ECE-395 Microprocessor Laboratory

Department of Electrical and Computer Engineering

New Jersey Institute of Technology

Newark, NJ

Revision 3.16

January 17, 2017

Revised by

David Harrison

Preface: 2014 Course Update 3
1 Introduction 4
1.1 Course Objectives 5
1.2 References 5
2 Microprocessor Experiment Platform 6
2.1 Initial Tool Setup 7
2.2 Board Setup 9
2.3 Developing With the Course Project File 12
3 Experiments 16
3.1 Lab 1 – Microprocessor Operation 16
3.2 Lab 2 – General Purpose Inputs and Outputs 25
3.3 Lab 3 – Annunciator (GPIO Application) 36
3.4 Lab 4 – UART Serial Port 39
3.5 Lab 5 – Calculator (UART Application) 58
3.6 Lab 6 – Digital-to-Analog Converter 64
3.7 Lab 7 – Analog-to-Digital Converter 71
3.8 Lab 8 – Interrupts and Exceptions 82
4 Troubleshooting 87
5 Document Change History 88

Preface: 2014 Course Update

This manual is for the 2014 update of ECE-395: Microprocessor Laboratory. The major change made in the 2014 update is the adoption of the ARM architecture as the course experiment platform. This change follows the fall of 2013 change in ECE-252 to focusing on the ARM as the primary architecture studied in the lecture course.

Prior to changing to the ARM, ECE-252 and ECE-395 dating back to the mid-1990's studied the Motorola 68k architecture. During this time, ECE-395 had several incarnations. At first students hand assembled (using wire-wrapping techniques) 68000 based single board computers (SBC's) and wrote code for various experiments on the platform. In the early 2000's, the SBC design was re-implemented as a PCB design which eliminated the need for the students to wire-wrap. In the later 2000's, the lab evolved to use a Coldfire Microcontroller Evaluation board from Freescale (also a 68k core). During this time, software for the laboratory assignments was done strictly in assembly language.

The 2014 update leverages some of the experiments that date back to the original versions of this course but have been updated and modified where necessary due to the platform change or to enhance the educational value of the course. Another change is the incorporation of higher level programming (i.e. C) for some of the experiments instead of solely using assembly language for the course. This change is to more closely follow trends in industry and to allow for the execution of more complicated labs.

1 Introduction

Microprocessors touch almost every aspect of modern life. They can be found in vehicles, consumer electronics, communications devices, appliances, toys and of course computers. With the exception of full computers (i.e. desktop or full laptops) which use full microprocessors with separate memory and peripherals, most applications today use system on chips (SoC's) or microcontrollers (MCU's) which integrate most of the processing and peripheral functions into a single integrated package.

A wide variety of processor architectures are available to address many applications. The selection of a particular architecture for a design might be done for a range of reasons including technical features (e.g. speed, peripherals, power consumption, etc), cost, compatibility with existing software or previous experience of the designers.

One of the dominant architectures for SOC's and MCU's in the market today is the ARM family of processors. ARM core devices are manufactured by dozens of IC vendors for a wide range of applications. ARM's command a huge market share in smart phones and appliances and are constantly growing in the embedded arena. Because of ARM's current and anticipated future market standing, it is an attractive architecture to use as a learning platform to explore microprocessors behaviors and their use.

All processors fundamentally run architecture specific machine languages to operate. Writing programs directly in machine language is rarely done so assembly languages are used to provide a more human friendly way to generate machine language. When higher level languages are complied or interpreted, machine language is generated to runs on the processor.

In industry, microprocessors are usually programmed in higher level languages for a variety of reasons including speeding development time, code portability, etc. The language used varies depending on the application. C is commonly used for embedded systems, operating systems and device drivers. Both C and C++ are used for applications along with languages such as Java, C#, Python where higher capability operating systems are used.

In order to truly understand how microprocessors operate, observing and manipulating the behavior of the processor at the machine language level is necessary. Because of this, a large part of this course focuses on development using assembly language. Later labs do introduce the use of C to allow for more complicated programs and to see how higher level languages are translated into machine language.

1.1 *Course Objectives*

1. Understand and apply the fundamentals of assembly level programming of microprocessors.
2. Work with standard microprocessor interfaces including GPIO, serial ports, digital-to-analog converters and analog-to-digital converters.
3. Troubleshoot interactions between software and hardware.
4. Analyze abstract problems and apply a combination of hardware and software to address the problem.
5. Use standard test and measurement equipment to evaluate digital interfaces.

1.2 *References*

FRDMKL25ZUM: FRDM-KL25Z User's Manual
This document provides an overview of the hardware board used in the course.

KL25P80M48SF0: KL25 Sub-Family - Data Sheet
This document provides details on the microcontroller used on the FRDM-KL25Z (specifically the electrical specifications for the part).

KL25P80M48SF0RM: KL25 Sub-Family - Reference Manual
This document provides details on the microcontroller control registers, IO assignments, etc.

KLQRUG: KLQRUG, Kinetis L Peripheral Module Quick Reference - User Guide
This document provides gives examples on how to operate the microcontroller for typical applications.

OPENSDAUG: OpenSDA - User Guide
This document describes the debug interface used on the FRDM-KL25Z.

Cortex™-M0+ Devices Generic User Guide

Cortex-M0+ Technical Reference Manual

2 Microprocessor Experiment Platform

The microprocessor platform used for this course is the NXP (formerly Freescale) Freedom Development Platform for Kinetis KL14/15/24/25 MCUs (aka FRDM-KL25Z or KL25Z). The KL25Z provides low cost (less than $15) platform to explore microprocessor principles.

The KL25Z features:

- Kinetis-L MCU (MKL25Z128VLK4)
  - ARM Cortex-M0+ core, up to48MHz CPU speed
  - 128kB FLASH
  - 16kB SRAM
  - DMA
  - UART / 2 SPI / 2 I^2C
  - 12-bit DAC
  - 16-bit ADC (up to 24 inputs)
  - USB 2.0 OTG/Host/Device
- Capacitive touch slider
- MMA8451Q accelerometer (I^2C)
- Tri-color (RGB) LED
- USB, coin cell battery, external source power supply options
- I/O via Arduino compatible I/O connectors (53 I/O's available)
- Programmable OpenSDA debug interface

The FRDM-KL25Z does not come with headers installed for accessing the board IO. Headers need to be solderd to the board to gain acces to the IO for some of the experiments in this course. Recommended headers are available from Digikey are:

Quantity	Part	Description
1	S6106-ND	Female, thru-hole, 20 Pin, dual row, 0.1" pitch, 8.5mm high
2	S7111-ND	Female, thru-hole, 16 Pin, dual row, 0.1" pitch, 8.5mm high
1	S7109-ND	Female, thru-hole, 16 Pin, dual row, 0.1" pitch, 8.5mm high

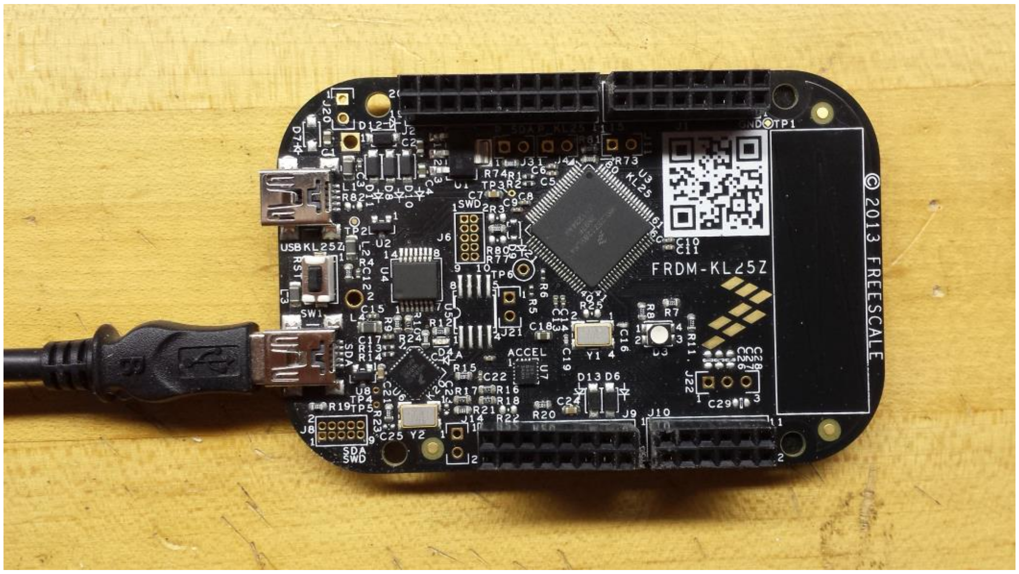


Figure 1 - FRDM-KL25Z with headers installed

Several software development tool sets support this processor and specifically the KL25Z including:

- Codewarrior Development Studio
- IAR Embedded Workbench
- KEIL MDK uVision
- mbed

The KEIL toolset has been selected for this course. A limited free version of is available, MDK-Lite, which is suitable to meet the development needs of this course.

The KEIL MDK uVision toolset features:

- Support for Cortex-M, Cortex-R4, ARM7, and ARM9 devices
- Support for C, C++ and assembly
- µVision4 IDE, debugger, and simulation environment
- CMSIS Cortex Microcontroller Software Interface Standard compliant

2.1 *Initial Tool Setup*

The following steps setup a Windows PC for developing and debugging programs on the FRDM-KL25Z.

Step 1: From the KEIL website (registration required), download and install KEIL MDK-ARM uVision 5. Note the free MDK-ARM Lite is sufficient for the lab. (https://www.keil.com/demo/eval/arm.htm)

Step 2: Open uVision. Select Project >>> Manage >>> Pack Installer…

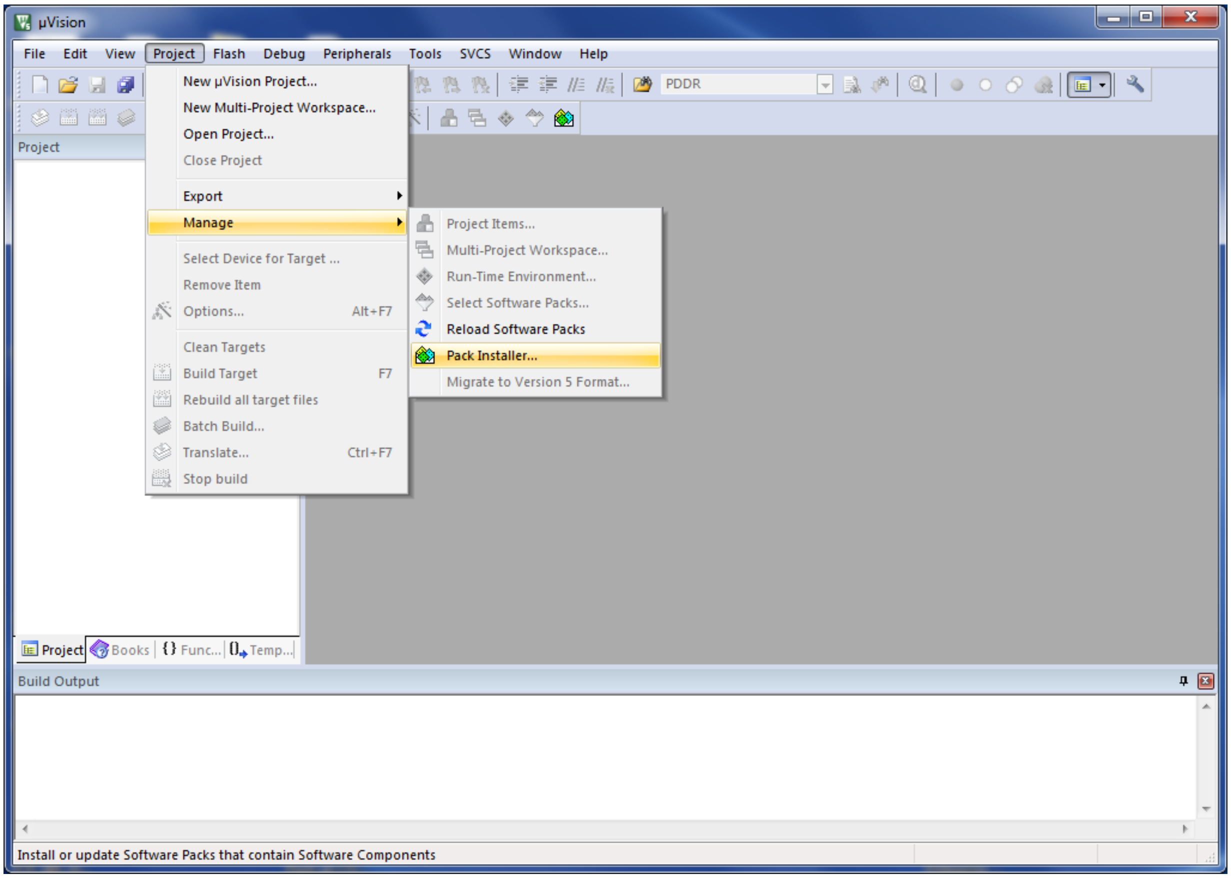


Figure 2 - Running Pack Installer

Step 3: In Pack Installer, search for MKL25Z128 under Devices. Hit Install on the pack called Keil::Kinetis_KLxx_DFP.

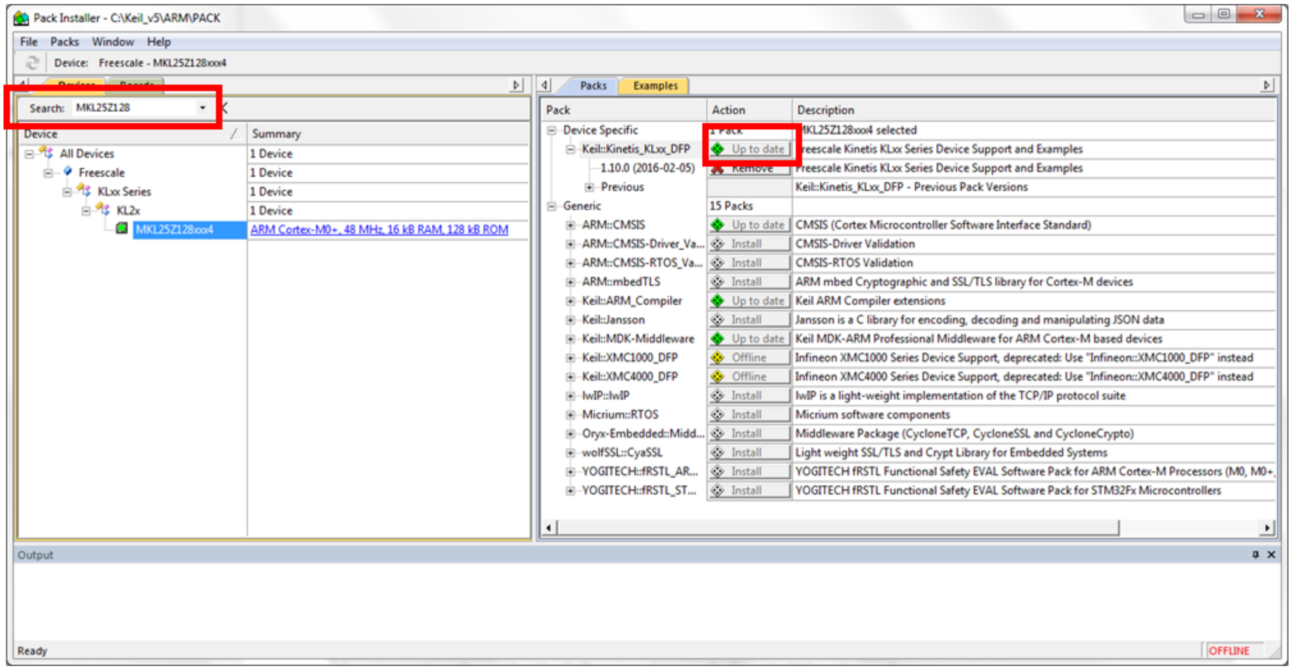


Figure 3 - Installing Device Pack

Step 4: From the KEIL website, download and install the Freescale Kinetis OSJTAG Drivers. (http://www.keil.com/download/docs/408.asp)

Step 5: From PE Micro website (registration required), download and install the OpenSDA Windows USB Drivers. (http://www.pemicro.com/opensda/)

2.2 *Board Setup*

The following steps must be executed to allow code to be loaded and debugged using the KEIL tools and software project used in the labs. This process only needs to be done once initially on a new board or if there are problems connecting to the board.

You can check the bootloader and application versions connecting the KL25Z SDA USB connector to a PC and in Windows Explorer, opening the file SDA_INFO.HTM in the drive labeled FRDM-KL25Z (Figure 5) and (Figure 6). Bootloader version 1.11 and application version 1.18 are the latest at the time this was written.

Note that this process may not work on computers running Windows 8 or newer if the KL25Z board has a bootloader older than version 1.11. If your personal PC has windows 8 or newer and this process does not work, use a PC in the lab running Windows 7 to update your boards firmware.

Step 1: From PE Micro website (registration required), download and extract the OpenSDA Firmware. (http://www.pemicro.com/opensda/).

Two files will be needed from this file:

1) MSD-DEBUG-FRDM-KL25Z_Pemicro_v118.SDA
2) BOOTUPDATEAPP_Pemicro_v111.SDA (which is in a second zip file called OpenSDA_Bootloader_Update_App_v111_2013_12_11.zip in the OpenSDA Firmware zip file)

Step 2: Connect the "USB B" end of a "USB B" to "USB Mini" cable to the development PC

Step 3: While holding the RST button on the KL25Z, connect the "USB Mini" connector of the USB cable to the connector labeled SDA on the KL25Z.

Step 4: Release the RST button. The D4 LED should flash green.

Step 5: In Windows Explorer, open the drive labeled BOOTLOADER. (Figure 4)

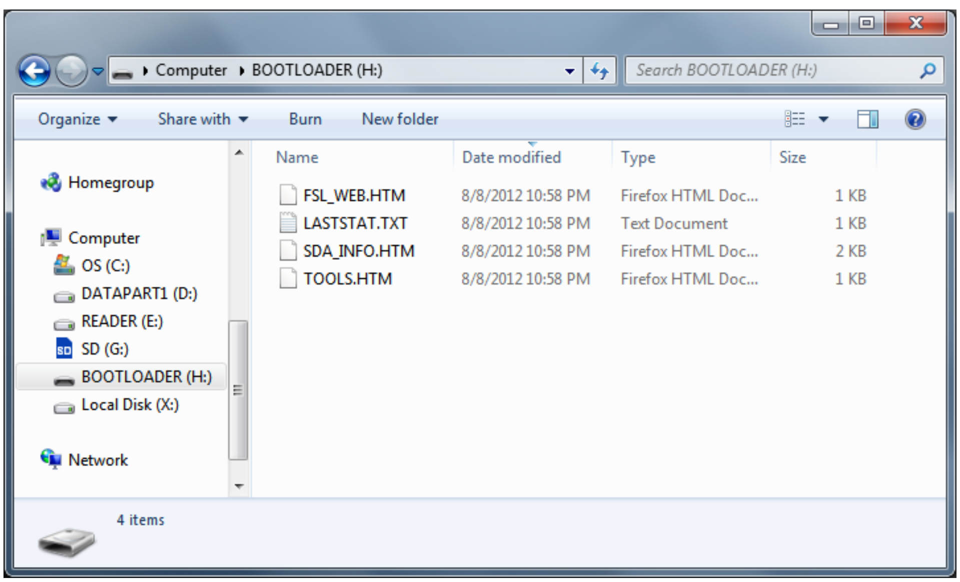


Figure 4 - BOOTLOADER drive

Step 6: Copy the firmware file BOOTUPDATEAPP_Pemicro_v111.SDA to the BOOTLOADER drive.

