAR_puts(volatile char *); (*C-function)

This function is used to output a string onto the serial terminal via USART. This function takes in a character array. This function first checks the transmission complete flag to ensure that a character can be sent. Then this function calls the USART2_SendData() function to send one character at a time over USART. The function continues to run until the array reaches the null character, '\0'.

2.8. Tasking Assembly Code

The following functions handle the concept of task switching for the cortex M4 processor.

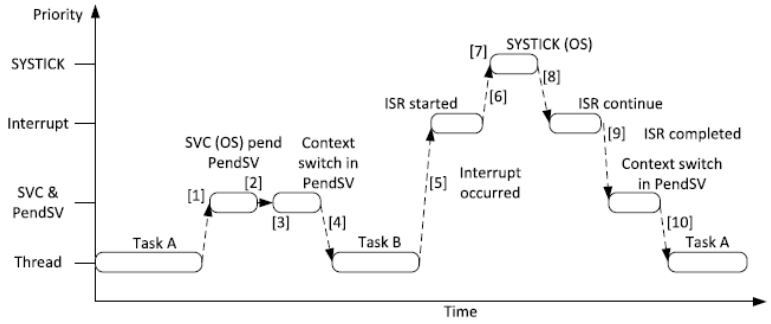


Figure 11: Context Switching

The diagram above illustrates context switching. The idea is to be running one task, then execute an SVC call (highest priority), which will enable the PendSV interrupt (lowest priority) to switch tasks. The tasks will then be switched when PendSV executes (after all other interrupts complete) and the new task will then run.

void SVC_Handler();

Interrupt code for the Service Call instruction, SVC #x. This code is executed after an SVC instruction is executed, since an SVC interrupt is set to the highest priority, as seen in the main() function. This code is intended to simply perform a task switch and nothing else. Therefore, the first step is to enable the PendSV interrupt, which will perform the actual switching of the task. The SVC_Handler then decodes the instruction itself by loading the stacked program counter and loading the first byte of the SVC instruction. This is a way of determining which SVC number was called, and ultimately which task to switch to, (**see figure 12**). The handler then performs a simple if statement based on the two values of the SVC number, 0 or 1. The variable, next_task, is then assigned the SVC number that was decoded and the SVC_Handler exits. Upon Exit from the SVC_Handler, we enter the PendSV_Handler interrupt, assuming there are no other higher level interrupts to handle. Proceed to the PendSV_Handler().

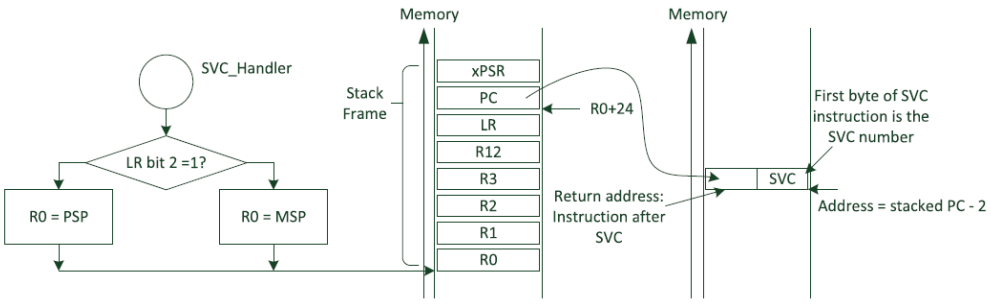


Figure 12: Extraction of SVC service number

void PendSV_Handler();

This code is triggered solely by an SVC call to switch tasks. Therefore, the code for this function is to just switch tasks. First the current context must be saved. The current process stack pointer is retrieved, then we save all registers not already pop'd onto the stack, R4 – R11. Then the current task value is loaded along with the process stack pointer array, PSP_array[]. The process stack pointer value is then saved into the PSP array at the corresponding index. Next, the next context (task) is loaded. The next_task value is loaded and set as the current task. Then the PSP value is loaded from the PSP_array along with R4-R11. All that is left is to set the PSP to the next task. The PendSV_Handler can then branch and link back, (see **figure 13** below).