Step 7: Wait 15 seconds, disconnect the USB cable then reconnect the USB cable, wait another 15 seconds. It is important to wait to ensure

the firmware update has time to complete. The D4 LED should flash green when complete.

Step 8: In Windows Explorer, open the drive labeled BOOTLOADER. (Figure 4)

Step 9: Copy the firmware file MSD-DEBUG-FRDM-KL25Z_Pemicro_v118.SDA to the BOOTLOADER drive.

Step 10: Disconnect and reconnect the USB cable from the KL25Z.

Step 11: In Windows Explorer, the drive should now be labeled FRDM-KL25Z. Proper installation can be verified by opening the file SDA_INFO.HTM (Figure 5) in the FRDM-KL25Z and verifying the application version matches that of the firmware files that were loaded (Figure 6).

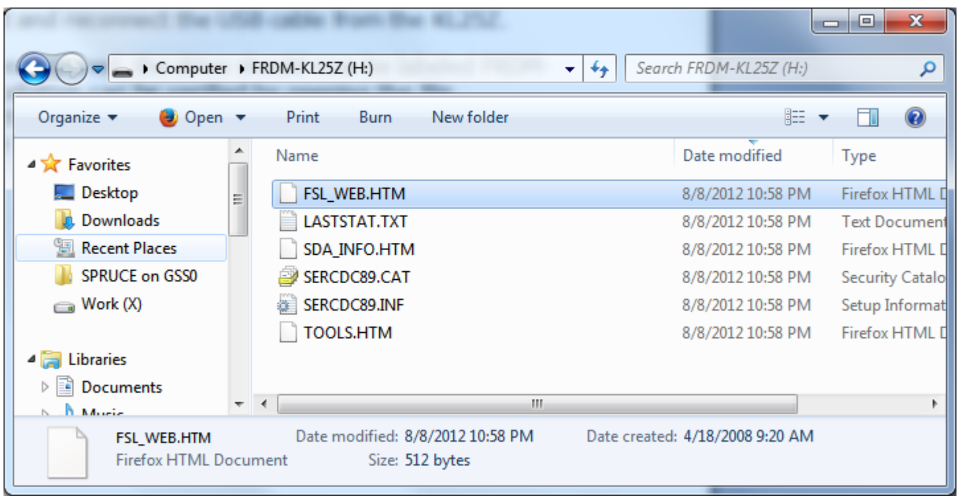


Figure 5 - FRDM-KL25Z drive

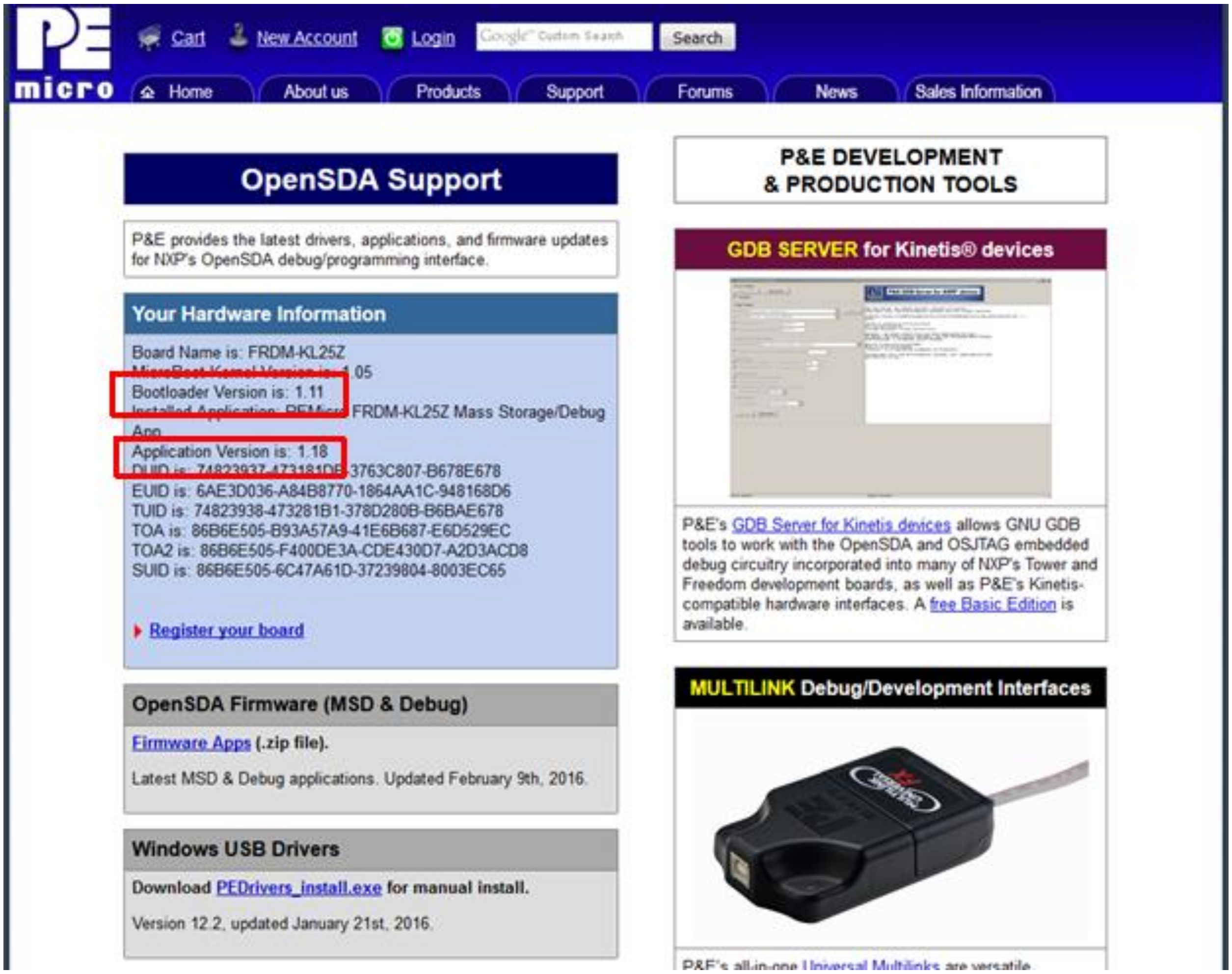


Figure 6 - Verifying KL25Z Firmware Versions

2.3 *Developing With the Course Project File*

This section describes how to use the customized course project file with the KEIL tools. This project file allows for both ASM and C programs targeted towards the KL25Z. The project only supports the PE Micro OpenSDA driver and only supports loading code into RAM.

For each experiment, it is HIGLY RECOMMENDED that you start a new project in a new directory.

There are different versions of the ECE395 Project File available to support various versions of the Keil tools. ECE395_ML25Z_Project_v2.zip supports Keil version 5.01 which is usually installed on the NJIT lab PC's. ECE395_ML25Z_Project_v2_5.##.zip (where ## matches a Keil tool version) is for use on more recent versions of the Keil tools. For example,

ECE395_ML25Z_Project_v2_5.21.zip is known to work with the most current version (at the time this was written), Keil version 5.21a.

If you want to move between older Keil versions like 5.01 and new versions like 5.21a (for instance moving between the NJIT lab PC's and a personal laptop with newer tools), it is easiest to create a separate project directories for each using the appropriate project file and to copy the main.c and asm_main.s source files between the directories with switching tool versions.

Step 1: Download from the course website the latest version of ECE395_ML25Z_Project zip file that matches the version of the Keil tools you are using and extract it to your working directory (Figure 7).

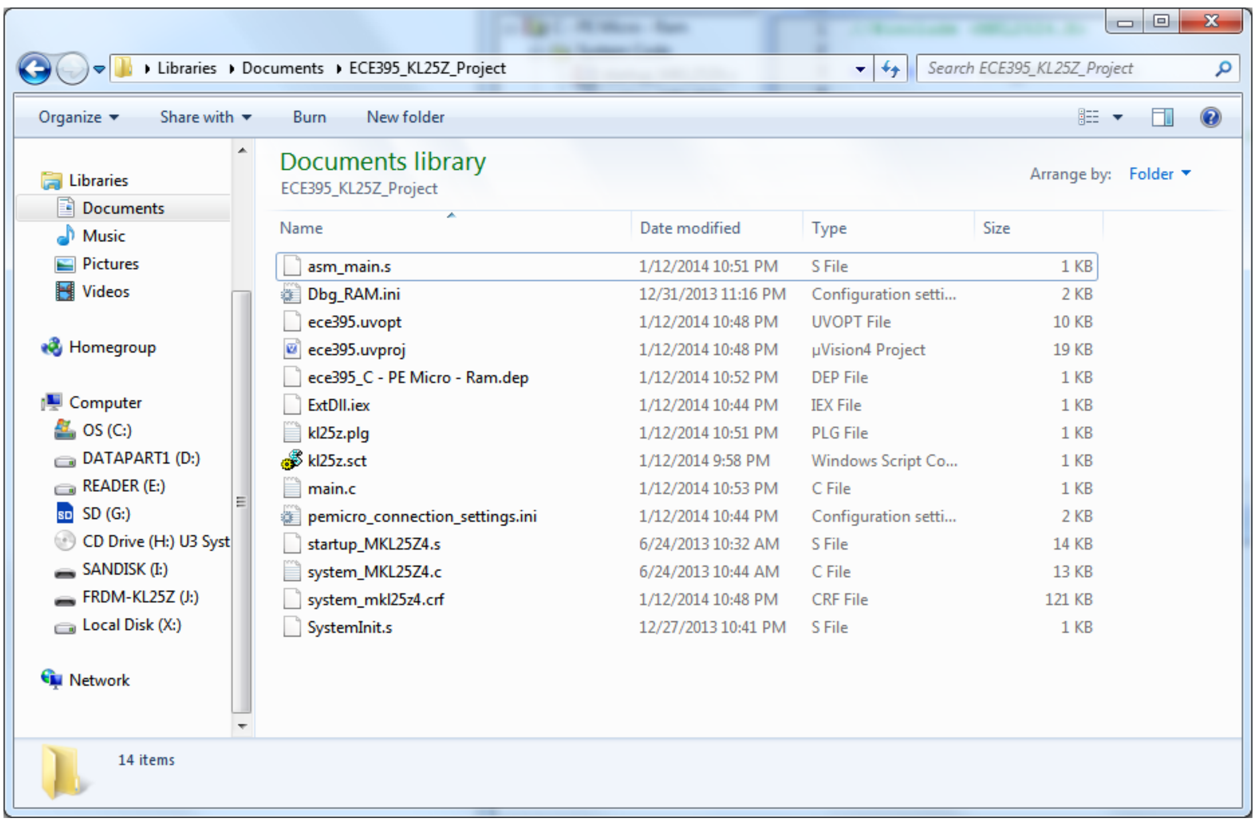


Figure 7 - Files in default project

Step 2: Start KEIL uVision5.

Step 3: Navigate to Project >> Open Project. Then open the file called ece395.uvproj.

Step 4: In the project, open the file "main.c". (Figure 8)

Step 5: If this is a program in assembly, in "main.c", uncomment the beginning of the line asm_main(); and open the file "asm_main.s". (Figure 9)

Step 6: Add your code to the appropriate file, main.c for a C project or "asm_main.s" for an assembly project. Follow the comments in the templates to keep the code in the correct sections.

Step 7: Navigate to Project >> Build Target (hotkey F7). In the build output window, make sure there are no errors and that any warnings are understood.

Step 8: Ensure the SDA connector on a FRDM-KL25Z running the PE Micro Firmware (see section 2.2 - Board Setup) is plugged into a USB cable connected to the PC.

Step 9: Navigate to Debug >> Start/Stop Debug Session (hotkey Ctrl + F5). (Note a warning about a 32k size limit may appear if you are using the lite version of the tool, this is ok).

Step 10: At this point, code can be ran, single stepped, etc.

To edit the code, the debug session needs to be stopped, navigate to Debug >> Start/Stop Debug Session (hotkey Ctrl + F5) and Step 6: through Step 9: need to be repeated

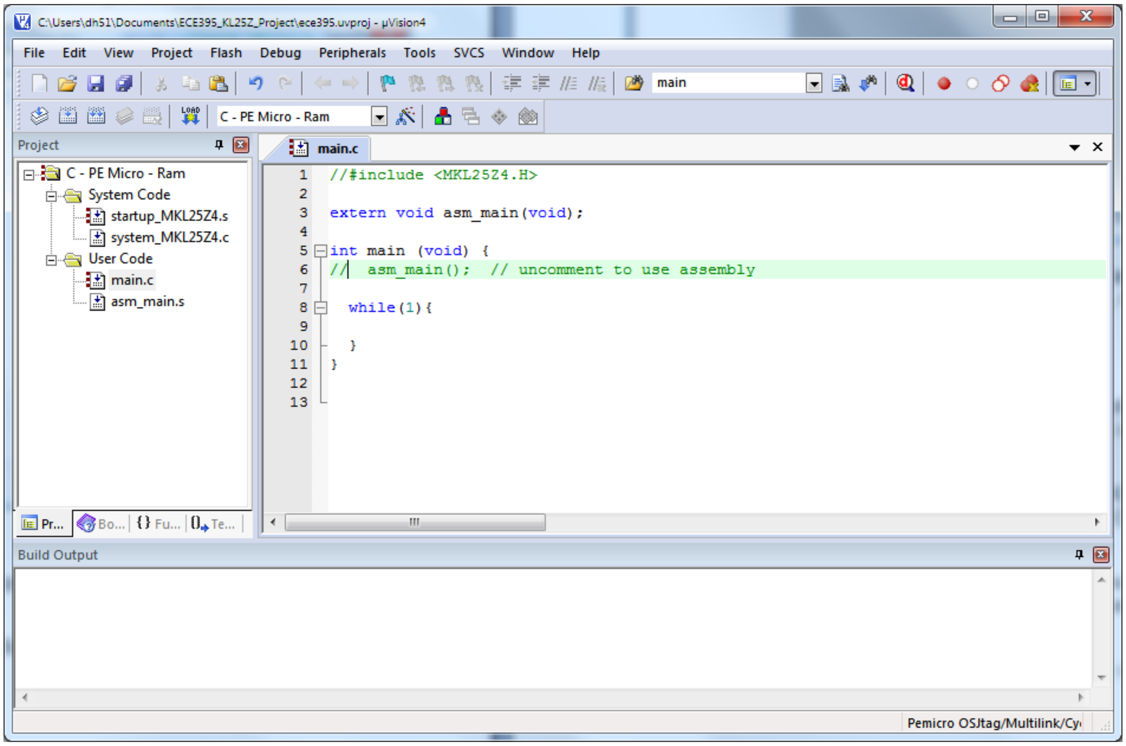


Figure 8 - Default project file main.c

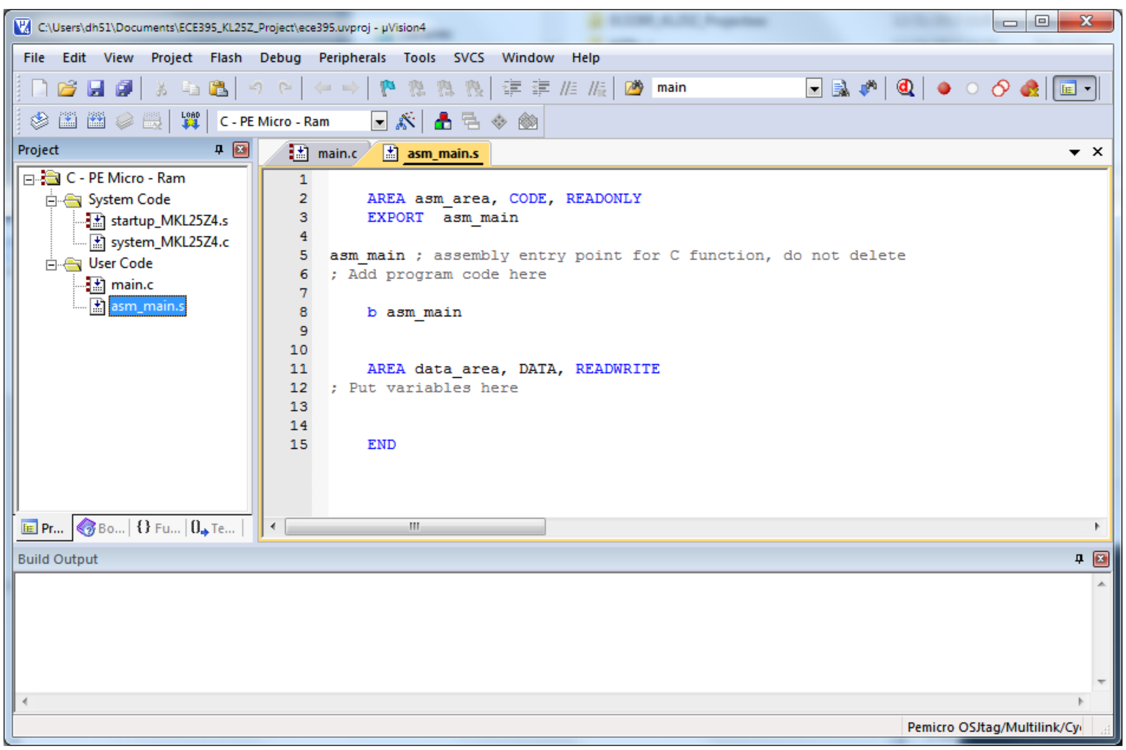


Figure 9 - Default project file asm_main.s

3 Experiments

3.1 *Lab 1 – Microprocessor Operation*

<u>Lab Objectives</u>

- To learn to create a uVision project then write, assemble and debug code
- To observe and document operation of microprocessor core as it executes code

<u>Procedure</u>

From the course website, download and print the lab worksheet form for Lab 1.

Follow the steps outlined in section 2.3 which explain how to create a new project file. Configure the project to be an assembly project by uncommenting the call to asm_main() in main.c

Add the code for each part to the asm_main.s file.

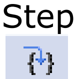
Build then debug the code as explained in section 2.3. Figure 10 shows the typical debug window display.

Figure 10, Section A lists the program code.

Figure 10, Section B lists the disassembly of the code. For and assembly program, this window should match fairly closely the program code. For a C program, both the C code and generated assembly code will be shown. Also shown in this window is the address and machine code for each.

Figure 10, Section C shows the registers as the program is debugged.

Figure 10, Section D show the contents of the processor memory. This window can be displayed with the menu View >> Memory >> Memory 1-4. The address field sets the address of the memory that is to be displayed.

Step though the code a single line at a time using the "step into" button .

Observe the behavior of each instruction and log the results on the lab worksheet.

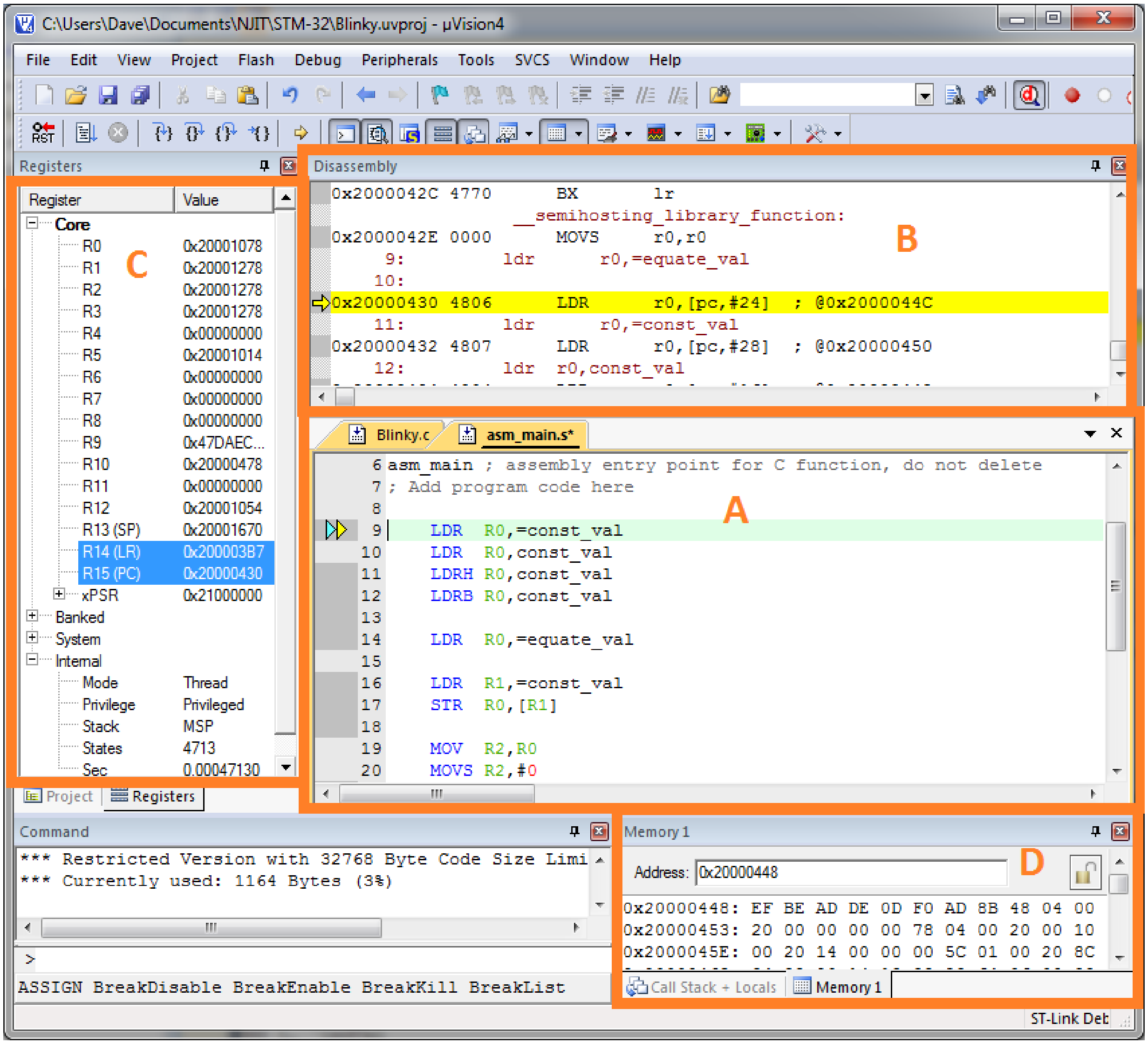


Figure 10 – Debug window

Part 1: This step demonstrates the basics of memory access and moving data within the processor.

Add the code shown in Figure 11 to the asm_main.s file.

The first load moves the 32-bit value (aka word length) from memory at the address const_val to register R1.

Then the address assigned to const_val to register R0. Note the '=' loads the address of the value, not the value its self. Then several methods for loading the value assigned to const_val into a register are demonstrated.

The first a 32-bit load, then a 16-bit load and finally an 8-bit load. Note how the results are different.

The next instruction puts the value associated with the equate equate_val into R0. Note the difference from the constant value move done previously.

The next pair of instructions loads the address for const_val into R1. Then the store instruction (STR) to puts the value in R0 (which is equate_val) into the memory location for const_val. Note that because out program resides

in RAM, the const_val can be changed. If this program was in non-volatile FLASH memory, the value would not change.

The last two move instructions show how to copy values between register and one way a register can be easily cleared.

Variations of these methods are used throughout the course. For example, a very common process is:

- Load the address for a Special Function Register (SFR) to a data register (e.g. LDR R0,=SFR_ADDR)
- Load the value to a second register (e.g. LDR R1,=0x12345678)
- Store the value to the SFR (e.g. STR R1,[R0])

```
equate_val      EQU  0x8BADF00D

     AREA asm_area, CODE, READONLY
     EXPORT  asm_main

asm_main ; assembly entry point for C function, do not delete
; Add program code here

     LDR   R1,const_val        ;load word (32-bit) from memory

     LDR   R0,=const_val       ;load address to R0
     LDR   R1,[R0]             ;2nd load word (32-bit) from
memory
     LDRH  R1,[R0]             ;load half word (16-bit) from mem
     LDRB  R1,[R0]             ;load byte (8-bit) from memory


     LDR   R0,=equate_val      ;load value to R0

     LDR   R1,=const_val       ;load address to R1
     STR   R0,[R1]             ;load value in R0 to memory at R1

     MOV   R2,R0               ;copy R0 to R2
     MOVS  R2,#0               ;clear R2

     B     asm_main

; Put constants here
const_val DCD       0xDEADBEEF

     AREA data_area, DATA, READWRITE
; Put variables here

     END
```

Figure 11 – Code for Lab 1 Part 1

Part 2: This step demonstrates some of the basic arithmetic and logic operations. Notice that instructions ending with an 'S' modify the application program status register (APSR) with the flags (Z,C,N,V).

Modify the code as show in Figure 12.

```
value1    EQU  50
value2    EQU  123
value3    EQU 0xFFFFFFF0

     AREA asm_area, CODE, READONLY
     EXPORT  asm_main

asm_main ; assembly entry point for C function, do not delete
; Add program code here
     MOVS R0,#0              ;clear R0
     LDR  R1,=value1         ;put value1 in R1
     LDR  R2,=value2         ;put value2 in R2
     LDR  R3,=value3         ;put value3 in R3

     MSR  APSR,R0            ;clear flags
     ADDS R2,R1              ;Add values, update APSR
     SUBS R2,R1              ;Subtract values, update APSR
     ADDS R3,R1              ;Add values, update APSR
     SUBS R3,R1              ;Subtract values, update APSR

     MSR  APSR,R0            ;clear flags
     ADD  R3,R1              ;Add values

     CMP  R1,R2              ;compare
     CMP  R2,R1              ;compare
     CMP  R1,R1              ;compare
     CMP  R1,#0x40           ;compare immediate
     CMP  R2,#0x40           ;compare immediate
     CMN  R1,R3              ;compare negative
     CMN  R1,R3              ;compare negative

     B    asm_main

; Put constants here

     AREA data_area, DATA, READWRITE
; Put variables here

     END
```

Figure 12 – Code for Lab 1 Part 2

Part 3: This step demonstrates program flow control operations using unconditional braches.

Modify the code as show in Figure 13.

Each label (i.e. spot1, spot2, spot 3 and spot4) has a memory address associated with the instruction following the label. When the branch instruction (i.e. B spot3) executes occurs, the program counter (R15) is changed to reflect the address associated with the label.

```
     AREA asm_area, CODE, READONLY
     EXPORT  asm_main

asm_main ; assembly entry point for C function, do not delete
; Add program code here

spot1
     B   spot3

spot2
     B   spot4

spot3
     B   spot2

spot4
     B   spot1

; Put constants here

     AREA data_area, DATA, READWRITE
; Put variables here

     END
```

Figure 13 - Code for Lab 1 Part 3

Part 4: This step demonstrates the use of conditional branches.

Modify the code as show in Figure 14.

Unlike the unconditional branch demonstrated in the previous step, the conditional branch uses the state of the processors flags to control the flow of the program. The branch is only taken if the condition for the specific

branch instruction is met. For instance the BNE (branch not equal) will only branch if the Z flag is cleared.

After running the code and recording the results using BNE, rerun the test but replace the BNE with BGE (branch greater than or equal) which branches when N == V.

```
     AREA asm_area, CODE, READONLY
     EXPORT  asm_main

asm_main ; assembly entry point for C function, do not delete
; Add program code here

rst_cnt
     MOVS R0,#3
dec_cnt
     SUBS R0,#1
     BNE   dec_cnt
     B     rst_cnt

; Put constants here

     AREA data_area, DATA, READWRITE
; Put variables here

     END
```

Figure 14 - Code for Lab 1 Part 4

Part 5: This step demonstrates using linked branches for calling subroutines.

Modify the code as show in Figure 15.

The branch and link instructions (BL) are uses to call a subroutine. When the BL is executed, the program counter (PC = R15) is changed to reflect the new address and the address for the next instruction after the BL is put into the link register (LR = R14). When the subroutine completes its execution, the branch and exchange (BX LR) instruction copies the link register into the program counter, returning to the instruction after the original function a call.

```
     AREA asm_area, CODE, READONLY
     EXPORT  asm_main

asm_main ; assembly entry point for C function, do not delete
; Add program code here

loop
     LDR R0,=value1          ;call change_val for value1
     BL   change_value

     LDR R0,=value2          ;call change_val for value2
     BL   change_value

     B    loop               ;do it again

;change_val takes 32-bit value from memory pointed to by R0
;and modifies it by incrementing, then XORing with the
;address, then clearing all byte the lower byte. This is then
;returned back to the address location in memory

change_value
     PUSH {R1,R2}             ;Save R1 and R2 to stack
     LDR R1,[R0]              ;Get value from memory
     ADDS R1,#1               ;Increment
     EORS R1,R0               ;XOR with address
     MOVS R2,#0xFF            ;Set mask
     ANDS R1,R2               ;Mask
     STR R1,[R0]              ;Save value back to memory
     POP {R1,R2}              ;Restore R1 and R2
     BX   LR                  ;Return

; Put constants here

     AREA data_area, DATA, READWRITE
; Put variables here
value1    SPACE    4
value2    SPACE    4

     END
```

Figure 15 - Code for Lab 1 Part 5

3.2 *Lab 2 – General Purpose Inputs and Outputs*

Lab Objectives

- To learn how to configure the MCU internal peripherals
- To learn how to setup and operate the GPIO pins of the MCU

Background

When creating a design using a processor, it is common to need inputs or outputs that operate in a binary (on/off) fashion. These signals are used for monitoring user inputs (switches or pushbuttons), driving indicators (lights or audible), controlling actuators, monitoring/driving discrete control lines from/to other circuits in the design, or for a variety of other purposes. Microcontrollers typical will have pins that can be configured as either outputs that can be driven by or inputs that can be monitored by the processor. These signals are commonly refer to as general purpose inputs and outputs (GPIO's).

The FRDM-KL25Z board provides 53 pins which can be used for assigned peripheral special functions (e.g. UART IO, DAC outputs, ADC inputs, etc.). If a specific peripheral which is tied to a given pin is not used in a design, the pin is available for use as a GPIO. When selecting pins to use as a GPIO, it is important to avoid pins that are assigned to a special function that will also be used in the design. For example, if the DAC was to be used in a design, pins associated with the DAC functions could not be used as GPIO.

For the KL25Z128VLK4 processor used on the KL25Z, section 10.3.1 of the KL25 Sub-Family - Reference Manual lists a table showing how pins are mapped to functions. The table lists up to 8 "ALT" options for any given pin. It can be seen that certain pins can have multiple functions mapped to it. If the function is set to ALT1, it is configured to be a GPIO.

Pins are grouped into 5 "ports" labeled A though E. The architecture allows for ports to have up to 32-bits but some ports have less than 32-bits due to limitations on the number of pins available in the device package. Pins are numbered 0 to 31. Figure 16 shows how the port pins are mapped to the headers on the FRDM-KL52Z. For example PTA1 is Port A, Pin 1.

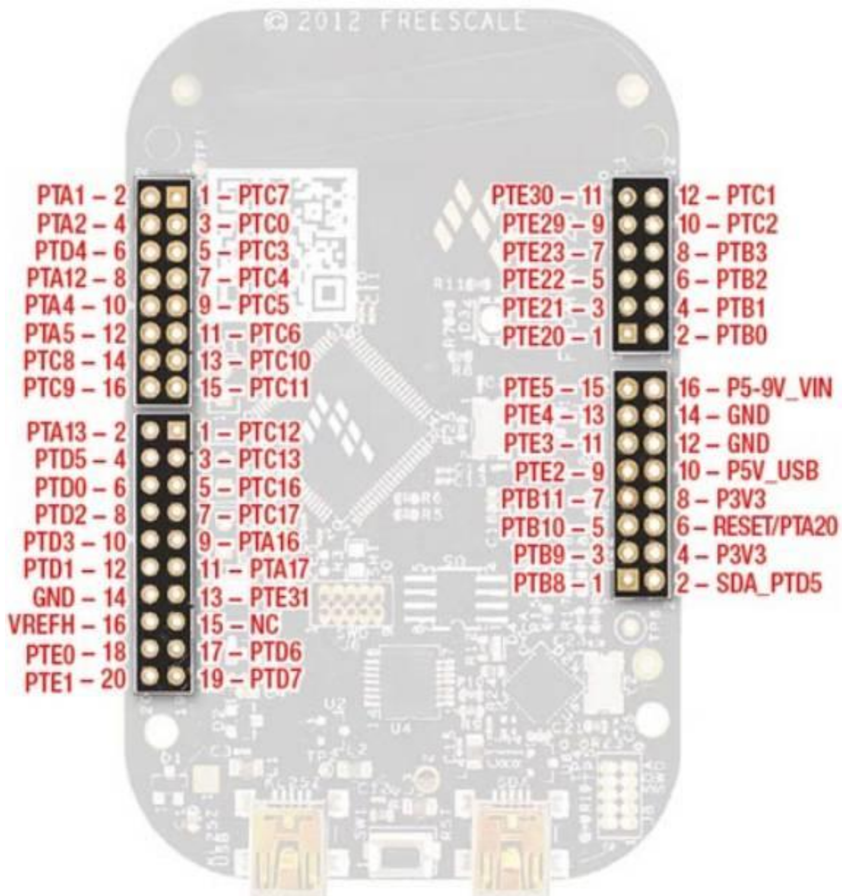


Figure 16 - Header Pin Assignments
(from the FRDM-KL25Z User Manual)

Once a GPIO is selected, several registers must be properly configured in the processor to allow it to be used as an input or output.

1. Each Port has a separate clock gate that must be enabled if IO associated with the Port will be used. The SIM_SCGC5 register contains the controls for the Port clock gates. See section 12.2.9 of the KL25 Sub-Family - Reference Manual for details on how to enable the clock gates using the SIM_SCG5 register.

2. Each pin has a pin control register, PORTx_PCRn where x is the the Port and n is the Pin, that is used to configure the pin behavior. There are setting fields for the ALT option, interrupt operation, drive strength, slew rate and pull-up/down resistor configuration. See section 11.5.1 of the KL25 Sub-Family - Reference Manual for details on how to set the PORTx_PCRn registers.

For this lab for the pins used as GPIO's, the ALT option will be set to ALT1, interrupts will be disabled, the drive strengths should be set to normal, the slew rate should be set to slow and the passive filter should be disabled. For outputs, the pull resistors will be disabled. For inputs, the pull resistors should be enabled and set appropriately based on the geometry if the circuit driving the input. See Figure 17.

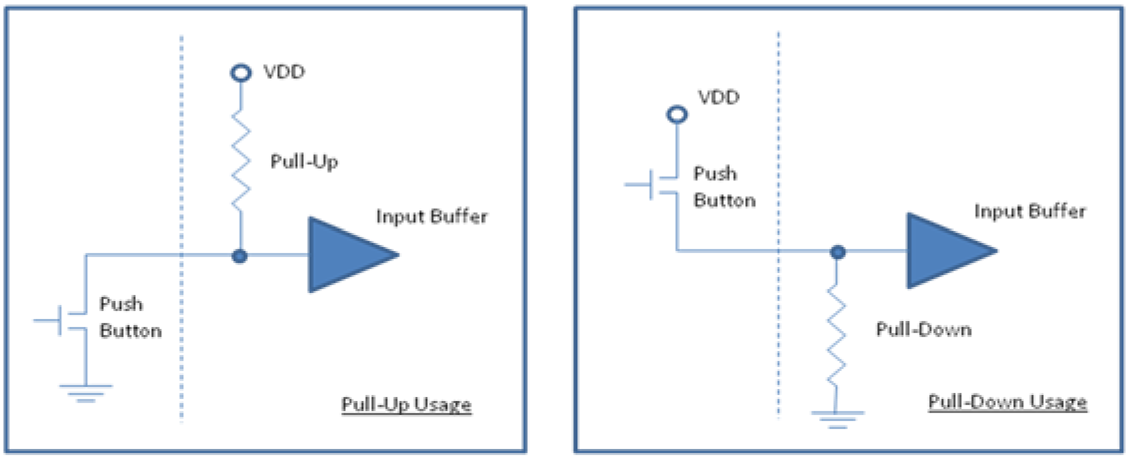


Figure 17 - Pull Resistor Configurations

3. Each Port has a data direction register, GPIOx_PDDR where x it the port, which configures whether a pin will be an input or output. All 32 bits for the port are grouped in the single register. See section 41.2.6 of the KL25 Sub-Family - Reference Manual for details on how to set the GPIOx_PDDR register.

After configuration, if the GPIO has been configured as an input, the state of the pin can be read using the GPIOx_PDIR register. All 32 bits for the port are grouped in the single register. See section 41.2.5 of the KL25 Sub-Family - Reference Manual for details on how to use the GPIOx_PDIR registers.

Otherwise, if the GPIO has been configured as an output, the state of pin can be controlled with several registers. All 32 bits for the port are grouped in each single register. Writing a 1 or 0 to a given bit in GPIOx_PDOR sets or clears the output based on the value in each bit. Note that using this register requires setting all the pins on the port simultaneously as each of the 32 bits must have a value of 1 or 0. Using the GPIOx_PSOR and GPIOx_PCOR registers allows pins to be respectively set or cleared individually by writing a 1 to the desired bit locations. Using the GPIOx_PTOR register toggles a given pin writing a 1 to the desired bit location. See sections 41.2.1 thru 41.2.4 of the KL25 Sub-Family - Reference Manual for details on how to use these registers.

There is an LED on the FRDM-KL25Z is a tri-color red/green/blue device. The common anode is tied to VDD. The three cathodes are tied through resistors to GPIO's as listed in Table 1. Because the LED is wired with the common anode to VDD, the GPIO's must be driven low to run on the LED color and driven high to turn off the LED.

Table 1- KL25Z LED GPIO Assignments

LED Color	GPIO
RED	PTB18
GREEN	PTB19
BLUE	PTD1

Required Equipment and Parts

- Solderless breadboard
- Pushbutton
- Jumper wires

Procedure

Part1: *GPIO's as Outputs*

The aim of this part of the lab is to learn how to control an LED. Code is given that blinks the LED red. You will modify it to: explore the various control options, change the LED color, change the blink rate and change the intensity.