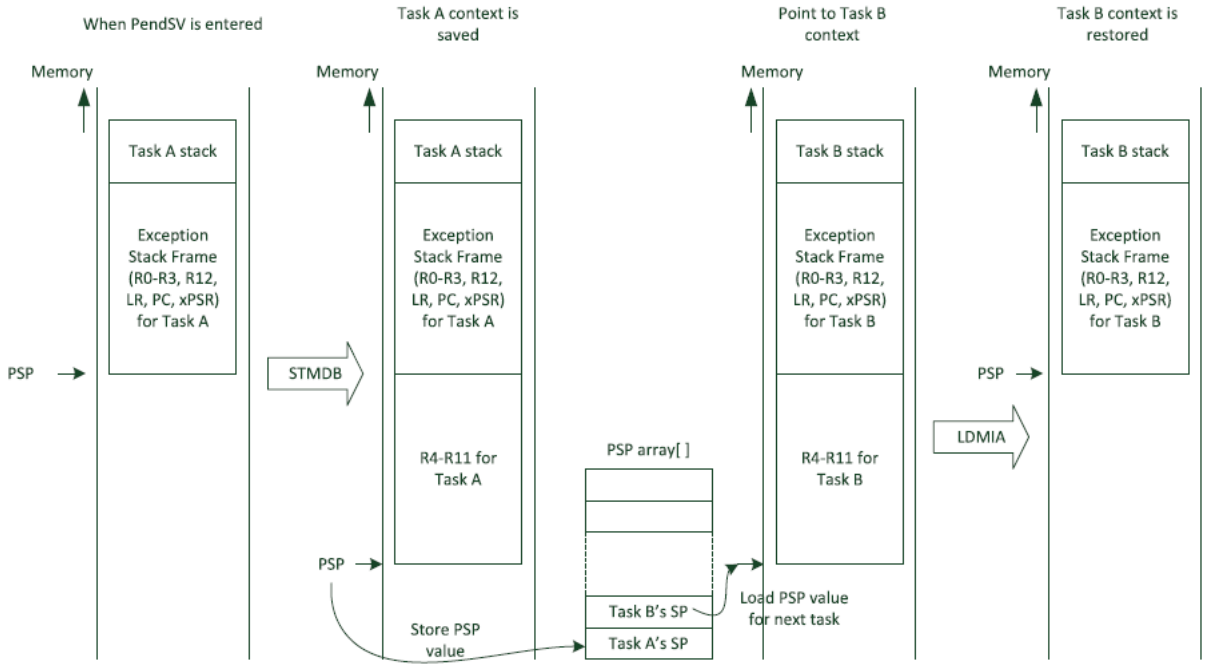


Figure 13: Task Switching (PendSV_Handler) in Action

void SVC_ACCEL_SPI();

Performs the context switch to task 1 (ACCEL_SPI). The task switching is performed via an SVC call. In this function, the code is simply just "SVC #1". This SVC call triggers the SVC_Handler interrupt code to execute which will eventually switch tasks to the ADC_DMA application.

void SVC_ADC_DMA();

Performs the context switch to task 0 (ADC_DMA). The task switching is performed via an SVC call. In this function, the code is simply just “SVC #0”. This SVC call triggers the SVC_Handler interrupt code to execute which will eventually switch tasks to the ADC_DMA application.

2.9. Display Functions

void decode_pattern(char *buffer); (*C-function)

This function is called when a value needs to be displayed on the board. This function takes the digit value to display and the display number, and converts then into a cathode and anode pattern that can be used in the SET_CAt and SET_ANt functions. This function calls the set_pattern() function with the decoded cathode and anode patterns.

void set_pattern(int a, int b, …. int h, int task, char *arr); (*C-function)

This function takes as input cathode and anode patterns, a task number to discern the cathodes from the anodes, and an array to store the cathode and anode patterns. This function stores the cathodes first then the anodes. It goes through each value passed in from decode_pattern() and tests it to see if the particular cathode/anode should be on(0) or off(1/0x0A). The constructed array is what eventually gets passed into the decode_pattern() function.

int load_SWque(int msTicks, int sw_num, int action);