1. Load the code shown in Figure 18

```
SIM_SCGC5   EQU  0x40048038          ;SIM_SCGC5 address

PORTB_PCR18 EQU  0x4004A000 + 4 * 18 ;PORTB_PCR18 address
PORTB_PCR19 EQU  0x4004A000 + 4 * 19 ;PORTB_PCR18 address
PORTD_PCR1  EQU  0x4004C000 + 4 * 1  ;PORTD_PCR1 address
PORTD_PCR3  EQU  0x4004C000 + 4 * 3  ;PORTD_PCR3 address

GPIOB_PSOR  EQU  0x400FF044          ;GPIOB_PSOR address
GPIOB_PCOR  EQU  0x400FF048          ;GPIOB_PCOR address
GPIOB_PTOR  EQU  0x400FF04C          ;GPIOB_PTOR address
GPIOB_PDDR  EQU  0x400FF054          ;GPIOB_PDDR address

GPIOD_PSOR  EQU  0x400FF0C4          ;GPIOD_PDIR address
GPIOD_PCOR  EQU  0x400FF0C8          ;GPIOD_PDIR address
GPIOD_PDIR  EQU  0x400FF0D0          ;GPIOD_PDIR address
GPIOD_PDDR  EQU  0x400FF0D4          ;GPIOD_PDDR address

BLUE_MASK   EQU  0x00000002
RED_MASK    EQU  0x00040000
GREEN_MASK  EQU  0x00080000

BUTTON_MASK EQU  0X00000008

DELAY_CNT   EQU  0X00800000

     AREA    asm_area, CODE, READONLY
     EXPORT  asm_main

asm_main  ;assembly entry point for C function, do not delete
; Add program code here

     BL init_gpio
loop
     BL  redon
     LDR R0, =DELAY_CNT
     BL  delay
     BL  redoff
     LDR R0, =DELAY_CNT
     BL  delay
     B   loop

redon
     LDR R0,=GPIOB_PCOR  ;Load address of GPIOB_PCOR to R0
     LDR R1,=RED_MASK    ;Load value to R1
     STR R1,[R0]         ;Put value into GPIOB_PCOR
     BX LR
```

```
redoff
     LDR R0,=GPIOB_PSOR  ;Load address of GPIOB_PSOR to R0
     LDR R1,=RED_MASK    ;Load value to R1
     STR R1,[R0]         ;Put value into GPIOB_PSOR
     BX LR

redtoggle
     LDR R0,=GPIOB_PTOR  ;Load address of GPIOD_PTOR to R0
     LDR R1,=RED_MASK    ;Load value to R1
     STR R1,[R0]         ;Put value into GPIOD_PTOR
     BX LR

delay
     SUBS R0, #1
     BNE  delay
     BX LR

init_gpio
     ; Turns on clocks for all ports
     LDR R0,=SIM_SCGC5    ;Load address of SIM_SCGC5 to R0
     LDR R1,[R0]          ;Get original value of SIM_SCGC5
     LDR R2,=0x00003E00   ;Load mask for bits to set to R2
     ORRS R1,R2           ;Set bits with OR of orig val and
mask
     STR R1,[R0]          ;Put new value back into SIM_SCGC5

     ; Setup PORTB Pin 18 to be output
     LDR R0,=PORTB_PCR18   ;Load address of PORTB_PCR18 to R0
     LDR R1,=0x00000100    ;Load new value to R1
     STR R1,[R0]           ;Put value into PORTB_PCR18

     ; Setup PORTB Pin 19 to be output
     LDR R0,=PORTB_PCR19   ;Load address of PORTB_PCR19 to R0
     LDR R1,=0x00000100    ;Load new value to R1
     STR R1,[R0]           ;Put value into PORTB_PCR19

     ; Setup PORTD Pin 1 to be output
     LDR R0,=PORTD_PCR1     ;Load address of PORTD_PCR1 to R0
     LDR R1,=0x00000100     ;Load new value to R1
     STR R1,[R0]            ;Put value into PORTD_PCR1

     ; Setup PORTD Pin 3 to be input
     LDR R0,=PORTD_PCR3     ;Load address of PORTD_PCR3 to R0
     LDR R1,=0x00000103     ;Load new value to R1
     STR R1,[R0]            ;Put value into PORTD_PCR3

     ;Setup R2 for mask for Green and Red LED control lines
```

```
     LDR R2,= GREEN_MASK :OR: RED_MASK

     ;Set bits in DDR register to enable outputs to drive LED's
     LDR R0,=GPIOB_PDDR   ;Load address of GPIOB_PDDR to R0
     LDR R1,[R0]          ;Get original value of GPIOB_PDDR
     ORRS R1,R2           ;Set bits with OR of orig val and
mask
     STR R1,[R0]          ;Put new value back into GPIOB_PDDR

     ;Turn off LED's by setting control lines
     LDR R0,=GPIOB_PSOR   ;Load address of GPIOB_PSOR to R0
     LDR R1,[R0]          ;Get original value of GPIOB_PSOR
     ORRS R1,R2           ;Set bits with OR of orig val and
mask
     STR R1,[R0]          ;Put new value back into GPIOB_PSOR

     ;Setup R2 for mask for Blue LED control line
     LDR R2,=BLUE_MASK

     ;Setup R3 for mask of pushbutton input
     LDR R3,=:NOT: BUTTON_MASK

     ;Set bits in DDR register to enable outputs to drive LED's
     ; and ensure inputs are clear
     LDR R0,=GPIOD_PDDR   ;Load address of GPIOD_PDDR to R0
     LDR R1,[R0]       ;Get original value of GPIOD_PDDR into
R1
     ORRS R1,R2        ;OR original value with mask to set bits
     ANDS R1,R3        ;AND value with mask to clear bits
     STR R1,[R0]       ;Put new value back into GPIOD_PDDR

     ;Turn off LED's by setting control lines
     LDR R0,=GPIOD_PSOR    ;Load address of GPIOD_PSOR to R0
     LDR R1,[R0]       ;Get original value of GPIOD_PSOR into
R1
     ORRS R1,R2        ;OR original value with mask to set bits
     STR R1,[R0]       ;Put new value back into GPIOD_PSOR

     BX LR

     END
```

Figure 18 - Code for GPIO's as outputs

The asm_main routine starts with a call to the gpio_init subroutine. This code initializes the registers needed to set the three GPIO lines connected to the on-board tri-color LED to be used as outputs. It also sets up GPIO PTD3 for use as an input.

The code in the gpio_init routine used two methods to set registers. In the first method for loading the value to SIM_SCGC5, the code reads the original value of the register, then OR's the bits that need to be set with the original value, then writes the new value back to the register. This is done to maintain any settings that were previously set. Note that the value that is being written will turn on all the Port clocks. This is not ideal for a design with an objective to minimize power consumption but it is fine for our needs.

The second register setting method used for loading the value into PORTB_PCR18, which directly writes the new value into the register. This is ok here because the whole register only impacts the behavior of the concerned pin.

The asm_main loop blinks the LED red. The loop call the redon routine to turn on LED by clearing the control line. The delay waits a bit of time. Then the redoff routine sets the control line, turning off the LED. Then the delay is called again. The program then loops.

Remove the second 'BL delay' call. **What happens and why?**

Leaving the previous change, remove the 'BL redoff' call. **What happens and why?**

Leaving the previous changes, replace the 'BL redon' call with 'BL redtoggle'. **Now what happens and why?**

2. Add code to turn the LED green and blue (Hint copy and modify the redon, redoff and redtoggle routines). Change the asm_main routine to cycle the LED the following colors. Make the delay between each color longer so they can be easily seen.

    Red

    Green

    Blue

    Yellow (Red and Green)

    Cyan (Green and Blue)

    Magenta (Red and Blue)

    White (Red, Green and Blue)

Record the values of the GPIOD_PDOR and GPIOB_PDOR for each color. To do so, add a breakpoint for each color step and use the system view to see the register value. The system registers can be viewed by selecting Peripherals >> System Viewer >>> {Register Group}. Register groups GPIOB and GPIOD should be selected. The values of the registers should be observed as the program executes as show in Figure 19.

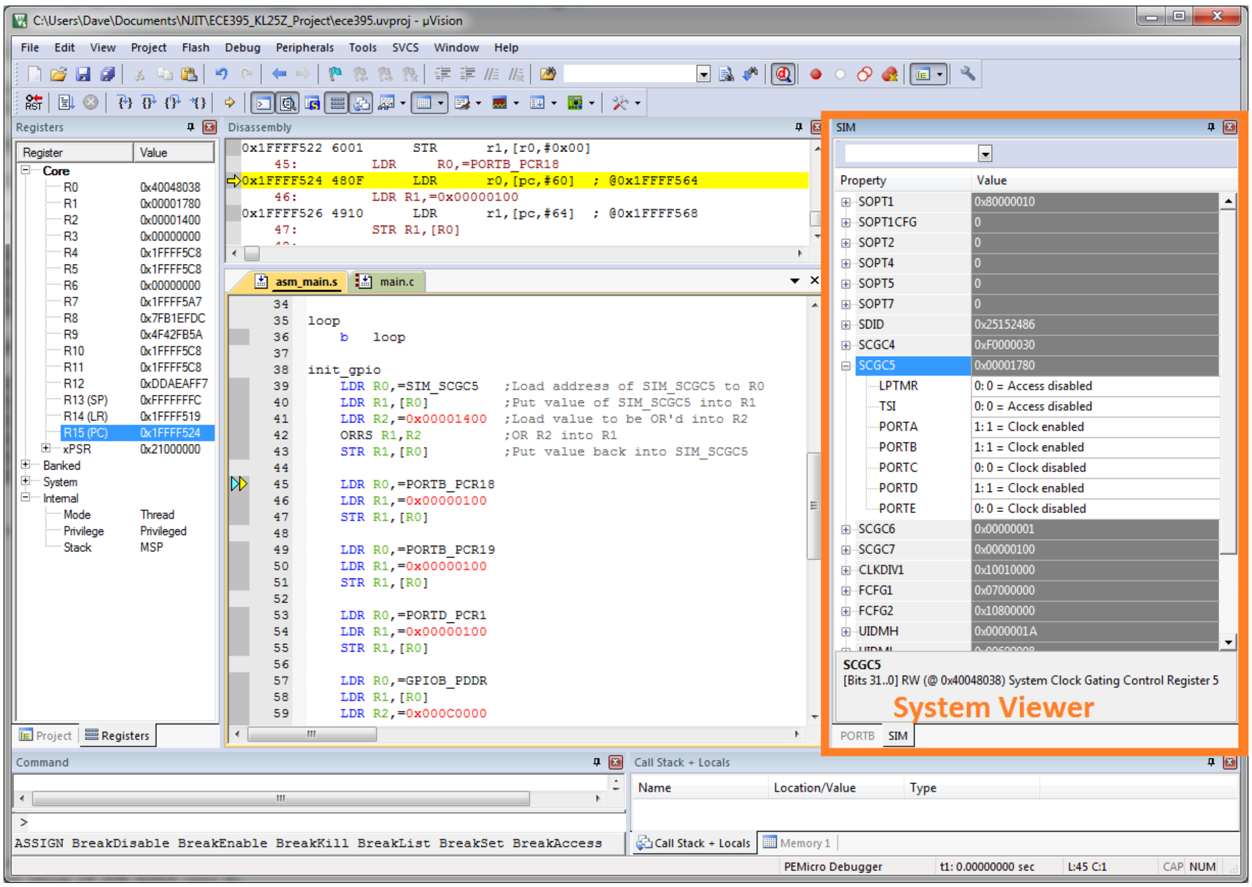


Figure 19 - Observing Registers with System Viewer

3. Return the code to its original state as shown in Figure 18. The LED blink rate is controlled by the value in the equate DELAY_CNT which is used to set how many times the delay sub-routine loops. Run the original code and **measure the blink rate** by counting the number of times the LED blink in a given interval (Hint: Use the stopwatch feature on a phone to time 30 or 60 seconds).

   Change increase the DELAY_CNT by 50%. **Measure the blink rate.**

Change increase the DELAY_CNT by another 50%. **Measure the blink rate.**

4. The brightness of a LED can be changed by very quickly blinking with a variable duty cycle. For example, 50% on and 50% off will appear as half a bright. 25% on and 75% off will appear has a quarter as bright. Modify the code to use a separate delay count for the off and on cycle. Make the delay much smaller (about 100x smaller) but equal in size for off and on. Test the code and **observe the LED brightness**.

   Make the off duration 25% of the on duration (keep the total ratio the same). **Observe the LED brightness**.

   Now make the on duration 25% of the off duration (keep the total ratio the same). **Observe the LED brightness**.

Part 2: *GPIO's as Inputs*

The aim of this part of the lab is to wire a pushbutton to a GPIO on the FRDM-KL25Z, then to run software to change the color of the on-board LED when the button is pushed.

1. Replace the loop in the asm_main function from part 1 with the code listed in Figure 20. Connect a normally open pushbutton between ground and pin D3 on the KL25Z. Run the code and observe the behavior when the button is pressed.

   With the button not pressed, record the value of GPIOD_PDIR after it is read into R1. Then record the flags after the TSTS command is executed.

   With the button pressed, record the value of GPIOD_PDIR after it is read into R1. Then record the flags after the TSTS command is executed.

```
     BL init_gpio
loop
     LDR R0,=GPIOD_PDIR     ;Load address of GPIOD_PDIR
     LDR R1,[R0]            ;Get value of GPIOD_PDIR
     LDR R2,=BUTTON_MASK    ;Load mask for Pin D3
     TSTS R1,R2             ;TST mask vs inputs
     BNE button_open        ;If PB open branch to turn off
     BL  redon              ;Else turn on
     B   loop               ;Do it again
button_open
     BL redoff              ;Turn it off
     B  loop                ;Do it again
```

Figure 20 – Code for GPIO's as inputs

3.3 *Lab 3 – Annunciator (GPIO Application)*

Lab Objectives

- To apply knowledge on the use of GPIO's gained in lab 2 to solve a design problem

Problem

A maple syrup factory in Vermont has a problem. They have a holding tank that stores their product that overflows from time-to-time. When this happens, an operator in a remote monitoring room is sent to clean up the mess. They have asked you group to implement an "Annunciator" system to monitor the holding tank and report its status to the operator in the monitoring room. The system has two objectives, to notify the operator when the tank is near full (so they can manually turn off the fill valve), then to notify the operator when the tank has over flown (so they can be sent to clean it up).

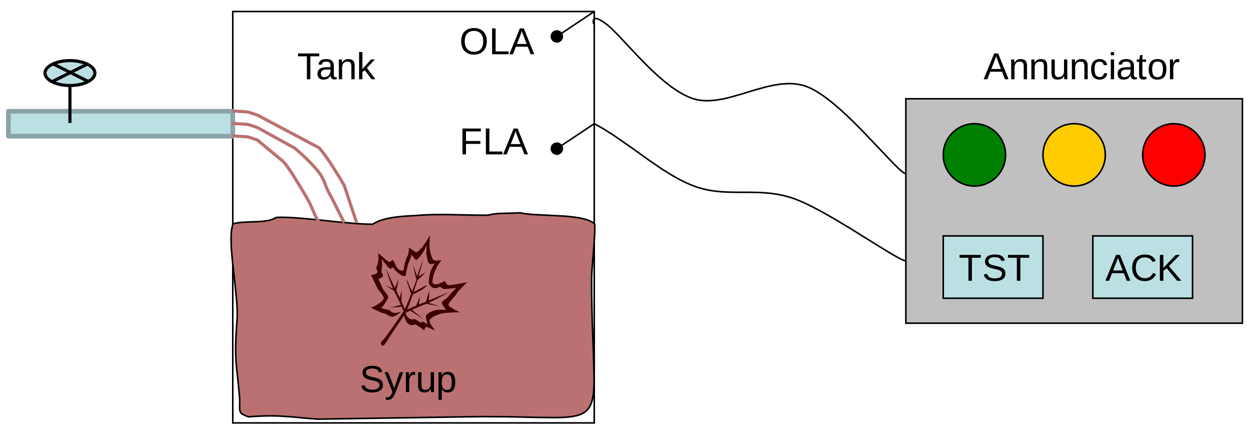


Figure 21 - Annunciator System

There are 4 input to the system. There are two level switches in the tank, full level alarm (FLA) and overflow level alarm (OLA). On the Annunciator box in the control room, there are two momentary push buttons, acknowledge (ACK) and test (TST).

On the Annunciator box, there are 3 outputs from the system, a green ok indicator, a yellow full indicator and a red overflow indicator.

The system has 6 states as shown in Figure 22.

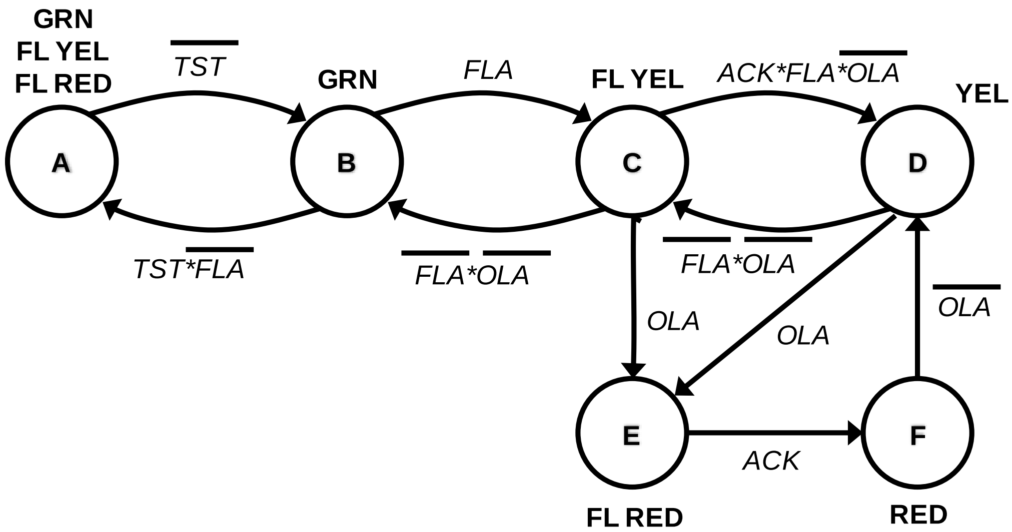


Figure 22 - Annunciator State Diagram

Required Equipment and Parts

- Solderless breadboard
- 2 Pushbuttons
- 1 DIP switch
- 1 Red LED
- 1 Yellow LED
- 1 Green LED
- 3 220 ohm resistors
- Jumper wires

Procedure

Use the knowledge gained in lab 2 to configure the GPIO's to monitor the four inputs and control the three outputs. Write software to implement the state machine for the annunciator system.

1. Select the GPIO's to be used for the inputs. Avoid the GPIO's assigned to the on-board LED. On the solderless breadboard, wire the 2 pushbuttons and 2 of the DIP stitches to the 4 GPIO's selected as the

inputs in a pull-up resistor configuration with one side of the switch to the GPIO and the other side to ground.

2. Select the GPIO’s to be used as the outputs. Avoid the GPIO’s assigned to the on-board LED. On the solderless breadboard, wire the 3 outputs to the anode’s of the red yellow and green LED’s. Tie the cathodes of the LED to ground though 200 ohm resistors.

3. Modify the gpio_init routine from lab 2 to setup the GPIO inputs and outputs appropriately.

4. Add code to handle the states of the Annunciator state machine.

5. Test and debug the code.

2. After fully testing the program, demonstrate it to the course instructor for credit.

3.4 *Lab 4 – UART Serial Port*

Lab Objectives

- To learn how to setup and operate the MCU serial ports
- To create portable functions for serial port initialization and write and read operations
- To learn how to use an oscilloscope to observe a serial waveform
- One lab report is required from group

Background

Serial communications is a fundamental principal for microprocessor systems. In serial communications, data is transferred sequentially bit-by-bit along a channel in contrast to parallel communications where multiple bits are sent simultaneously over multiple channels. In modern digital systems, there are various protocols which employ serial transmission techniques that are aimed at a variety of applications. Some common examples are: USB (Universal Serial Bus) which is commonly used to interface peripherals to computers; SATA (Serial ATA), which is used to interface storage devices in computers; and Ethernet, which is used for computer networks. Other examples of serial buses are I2C and SPI (Serial Peripheral Interface) buses which are commonly found in embedded processor systems as interfaces busses for memories, DAC's and ADC, etc and CAN Bus (Controller Area Network) which is used to interface various systems in vehicles.

One of the simplest implementations of serial communications is the asynchronous serial port. Historically these were common on personal computers for uses such as interfacing to external modems, peripherals such as mice and computer terminals. On PC's, these serial ports used RS-232 complaint signaling and DB-25 or DE-9 connectors. RS-232 specifies the electrical characteristics of the signals. In the last decade, serial ports on PC's have become rarer features as USB has replaced most of the consumer applications that were previously handled by serial ports.

In embedded systems and industrial controls, asynchronous serial communications is still very common and useful. In one common embedded application, asynchronous serial ports are used for debug console interfaces. Most microcontrollers feature UART (Universal Asynchronous Receiver Transmitters) peripherals internal to the microcontroller. A common implementation would be to connect the microcontroller UART to an RS-232

converter IC (integrated circuit) on the embedded system which would then interface to the RS-232 serial port on a PC.

In newer PC's which do not have built in RS-232 serial ports, a USB-to-RS-232 converter would be used. These converters usually have a DE-9 connector, RS-232 converter IC and a serial-to-USB converter IC with a USB cable to interface to the PC. These serial-to-USB converter IC's are available from a variety of manufactures (Prolific and FTDI are very common). The use of these IC's required a driver to be installed on the PC but typically do not required any custom firmware to use the IC.

In newer embedded systems (in the Arduino for instance), the RS-232 interface is completely removed and the serial-to-USB IC is directly put on the embedded board. This allows the embedded system to directly connect to a PC without the use of a USB-to-RS-232 converter.

The KL25Z uses a similar approach but uses a secondary ARM processor as the serial-to-USB interface instead of the serial-to-USB IC. This secondary ARM processor also serves as the programming and debug interface that is used to load and test code on the main ARM processor. The functions performed by the secondary ARM processor are called OpenSDA.

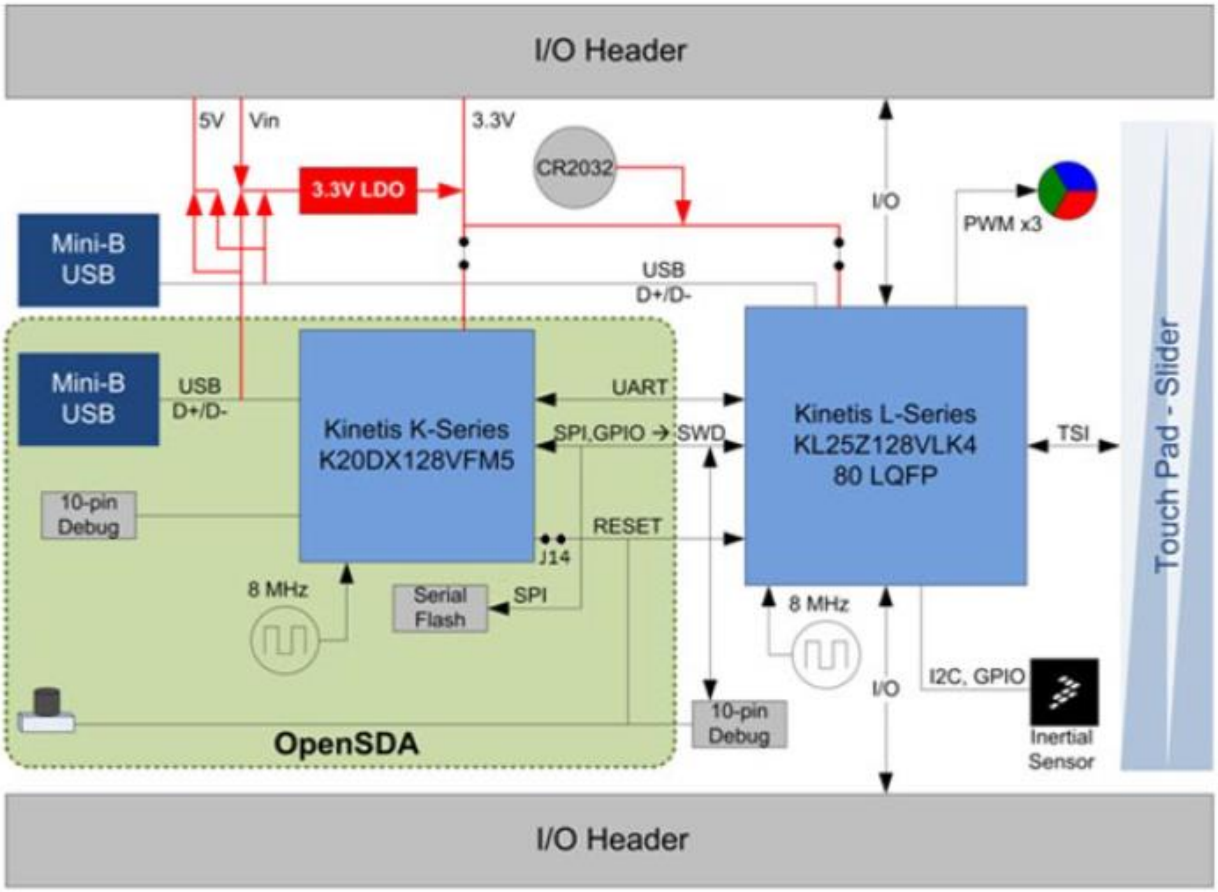


Figure 23 - KL25Z OpenSDA Interface

Then using serial communications, both the transmitter and receiver must use a similar clock rate that are synchronized in some fashion so the receiver can sample and decide if a bit is high or low. In some serial communications schemes, a clock is sent in parallel with the data to align the transmitter and receiver. In other schemes, the receiver does clock recovery, where it generates a local clock that is aligned to the transitions in the data pattern to provide a sampling clock. The UART uses asynchronous sampling to align the receiver to the transmit stream.

For asynchronous sampling to work, both the transmitter and receiver must be pre-configured to share the same data rate and format. When no data is being sent, the transmitter idles at a fixed level, high in the case of traditional UART's . When a data byte is to be transmitted, the transmitter starts will a "start" bit, which is always a low. The transmitter then follows with the data bits, which are usually sent LSB (least significant bit) first, high is a "1" and low is a "0". The transmitter closes the transmission with a "stop" bit which is always a high.

When the receiver sees the transition from high (idle) to low (start bit), it knows a data byte is coming. It starts sampling the subsequent bits roughly 1.5 bit periods after the beginning of the start bit. It samples at the bit period for the number of data bits it has been configured for. The presence of the stop bit forces the line to go high so the receiver can observe the next high to low transition. Because the receiver re-synchronizes its sampling after each transmitted byte, differences of up to about +/-5% are possible in the transmit and receive clocks.

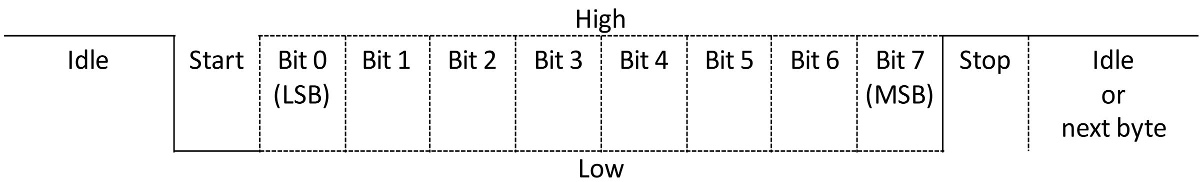


Figure 24 - Asynchronous Serial Transmission

The processor on the KL25Z has 3 UART's. UART0 is wired on KL25Z to the Open SDA interface hence it will be used in this lab. The OpenSDA serial connection is wired to pins PTA1 (RX) and PTA2 (TX). To use a UART0, several parts of the processor must configured. Code is provided that does the following.

1. The UART0 clock source select bits (UART0SRC) must be set in the SIM_SOPT2 register. In the code provided, these bits are set to 01b = MCGFLLCLK clock or MCGPLLCLK/2 clock. This means either the MCGFLLCLK (96MHz on the KL25Z) or MCGPLLCLK/2 (48MHz) will be the clock that drives UART0. The next setting selects which one is used.
2. The PLL/FLL clock select (PLLFLLSEL) must be set in the SIM_SOPT2 register. In the code provided, this bit is set to 1b = MCGPLLCLK clock with fixed divide by two. This means the UART0 clock will be 48MHz.
3. The UART0 Clock Gate Control (UART0) in the SIM_SCGC4 register must be enabled (set to 1).
4. The Port A Clock Gate Control (PORTA) in the SIM_SCGC5 must be enabled (set to 1). This is because the UART0 will use IO on port A.
5. The Pin Mux Control (MUX) bits of PORTA_PCR1 and PORTA_PCR2 must be set to 010b = Alternative 2. Alternate 2 on these pins is UART0_RXfor PTA1 and UART0_TX for PTA2.

6. The Over Sampling Ratio (OSR) size must be set in UART0_C4. In the code provided, it has been set to x8. This factors into the baud rate calculations.
7. The Baud Rate Modulo Divisor (SBR) must be set in the UART0_BDH and UART0_BDL registers based on the desired baud rate and the clock settings. The SBR is a 13 bit long field split between UART0_BDH and UART0_BDL. The lower 8 bits (SBR[7:0]) are in UART0_BDL and the upper 5 bits (SBR[12:9]) are the lowest bits in UART0_BDH.

   SBR = clock_rate/(OSR * baud_rate)
   clock_rate = 48MHz (based on the settings for UART0SRC and PLLFLLSEL)
   OSR = 8

   For baud_rate = 9600 bps
   SBR = 48000000/(8 * 9600) = 625 = 0x271
   UART0_BDH = 0x02
   UART0_BDL = 0x71

8. The Transmitter Enable (TE) and receiver Enable (RE) bits in UART0_C2 must be set to 1 to enable the transmitter and receiver.

After the UART is configured, the UART can transmit and receive.

Data can be transmitted by writing to the UART Data Register (UART0_D). Data should only be written to the UART if the transmitter is not busy (e.g. still sending a byte). The status of the transmitter can be monitored with the Transmit Data Register Empty Flag (TDRE) in UART Status Register 1 (UART0_S1). When TDRE is 1, the transmitter can be written to.

Received data can be read from the UART Data Register (UART0_D). Data is available when the Receive Data Register Full Flag (RDRF) in UART0_S1 is 1. Data should only be read from UART0_D after verifying RDRF is 1.

The UART can experience errors which will lock up the receiver until they are cleared. They are indicated by the OR (Receiver Overrun Flag), NF (Noise Flag), FE (framing Error Flag) and PE (Parity Error Flag) in the UART0_S1 register. If these bits are set, they must be cleared by writing a 1 to the corresponding bit field before a character can be successfully read from the UART.

Data transmitted and received on serial ports is often formatted as ASCII (American Standard Code for Information Interchange) characters. ASCII provides a standard way to translate hex bytes to characters (letters, digits, punctuation, etc). When using a terminal emulator to send text, the data is usually ASCII.

Figure 25 shows a summary of the registers for UART0. Note that the fields that are described are in bold. Also note all fields are 8 bits and length and should be accessed with 8 bit instructions (LDRB and STRB).

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
UART0_C1	LOOPS	DOZEEN	RSRC	M	WAKE	ILT	PE	PT
UART0_C2	TIE	TCIE	RIE	ILIE	**TE**	**RE**	RWU	SBK
UART0_C3	R8T9	R9T8	TXDIR	TXINV	ORIE	NEIE	FEIE	PEIE
UART0_C4	MAEN1	MAEN2	M10	**OSR**				
UART0_C5	TDMAE	0	RDMAE	0	0	0	BOTHEDGE	RESYNCDIS
UART0_S1	**TDRE**	TC	**RDRF**	IDLE	OR	NF	FE	PF
UART0_S2	LBKDIF	RXEDGIF	MSBF	RXINV	RWUID	BRK13	LBKDE	RAF
UART0_BDH	LBKDIE	RXEDGIE	SBNS	**SBR[12:8]**				
UART0_BDL	**SBR[7:0]**							
UART0_D	**Data[7:0]**							

Figure 25 - UART0 Registers

(see section 39 of KL25 Sub-Family Reference Manual for details)

Required Equipment and Parts

- Tektronix DPO2012B Oscilloscope (in lab)
- Oscilloscope Probe (from stockroom)
- Tektronix DPO2COMP Computer Serial Module (from stockroom)
- Jumper wires

Procedure

3.4.1 Lab 4, Part 1: Reading and writing characters for the UART

The objective of this part is to develop a program to initialize the UART and enter a loop that reads a character from the UART and then writes it back to the UART. This is known as "echoing" back. A terminal emulator running on the PC will be used to send characters to the board and display what the board sends back.

A starting point for the code for this program is given in Figure 26. This code needs to have the char_out and char_in functions completed.

```
SIM_SOPT2       EQU       0x40048004
SIM_SCGC4       EQU       0x40048034
SIM_SCGC5       EQU       0x40048038

PORTA_PCR1      EQU       0x40049000 + 4 * 1
PORTA_PCR2      EQU       0x40049000 + 4 * 2

PORTE_PCR20     EQU       0x4004D000 + 4 * 20
PORTE_PCR21     EQU       0x4004D000 + 4 * 21

UART0_BDH       EQU       0x4006A000
UART0_BDL       EQU       0x4006A001
UART0_C1        EQU       0x4006A002
UART0_C2        EQU       0x4006A003
UART0_S1        EQU       0x4006A004
UART0_S2        EQU       0x4006A005
UART0_C3        EQU       0x4006A006
UART0_D         EQU       0x4006A007
UART0_MA1       EQU       0x4006A008
UART0_MA2       EQU       0x4006A009
UART0_C4        EQU       0x4006A00A
UART0_C5        EQU       0x4006A00B

     AREA       asm_area, CODE, READONLY
     EXPORT     asm_main
     EXPORT     UART0Init
     EXPORT     char_in
     EXPORT     char_out

asm_main FUNCTION   ;assembly entry point for C function
; Add program code here

     BL   UART0Init
loop
     BL   char_in   ; read char from UART, char in R0
```

```
     BL   char_out  ; send char in R0 to UART
     b    loop
     ENDFUNC



; When char_out is called, R0 contains
; the char to be sent out the UART
char_out FUNCTION

     BX LR
     ENDFUNC



; When char_in returns, R0 contains
; the char that was received
char_in FUNCTION

     BX LR
     ENDFUNC

UART0Init FUNCTION
; SIM_SOPT2[UART0SRC] = 01b (MCGFLLCLK or MCGPLLCLK/2 clock)
; and SIM_SOPT2[PLLFLLSEL] = 1b for MGCPLLCLK/2
     LDR  R0,=SIM_SOPT2  ;Load address of SIM_SOPT2 to R0
     LDR R1,[R0]         ;Put present value of SIM_SOPT2 into
R1
     LDR R2,=0xF3FEFFFF  ;Load bits to clear
     ANDS R1,R2          ;AND values to clear bits
     LDR R2,=0x04010000  ;Load bits to set
     ORRS R1,R2          ;OR values to set bits
     STR R1,[R0]         ;Put value back into SIM_SOPT2

; SIM_SCGC4[UART0] = 1
     LDR  R0,=SIM_SCGC4
     LDR R1,[R0]
     LDR R2,=0x00000400
     ORRS R1,R2
     STR R1,[R0]

; SIM_SCGC5[PORTE thru A] = 1, turn on clock for all ports
     LDR R0,=SIM_SCGC5   ;Load address of SIM_SCGC5 to R0
     LDR R1,[R0]         ;Put value of SIM_SCGC5 into R1
     LDR R2,=0x00003E00  ;Load value to turn on all port
                         ;clocks into R2
     ORRS R1,R2          ;OR R2 into R1
```

```
    STR R1,[R0]         ;Put value back into SIM_SCGC5

; PORTA_PCR1 , Clear ISF and set MUX = 2
    LDR  R0,=PORTA_PCR1
    LDR R1,[R0]
    LDR R2,=0x01000200
    ORRS R1,R2
    STR R1,[R0]

; PORTA_PCR2 , Clear ISF and set MUX = 2
    LDR  R0,=PORTA_PCR2
    LDR R1,[R0]
    LDR R2,=0x01000200
    ORRS R1,R2
    STR R1,[R0]

; UART0_C4[OSR]= 0x07 (for osr = x8)
    LDR  R0,=UART0_C4
    MOVS R1,#0x07
    STRB R1,[R0]

; 9600 baud
; uart0_baud_clk = MGCPLLCLK/2 = 96MHz/2 = 48MHz
; SBR = uart0_baud_clk/(baud*osr)
; SBR = 48MHz/(9600 * 8)
; SBR = 625 = (0x0271)

; UART0_BDH = 0x02
    LDR  R0,=UART0_BDH
    MOVS R1,#0x02
    STRB R1,[R0]

; UART0_BDL = 0x71
    LDR  R0,=UART0_BDL
    MOVS R1,#0x71
    STRB R1,[R0]

; UART0_C2 = 0x02 (TE and RE = 1)
    LDR  R0,=UART0_C2
    MOVS R1,#0x0C
    STRB R1,[R0]

    BX LR
    ENDFUNC

; Put constants here
```

```
     AREA data_area, DATA, READWRITE
; Put variables here

     END
```

Figure 26 - Starting Code to Serial Programs

The aim is for the char_out, char_in and UART0Init functions to be callable from the main C program for use in subsequent experiments. The EXPORT and FUCNCTION, and ENDFUN statements make these runtimes appear as functions in C.

When C calls a function, the first four parameters are passed to the function in R0, R1, R2 and R3. So if the template for char_out in C is:

extern void char_out(unsigned char);

Then the character to be sent out the UART will be the first parameter in R0.

Likewise, when a function return a value, the value that will be returned is in R0. So if the template for char_in in C is:

extern unsigned char char_in(void);

Then the character that the UART received should be returned in R0.

Note that char_in and char_out are defined as "*unsigned char*", this means that they will be only 8 bits in length which is the length the UART uses.

When writing the char_out and char_in routines, only use registers R0 to R3 and R12 as using any other registers requires the values to be saved and restored in the function.