This is my version of the intended void get_sw() function. This function takes in the msTicks time, the switch number, and the action (pressed ‘1’ or unpressed ‘0’). This function returns the value or ”event” to hold in the que by combining the 3 passed arguments into one 32 bit value. The format for the returned value is: 0xTTTTTTAS , T-Time, A-Action, S-Sw_num. This function is called in the function read_switches(), whenever a switch is first pressed.

void read_switches();(*C-function)

This function polls the switches and calls the load_SWque function when needed. If a switch is pressed the load_SWque function is called and stores the event in the sw_que. From that point on the same switch is continually polled. Once the switch is unpressed, the event is received from the load_SWque() function and is stored into the sw_que. This sw_que is used in the get_voltage() function to get the user input for the over/under voltage.

void display_this(char *displaybuf);

This function sets the 8 anode/cathode patterns to the display latches. This function takes in a character array, displaybuf, which has the format: { {cathodes},{anodes} } (separate sub-arrays of cathodes, anodes). This function calls SET_Cat with the first 8 values of displaybuf to set the requested cathode pattern and it calls SET_Ant with the last 8 values to set the requested anode pattern.

void SET_CAt(int index);

This is a look-up table for the various cathodes, a, b, c,….dp. This lookup table selects the 'index' and performs a PORTBIT_write to set the particular cathode. This function is combined with the display_this() and SET_ANt() functions to display a value on the display.

void SET_ANt(int pos);

Turns on the requested DIGIT, 'pos'. This function is a look-up table for the 4 different seven segment displays. This function utilizes the PORTBIT_write function to enable the anodes for the displays. This is called by the display_this() function.

void refresh_VOLTdisplay(); (*C-function)

This function is used to set the display for the logging voltage meter application. It first decodes the voltage value into an array (using the decode_voltage() function) then displays each value on the corresponding display. This function can also discern when to display "----" for the cutoff voltage, or a blank screen for initializations.

void refresh_SWdisplay(); (*C-function)

This function is used to set the display for the over/under voltage settings. This function decodes the test_voltage value, entered from user, and displays the digits onto the corresponding display at a 1 KHz refresh rate. This function also determines where to place the decimal place based on the variable, precision_cnt.

void refresh_ACCELdisplay(); (*C-function)

This function is used to set the display for the Accelerometer interface application. Just like the refresh_VOLTdisplay() function, this function decodes the Accelerometer value and places the correct digits onto the display at a 1 KHz refresh rate. This function also determines when to place the negative sign, or blank screen based on the Acc_value.

2.10. Rotary Encoder

int RE_position();

This function returns 1 if the rotation was clockwise, 2 if the rotation was counter-clockwise, or 0 if there was no rotation. The rotary encoder is tested by reading the pull-up input pins connected to the switches. Side 'A' of the RE is connected to one input pin and side 'B' is connected to the other input pin (see figure 14). When A transitions from low to high, we check the state of B. If B is high at the A (l to h) transition then the rotary encoder was turned clockwise. If B is low at the A(l to h) transition, then the rotary encoder was turned counter-clockwise (see figure 15).

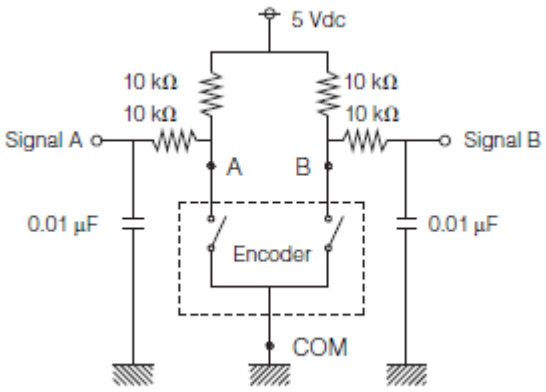


Figure 14: Rotary Encoder Schematic

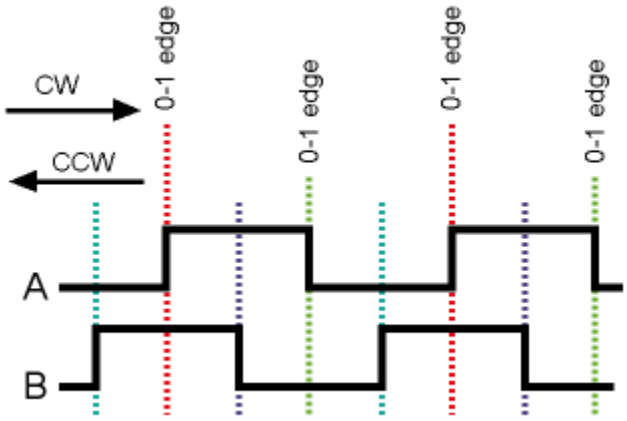


Figure 25: Rotary Encoder Waveform

void RE_reset();

This function just resets the precision_cnt value back to 0. This in effect resets the state of the rotary encoder.