The char_out and char_in routines should implement the logic shown in Figure 27. Note that the char_in routine needs to verify that the OR, NF, FE and PF bits are not set. If they are set, they can be cleared by writing a one to the specific bit location or by simply writing 0x0F to clear any that are set.

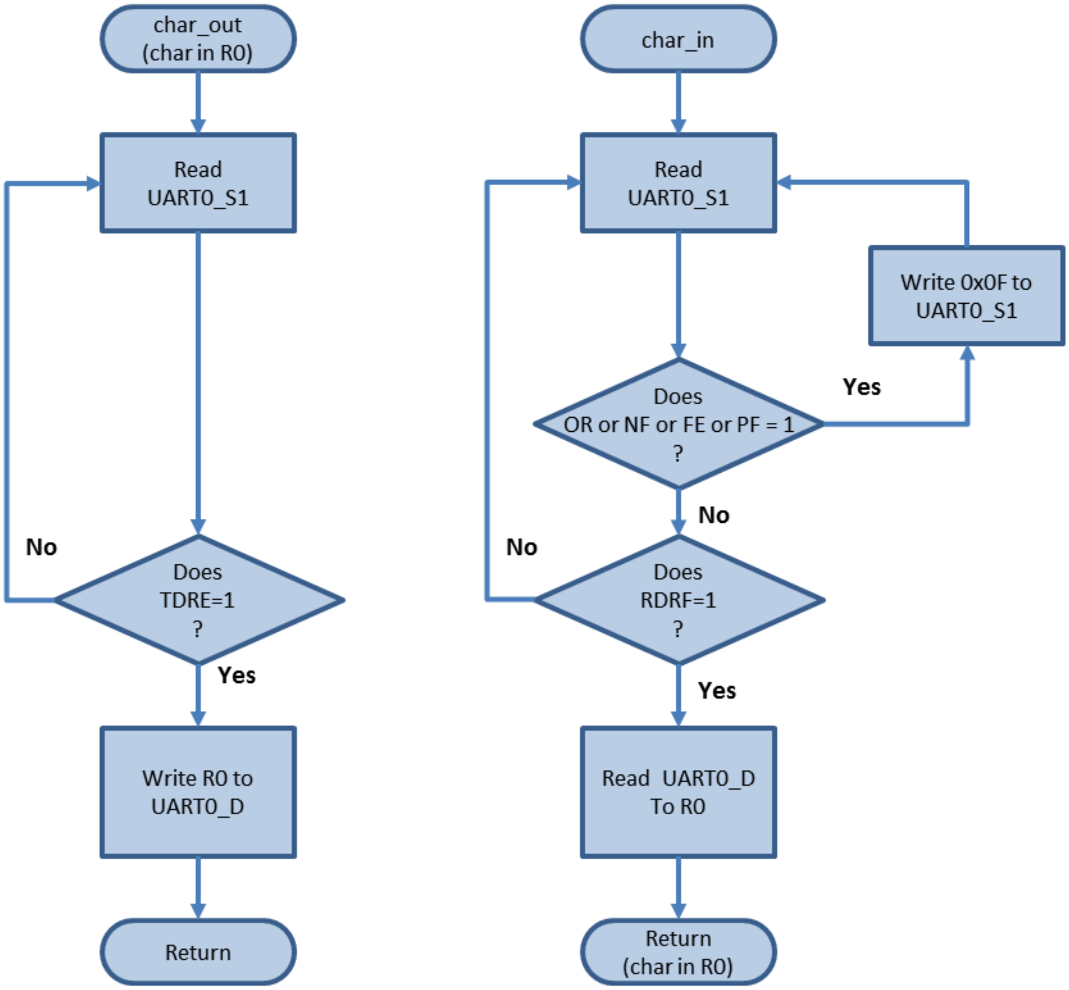


Figure 27 - char_out and char_in diagrams

The UART0Init subroutine should be completely functional as provided. By default, it configures UART0 to use 8 data bits, no parity, 1 stop bit and a baud rate of 9600 bps. The clock source is configured to be MGCPLLCLK/2 which operates at 48MHz.

The asm_main routine is complexly functional as provided. asm_main calls the UART0Init subroutine then enters a loop that calls char_in followed by char_out. char_in should wait for the user to enter a character into the terminal. When a character is entered, char_in should return the character in R0. char_out is then called which should be echoed back to the user. This process will then repeat indefinitely.

After completing the char_in and char_out routines connect the KL25Z to the PC. Build, debug and then run the code onto the board a usual. Open Tera Term on the PC that is connected to the board (other terminal programs can be used but Tera Term is recommended).

When Tera Term starts, select New Connection, Serial, then the COM port labeled OpenSDA in the port pull-down. The hit the OK button. (Figure 28)

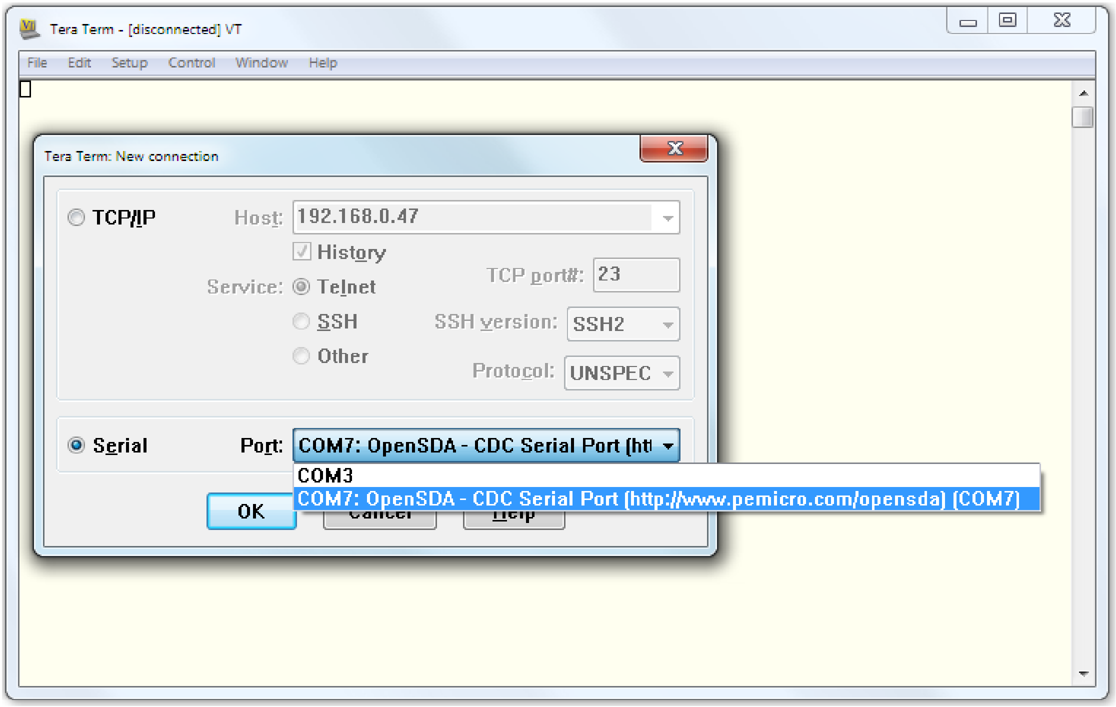


Figure 28 - Tera Term New Connection

Next select Setup >>> Serial port. In the Serial port setup dialog, set the baud rate, data, parity, stop and flow control to match the KL25Z settings. In the default case use: Baud: 9600, Data: 8 bit, Parity: none, Stop: 1 bit and Flow control: none. Then click the OK button. (Figure 29)

With the code running on the KL25Z and Tera Term setup properly, anything typed on the Tera Term console will be echoed back and displayed in the console window. (Figure 30).

Next stop the code running on the KL25Z then type characters in Tera Term. **Lab report: What happens and why?**

Next add a second "`BL char_out`" line to the program after the first but before the "`b loop`" statement. Build and run the code. **Lab report: What happens and why?**

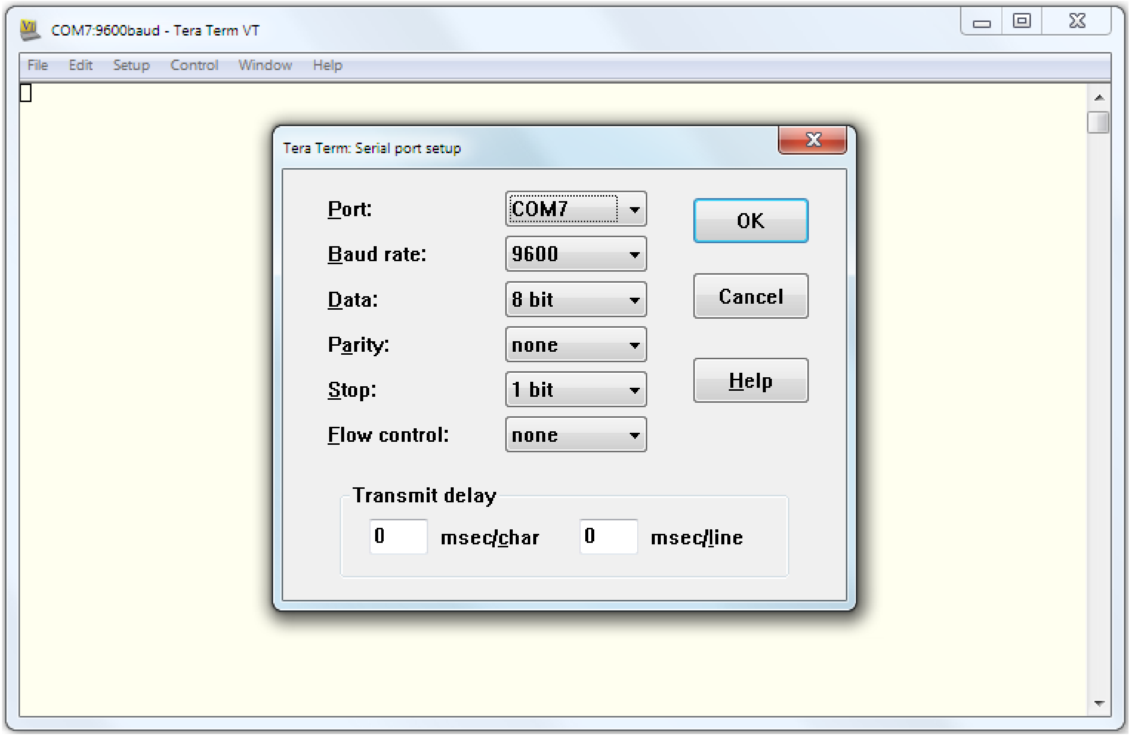


Figure 29 - Tera Term Serial Port Setup

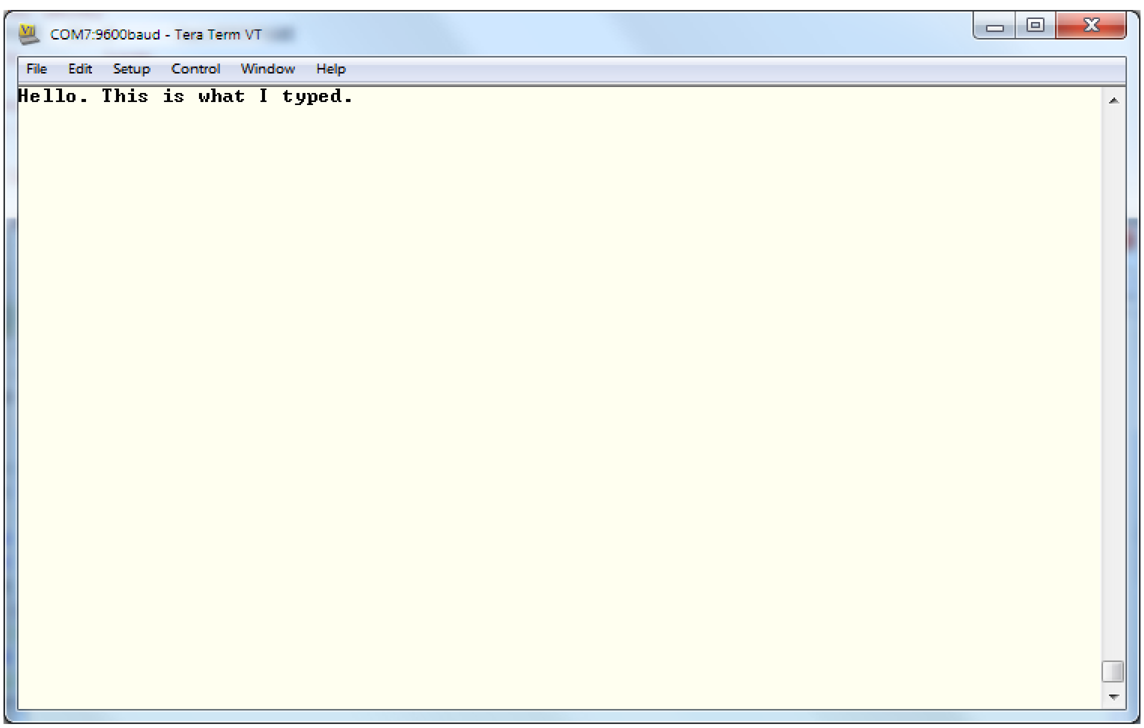


Figure 30 - Tera Term Displaying Echoed Characters

3.4.2 Lab 4, Part 2: Observing the Serial Port on the Oscilloscope

The objective of this part of the lab is to observe the serial port output on the oscilloscope. The code will be modified to continuously stream a single character. Then the oscilloscope will be used to view the output waveform. Oscilloscope traces will be captured for multiple UART configurations and analyzed both manually and with the serial decode utility on the scope.

Modify the code from the first part as follows:

In asm_main before the loop but after the call to UART0Init, load R0 with the ASCII code for the group (bench) number.

In the loop in asm_main, remove the call to char_in and add a call to a delay routine. The delay is to add some time between sending so it is easy to observe individual characters on the oscilloscope.

Run the code and verify output of the characters on Tera term.

With the oscilloscope off, install the Tektronix Computer Serial Module (DPO2COMP) into the DPO2012B oscilloscope (Figure 31).

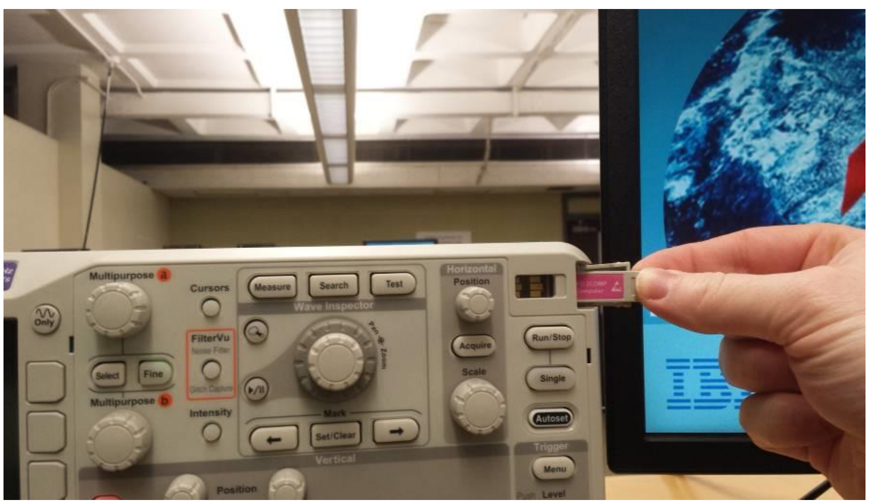


Figure 31 - Installing the DPO2COMP Module

Connect the oscilloscope to the KL25Z as shown in Figure 32. The UART0 transmit signal is output on pin PTA2. Use a scope probe to view the signal. Connect the ground of the scope probe to one of the ground pins on the board.

Turn on the oscilloscope.

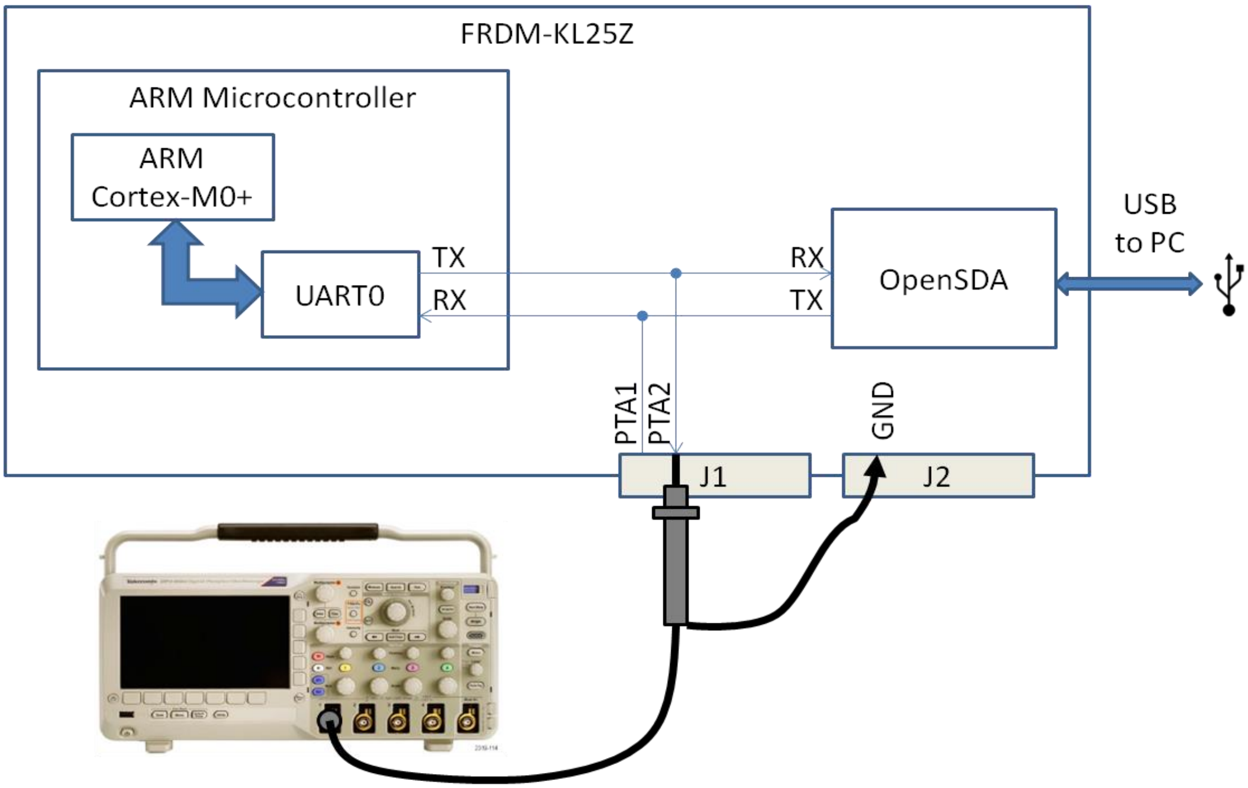


Figure 32 - Oscilloscope Setup

Verify the scope probe impedance is set to match the probe. Verify the probe scaling matches the probe (e.g. 1x, 10x). Verify the scope is set to DC coupling. To start with, the vertical amplitude can be set to 1V/div and the time base can be set to 100uS/div. Set the trigger mode to "Normal", falling edge, using the scope channel that is connected to the board output. Adjust the trigger level to about 1.5V. The waveform should be seen on the scope. Adjust the settings to optimize the view.

Next setup the serial decode tool on the oscilloscope. Start by pressing the Bus – B1 button. Then do the steps in Figure 33 thru Figure 39.

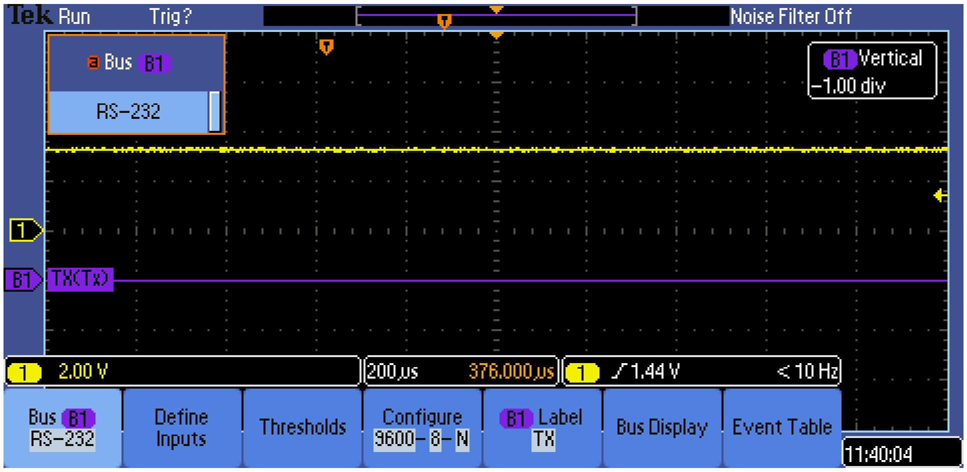


Figure 33 - Select bus type , RS-232

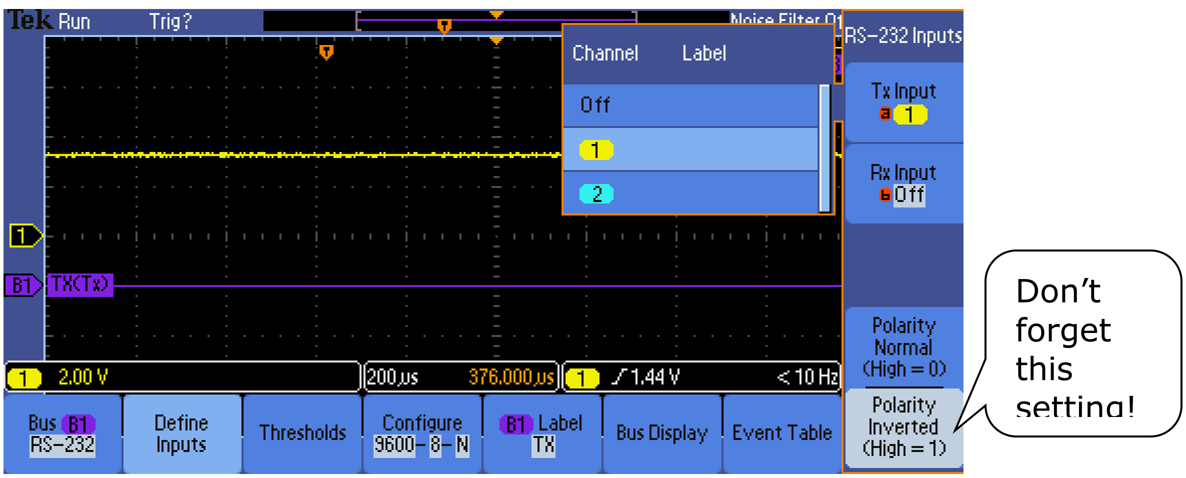


Figure 34 - Select the scope channel connected to the TX line

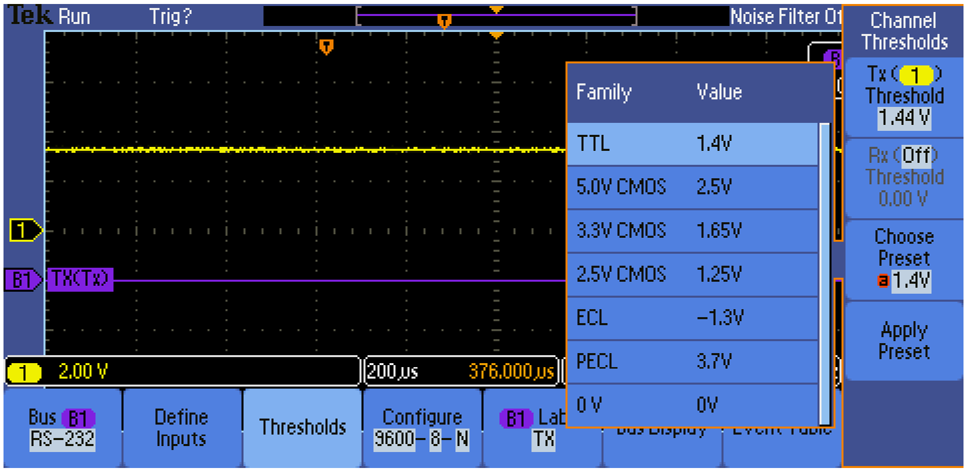


Figure 35 - Set the threshold using the TTL preset

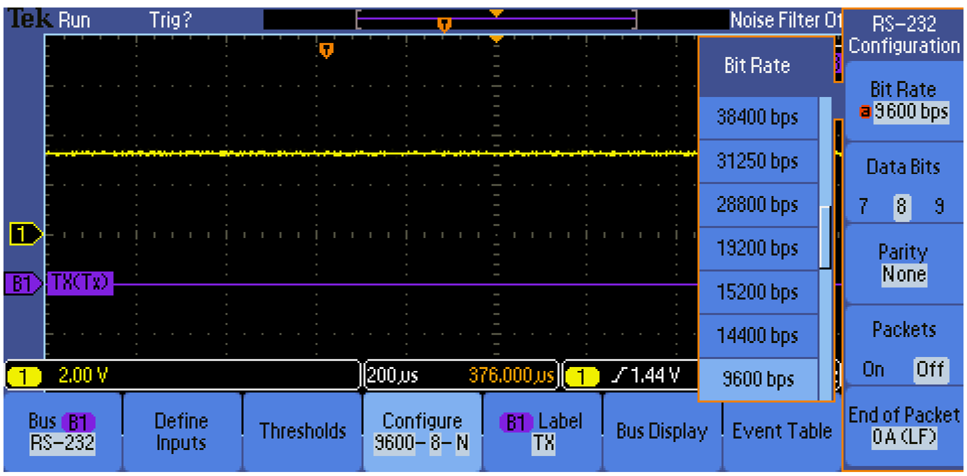


Figure 36 - Configure the serial settings to match the TX config

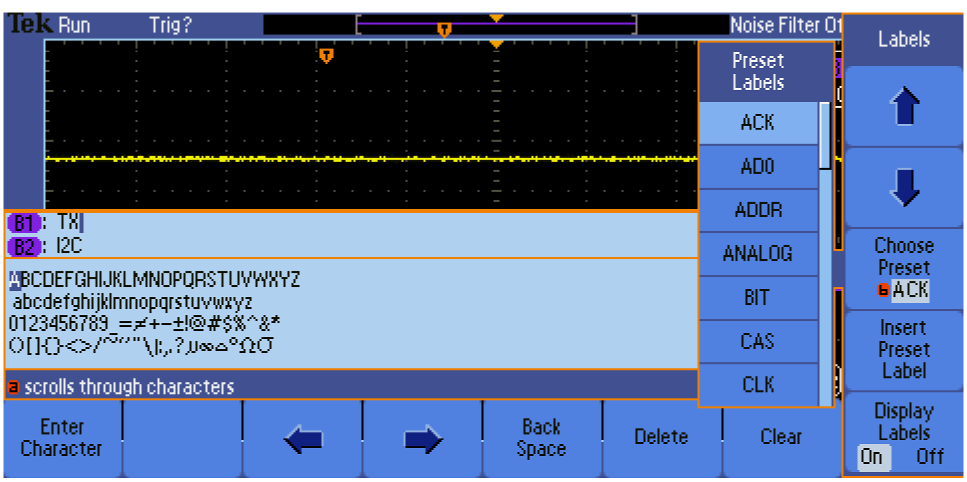


Figure 37 - Apply a label

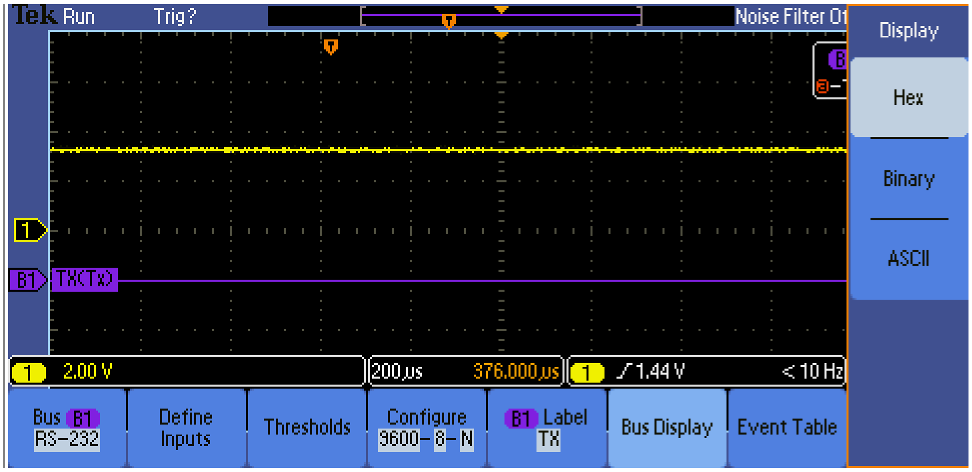


Figure 38 - Select how to view the characters

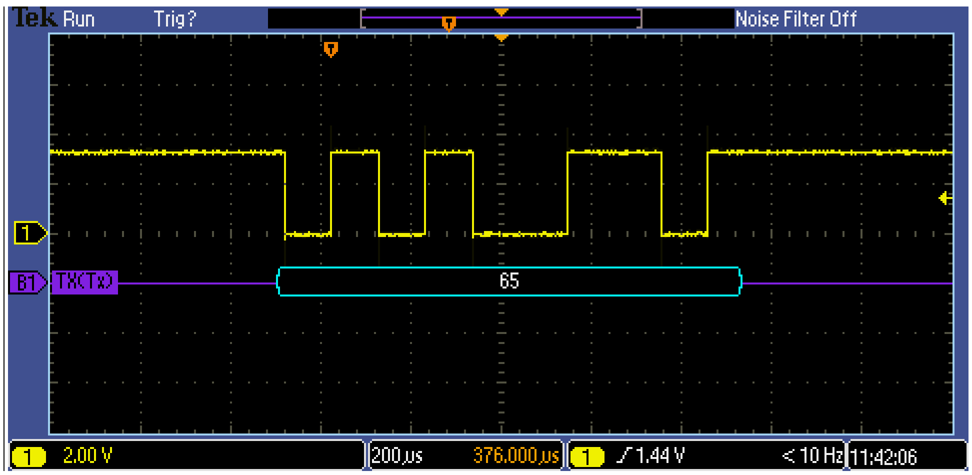


Figure 39 - The end result, 0x65 = ASCII 'e'

Store the waveform for the lab report.

Capture the waveforms for the configurations listed in Figure 40 for the lab report. For each baud rate, calculate new values for UART0_BDH and UART0_BLD and update the code in UART0Init. Change the baud rate in Tera Term and on the oscilloscope serial decode setup to match.

ASCII Character	Baud Rate	Data Bits	Parity	Stop Bits
Group number	9600	8	None	1
Group number	19200	8	None	1
Group number	115200	8	None	1
Last name initial of report author	115200	8	None	1

Figure 40 - Capture Waveform Settings

3.4.3 Lab 4, Lab report

For the lab report on this lab, be sure to include:

A description of how the lab was performed.

For Part 1, observed results and answers to two highlighted questions

For Part 2, four waveforms, (manual identification and serial decode of the four settings listed). For the manual identification of the serial waveforms in the report, identify:

- the voltage levels for high and low
- bit time
- start bit, data bits 0 thru 7, stop bit
- data bit values (0/1)
- match the data bits to the ASCII character value

Also include the new code that was written (e.g. char_in and char_out routines) and any changes made for each of the different setting configurations.

3.5 *Lab 5 – Calculator (UART Application)*

Lab Objectives

- To apply knowledge on the use of serial ports gained in lab 4 to solve a design problem

Problem

Your group needs to develop a simple calculator program. Unlike the previous labs, this lab will be done in C to provide some insight into how higher level languages operate on microprocessors. The calculator will use the serial routines from lab 4 to handle the data input and output interface with the user. All code for this program can only use base C instructions and the functions developed in lab 4. ***Use of other libraries, for example string.h, stdlib.h and stdio.h, is prohibited in the solution of this problem.***

There are multiple tiers of functionally which can be implemented, the more functions that are implemented, the higher the grade on the lab.

For tier 1, the calculator needs to take in 2 four-digit numbers separated by a plus sign and return the answer after an equals sign.

Examples: *User enters,* ***Program returns***

0001+0001= ***00002***

0100+0020**=00120**

1234+5678= ***06912***

9999+9999= ***19998***

Notice that the user is always entering four digits, even for values less than 1000. Also, notice the result is always five digits. The four digit decimal values have a range of 0000 to 9999, giving the range of the sum to be 0000 to 19998.

For tier 2 functionality is removing the need to enter the leading zeros.

Examples:

1+1=**00002**

100+20**=00120**

For tier 3 of functionality is outputting the results without the leading zeros.

Examples:

1+1=**2**

100+20**=120**

For ambitious students looking for a challenge, there is an extra tier of functionality, adding subtraction.

Examples:

10-1=**9**

20-23**=-3**

9999-9999=**0**

0-9999=-**9999**

Approach

Start with the project from lab 4. In the main.c file, add the external references to the assembly routines UART0Init, char_in and char_out and comment out the call to asm_main() as shown in Figure 41. In the main() function, the UART0Init() call runs the initialization function from the assembly file. The while(1) loop uses the char_in() function from the assembly file to read a character into myChar from the UART then outputs it back (aka ‘echoes’) to the UART using the char_out call. This loop repeats forever. Note the variable myChar has been defined as an single 8-bit byte to match the characters that are handled by the serial UART.

```
#include <MKL25Z4.H>

extern void asm_main(void);

extern void UART0Init(void);
extern unsigned char char_in(void);
extern void char_out(unsigned char);

int main (void) {

//   asm_main();  // uncomment to use assembly

     unsigned char myChar;

     UART0Init();

     while(1){
          myChar = char_in();
          char_out(myChar);
     }
}
```

Figure 41 - Serial IO C code Example

To achieve the basic calculator functionality, individual characters need to be read from the UART. These characters will be received in ASCII format. (Hint: Lookup at an ASCII table for reference!) The individual ASCII characters need to be converted into a numeric value to support the math operations. In ASCII formatting, digits are represented by the number+0x30.

For example the decimal number 1976 will come in as 0x31, 0x39, 0x37, 0x36. To convert the separate ASCII characters to a single value, the ASCII offset (0x30) needs to be removed and the individual values need to be weighted by its decimal place and summed.

The number 1976 can be thought of as:

1 * 1000 + 9 * 100 + 7 * 10 + 6 = 1976

This method can be coded directly but an alternative way to approach the solution is to use a loop like the one depicted in Figure 42. In this method, the value is built up as digits are entered and the loop is terminated when an ASCII value that is not a digit is entered.

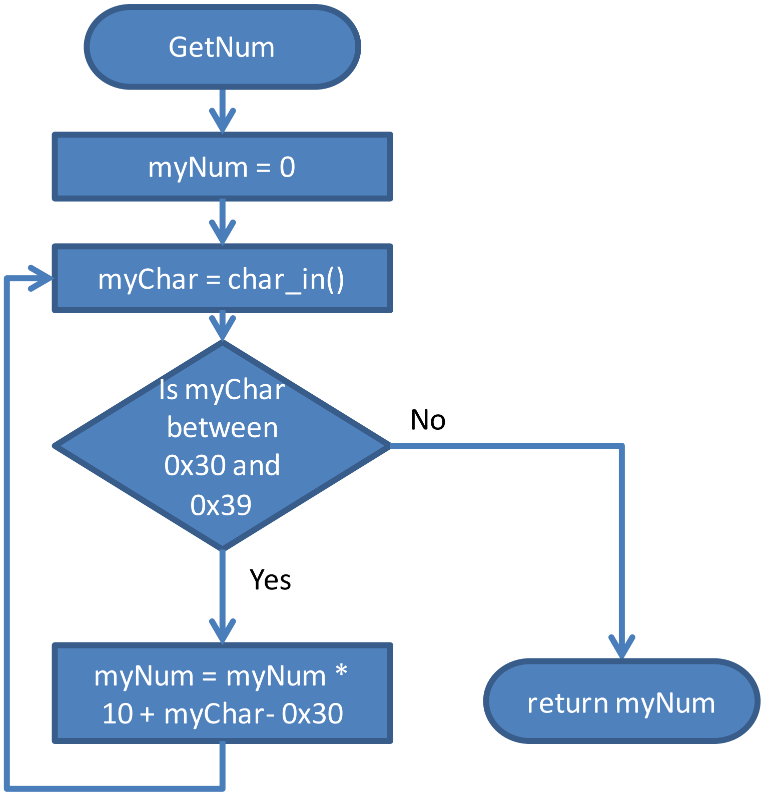


Figure 42 - GetNum Block Diagram

The algorithm for the GetNum routine can be thought of as:

((((0 * 10 + 1) * 10 + 9) * 10 + 7) * 10 + 6) = 1976

GetNum can be called twice and the two returned numbers can be added. Then the result needs to be displayed.

For display on the serial terminal, the result needs to be converter back to individual ASCII characters representing each digit. For example, the value 76543 would be displayed as ASCII characters 0x37, 0x36, 0x35, 0x34 0x33.

Two similar approaches can be taken to generate this output. For the first, the value is divided by subsequently smaller powers of ten (e.g. 10000, 1000, 100, 10, 1). The quotient from the division yields the digit to be converted to ASCII (add 0x30) for output and the remainder from the division yields the next value to be divided.

76543 / 10000 = Q: 7, R: 6543, Output Q + 0x30 = 0x37

6543 / 1000 = Q: 6, R: 543, Output Q + 0x30 = 0x36

543 / 100 = Q: 5, R: 43, Output Q + 0x30 = 0x35

43 / 10 = Q: 4, R: 3, Output Q + 0x30 = 0x34

3 / 1 = Q: 3, R: 0, Output Q + 0x30 = 0x33

Note that this method requires setting the initial divisor equal to or larger than the power of largest value to be outputted. If it is larger than the value, leading zeros will be outputted. This can be avoided by either sizing the initial divisor appropriately for the value being outputted or by suppressing the leading zeros until a non-zero digit comes out.

543 / 10000 = Q: 0, R: 543, Output Q + 0x30 = 0x30

543 / 1000 = Q: 0, R: 543, Output Q + 0x30 = 0x30

543 / 100 = Q: 5, R: 43, Output Q + 0x30 = 0x35

43 / 10 = Q: 4, R: 3, Output Q + 0x30 = 0x34

3 / 1 = Q: 3, R: 0, Output Q + 0x30 = 0x33

An alternative way to output the result is to always divide by 10, the remainder yields the digit to be converted to ASCII for output and the quotient gives the next value to be divided. The process stops once the quotient equals zero. Note the values come out in reverse order and need to be stored until the division is complete and the results can be output in reverse order.