3. Limitations of Code

- The polling of the switches is very touchy, especially for the Accelerometer interface. The polling of switch 13 is sometimes missed on this application, so it is needed to press again. This can be fixed by tying switch 13 to an external interrupt.

- There are several locations where I calculate the register setting values and input them. This does not allow for further additions, since the settings would be completely changed. It would be better to have every setting as a separate line of code, which could then be altered individually.

- The USART2_puts is not tied to an interrupt, which ties up the processor.

- Many data structures are set limits, which the code is tailored to. This limits the reusability of the code itself for further additions.

4. Function File Listing

Source Files				
SimpleStartSTM32F4_01.asm	Blinky.c	CortexM4asmOps_01.asm	stm32f4_P24v04_definitions.asm	stm32F4xx_DISPLAY_Func.c
Reset_Handler()	main()	init_var()	.macro SET_bit	set_pattern()
	Systick_Handler()	switch_init()	.macro PORTBIT_init	decode_pattern()
	Delay()	SET_CAt()	.macro PORTBIT_write	refresh_ACCELdisplay()
		SET_Ant()	.macro PORTBIT_read	refresh_Swdisplay()
	ADC_DMA_GO()	getSWITCH()	.macro CATHODE_write	read_switches()
	ADC3_DMA_Config()	DISPLAY_on()	.macro ANODE_wirte	refresh_VOLTdisplay()
	get_Voltage()	DISPLAY_off()	.macro SWITCH_read	decode_voltage()
		load_SWque()		
	ACCEL_SPI_GO()	RE_reset()		
	Poll_Acc()	RE_position()		
	TimingDelay_Decrement()	put_neg()		
	my_init()	display_this()		
	MEMS_write()	setONES()		
	MEMS_read()	write_PD()		
	send_config()	level()		
	get_FlagStatus()	turn_left()		
	turn_on_LED()	turn_right()		
	decode_Acc_val()	LED_Logging_ON()		
		LED_Cutoff_ON()		
	USART_puts()	LED_OFF()		
	get_string()			

Peripherals			
stm32f4xx_ADC_DMA.asm	stm32f4xx_MEMS.asm	stm32f4xx_USART2.asm	stm32f4xx_Tasking.asm
.macro MOV_imm32	SPI_GPIO_init()	USART2_SendData()	PendSV_Handler()
.macro CLR_bit	configure_MEMS()	USART2_init()	SVC_Handler()
.macro TST_bit	send_writeAddr()	RxFlag_Status()	SVC_ACCEL_SPI()
.macro PORTBIT_config	test_TX_flag()	TcFlag_Status()	SVC_ADC_DMA()
DMA_config()	test_RX_flag()	getChar()	
ADC_config()	send_settings()		
ADC_Ch10_Enab()	return_byte()		
ADC3_START()	setChip_LOW()		
ADC_DeInit()	setChip_HIGH()		
DMA_DeInit()	send_readAddr()		
	receive_byte()		

5. References

- *UserManual.pdf*: Detailed description of the process and functionality of the system
- *QA_Testing.pdf:* Testing procedures for the software/hardware
- *STM32F4 Reference Manual* (*DM00031020.pdf):* Detailed descriptions for each peripheral used and the specific register and bit settings
- *LIS302DL.pdf*: Accelerometer Data Sheet
- *www.ST.com/stm32f4 :* Main website for the stm32f4 discovery board
- *Panasonic_ATC000CE4.pdf:* Rotary Encoder Data sheet
- Yiu, Joseph. *The Definitive Guide to ARM® Cortex®-M3 and Cortex-M4 Processors*. N.p.: n.p., n.d. Web.