76543 / 10 = Q: 7654, R: 3, Output R + 0x30 = 0x33

7654 / 10 = Q: 765, R: 4, Output R + 0x30 = 0x34

765 / 10 = Q: 76, R: 5, Output R + 0x30 = 0x35

76 / 10 = Q: 7, R: 6, Output R + 0x30 = 0x36

7 / 10 = Q: 0, R: 7, Output R + 0x30 = 0x37

Note that this technique has the benefit of inherently suppressing any leading zeros.

Either of the methods requires the use of C instructions to get the quotient and remainder. The quotient is returned by the division operator (/) and the remainder is returned by the modulus operator (%).

```
quotient = dividend / divisor;

remainder = dividend % divisor;
```

3.6 *Lab 6 – Digital-to-Analog Converter*

Lab Objectives

- To learn how to setup and operate the MCU digital-to-analog converter (DAC)
- To use the DAC to create a waveform generator

Introduction

Digital-to-Analog converters (DAC's) convert digital data to analog signals. They are a common peripheral used with microprocessors for applications such as controlling analog circuitry, audio and video generation, radio signal generation, etc.

The processor on the KL25Z board features an integrated, single-channel, 12-bit general-purpose DAC.

To use the DAC, the DAC0 clock enable bit must first be set in the SIM_SCGC6 register. Then the DACEN bit must be set in the DAC0_C0 register.

The output voltage of the DAC can then be set by writing the 12-bit DATA field which is split between the DAC0_DAT0L and DAC0_DAT0H registers. The DAC output voltage is based on the formula:

Vout = Vref * (1 + DATA[11-0])/4096.

Vref is set by wiring the Vref pin to a reference voltage. For the KL25Z, Vref is 3.3V.

When writing the DATA field, a 16-bit half-word store to the DAC0_DAT0L register can be used (STRH) to write both the lower 8-bits to DAC0_DAT0L and upper 4-bits to DAC0_DAT0H with one instruction. This is because the address of DAT0H is immediately after DAT0L and the processor is little-endian so the lower bits will be written to the first byte pointed to by the address followed by the upper bits in the next byte.

The DAC output DAC0_OUT is available on pin PTE30. On boot, this pin defaults to DAC0_OUT as its output function so the pin control register does not need to be changed.

Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DAC0 _DAT0L	**DATA[7-0]**							
DAC0 _DAT0H	Reserved				**DATA[11-8]**			
DAC0_C0	**DACEN**	DACRFS	DACTRGSEL	DACSWTRG	LPEN	0	DACBTIEN	DACBBIEN

Figure 43 - DAC Control Registers

Procedure

3.6.1 DAC Characterization

The goal of the first step is to configure the DAC to output fixed voltages and observe the output on the oscilloscope.

Connect the oscilloscope as shown Figure 45.

Load the code shown in Figure 44. The code starts by running the dac_init function which turns on the DAC clock and DAC enable. Then the code loads the address of the DAC0_DAT0L to R0 and clears R1. At the label dac_loop, the lower 16-bits of R1 are copied to [R0] which points to DAC0_DAT0L and DAC0_DAT0H, then loops around to repeat the copy. Note the use of STRH to do the 16-bit write.

Insert a breakpoint at the line 'B dac_loop'. Run the code, when the break point is reached, note the voltage on the oscilloscope. Now increase the value in R1 by about 150 to 250 by clicking on value in the register debug window and entering the new value. Run the code again until the breakpoint is reached. Note the new voltage on the oscilloscope. Repeat this process until the max value of 4095 (0xFFF) is reached on the DAC command. Use different step sizes each time that R1 is changed.

```
SIM_SCGC6       EQU  0x4004803C
DAC0_DAT0L      EQU  0x4003F000
DAC0_C0         EQU  0x4003F021

     AREA    asm_area, CODE, READONLY
     EXPORT  asm_main

asm_main  ;assembly entry point for C function, do not delete
; Add program code here
     BL  dac_init
     LDR R0,=DAC0_DAT0L
;    B   square_wave
;    B   saw_tooth
;    B   sine_wave
     MOVS R1,#0
dac_loop
     STRH R1,[R0]
     B    dac_loop

dac_init
     ; SIM_SCGC6[DAC0] = 1
     LDR  R0,=SIM_SCGC6
     LDR R1,[R0]
     LDR R2,=0x80000000
     ORRS R1,R2
     STR R1,[R0]

     ; Set DAC0_C0[DACEN] = 1 to enable DAC
     LDR  R0,=DAC0_C0
     MOVS R1,#0x80
     STRB R1,[R0]
     BX   LR

square_wave
     B square_wave

saw_tooth
     B saw_tooth

sine_wave
     B sine_wave

     AREA data_area, DATA, READWRITE
; Put variables here

     END
```

Figure 44 – DAC Code

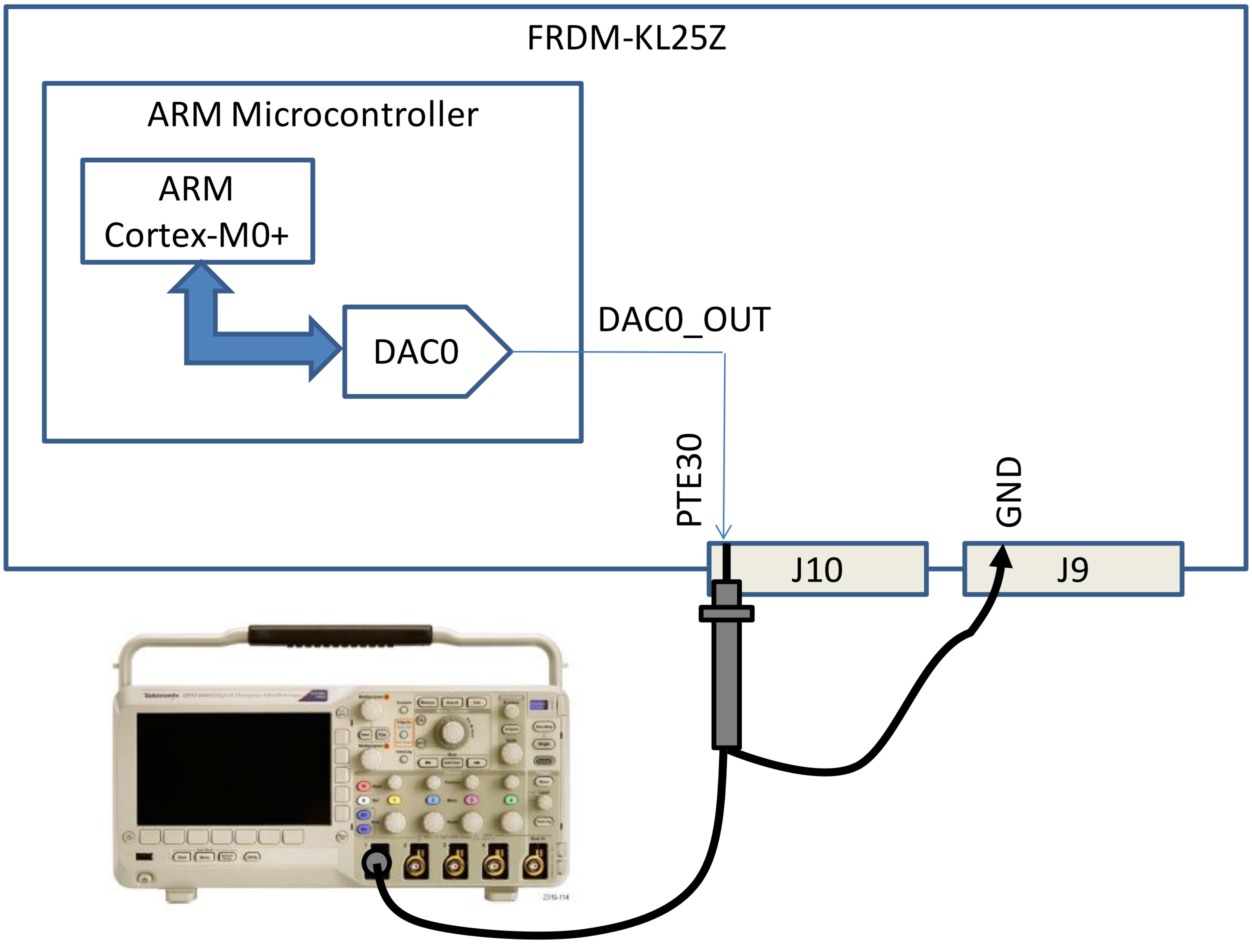


Figure 45 - DAC Oscilloscope Connection

3.6.2 Waveform Generator

Edit the code from the first part of the experiment to add functions to generate square, saw tooth and sine waves as shown in Figure 47, Figure 48 and Figure 49. Each waveform should be 1 kHz in frequency and 0 to 3.3 V in amplitude as shown in the plots.

For all three waveforms the timing can be achieved by inserting a delay loop and adjusting the delay count to set the desired interval between samples for the number of samples in the period of the given waveform. Figure 46 shows an example for how to calculate the delay for a 10 point waveform. This needs to be adjusted for the specific number of points in each waveform.

```
; clk is 48MHz, total of 3 clock cycles per delay_loop
; 1ms delay = 48000 clk cycles
; 1ms delay = 16000 delay loops
; for 10 point waveform, delay should be 1/10th, 1600 loops
; subtract 1 loop to compensate for fixed delay of 4 clks, 1599

delay
     LDR R2,=1599         ;2 clk cycles
delay_loop
     SUBS R2,#1           ;1 clk cycle
     BNE  delay_loop      ;2 clk cycles when branch taken
     BX   LR              ;2 clk cycles
```

Figure 46 – Delay Loop

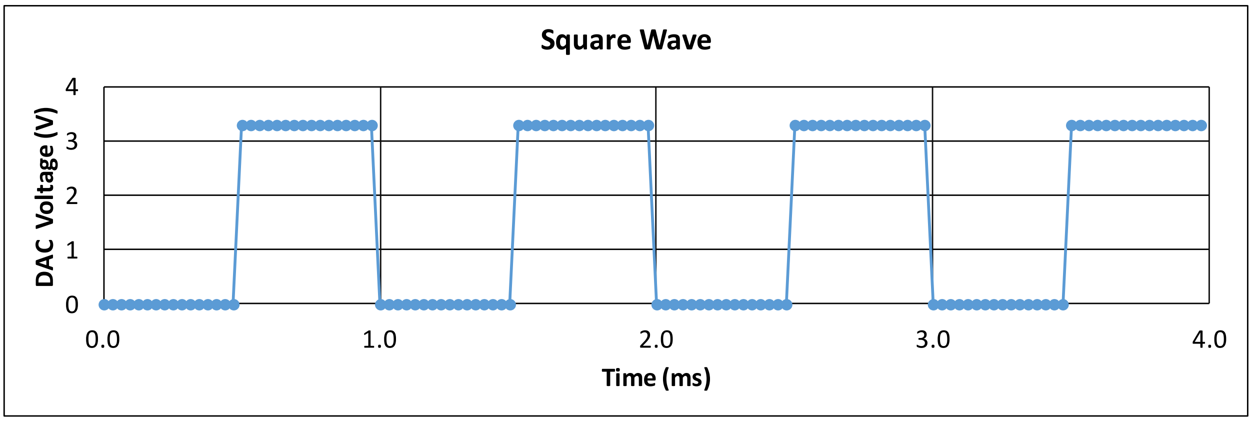


Figure 47 - DAC Square Wave

The Square Wave can be generated by a loop that writes two 16-bit values (0 and 4095) out in a repeated fashion to the DAC0_DAT0L/DAC0_DAT0H registers.

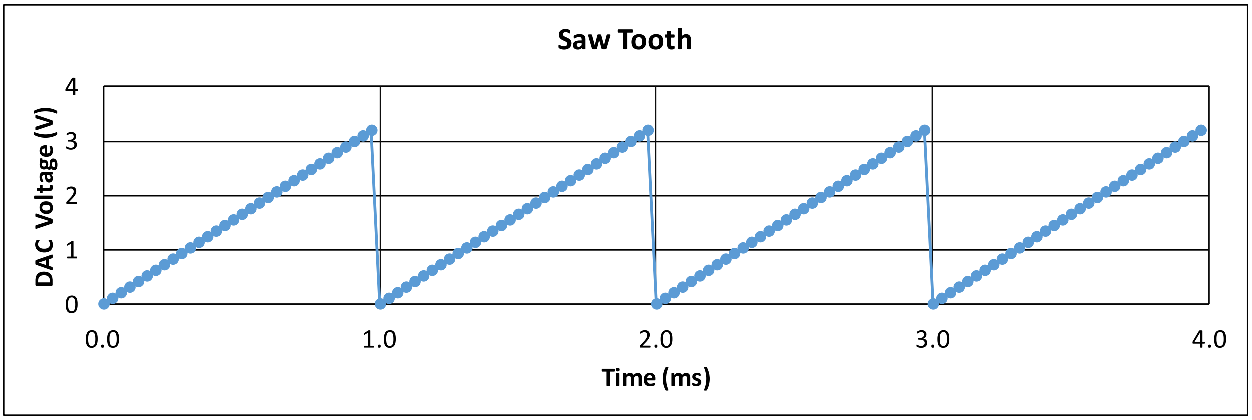


Figure 48 - DAC Saw Tooth

The Saw Tooth can be generated by incrementing a value by a fixed amount between 0 and 4095 and writing the value to the DAC0_DAT0L/DAC0_DAT0H registers.

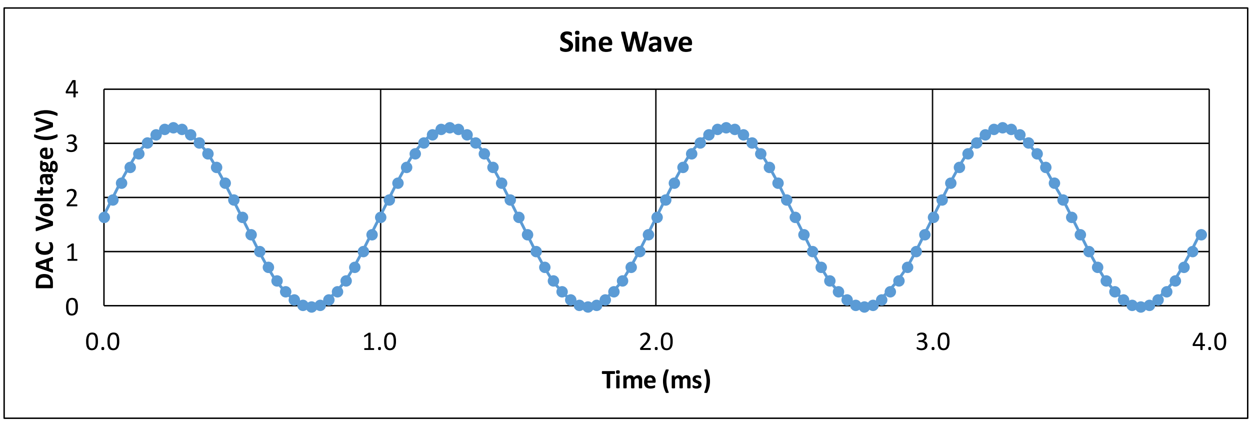


Figure 49 - DAC Sine Wave

The easiest way to generate Sine Wave is to pre-calculate values for the sine wave then to loop through the list of values to generate the waveform. The formula DAC0_DAT0 = 2047 + 2047 * sin(2 * pi * t / N) can be used to calculate a list of values. N is the number of values and t = 0 to N. The code shown in Figure 50 provides an example of how to read values out of a list of values.

```
sine
     LDR R0,=DAC0_DAT0L
sine_restart
     LDR R3,=sine_val        ;R3 is pointer address
     LDR R4,=sine_val_end    ;R4 is end address
sine_loop
     LDRH R1,[R3]        ;read 16-bit val from pointer address
     ADDS R3,#2          ;inc pointer 2 bytes (16-bits)
     STRH R1,[R0]        ;write 16-bit val to dac
     CMP R3,R4           ;check if at or past end of array
     BGE sine_restart    ;if at end restart
     B sine_loop         ;else loop

sine_val
     DCW 2048,4095,2048,1
sine_val_end
     DCW 0               ;placeholder for addr of end of array
```

Figure 50 - Example code for Sine Wave

3.6.3 Lab 6, Lab report

For the lab report on this lab, be sure to include:

A description of how the lab was performed.

For Part 1, include a table and plot of the command, expected (calculated) voltage and measured voltage for each step measured.

For Part 2, include a description of all calculations for DAC settings and delay time counts and three oscilloscope waveforms (square, saw tooth, sine) with the following identified:

- The signal amplitude
- The signal frequency

Also include the all code that was written.

3.7 *Lab 7 – Analog-to-Digital Converter*

Lab Objectives

- To learn how to setup and operate the MCU analog-to-digital converter (ADC)
- Characterize the ADC under different configurations

Introduction

Analog-to-Digital Converters (ADC's) convert analog signals to digital data. They are a common peripheral used with microprocessors for applications such as monitoring analog circuitry (voltages, temperature sensors, etc), digitizing audio and video signals, digitizing radio signal, etc.

The KL25Z microprocessor features an integrated 16-bit, successive approximation, analog-to-digital converter. The converter supports both single ended and differential inputs. Up to 24 inputs are available in single ended mode and up to 4 input pairs are available in differential mode. Additionally an internal temperature sensor, internal 1V reference or the ADC high and low references voltages can uses to feed the input to the ADC.

The ADC has multiple modes of operation and many features which are controlled by a set of 27 registers. Fortunately for this course, basic functionally can be achieved by just using a small subset of these. For this experiment, the ADC will be operated in "single conversion" mode. This means that the ADC will need to be commanded by software to perform each conversion (a conversion is the processes the ADC does to sample the analog input and produce a corresponding digital value).

Like all the other peripherals in the processor, the ADC has a clock that must be enabled with the ADC0 bit in the SIM_SCGC6 register.

Bit	31 - 8	7	6	5	4	3	2	1	0
ADC0_CFG1	Reserved	ADLPC	ADIV	ADLSMP		MODE		ADICLK	
Value	all bits 0	0	0	0	0	See Figure 52		0	0

Figure 51 - ADC0_CFG1 Register

Next the ADC resolution must be set using the MODE bits in the ADC0_CFG1 register. All the other configuration bits in this register can be left at their default values of zero.

MODE value	**ADC Input**
00b	8-bit Single Ended Conversion
01b	12-bit Single Ended Conversion
10b	10-bit Single Ended Conversion
11b	16-bit Single Ended Conversion

Figure 52- MODE Bit Definitions

Bit	31 - 8	7	6	5	4	3	2	1	0
ADC0_SC3	Reserved	CAL	CALF	Reserved		ADCO	AVGE	AVGS	
Value	all bits 0	0	RO	0	0	0	See text	See Figure 54 - AVGS Register SettingsFigure 54	

Figure 53 - ADC0_SC3 Register

AVGS value	**ADC Input**
00b	4 samples averaged
01b	8 samples averaged
10b	16 samples averaged
11b	32 samples averaged

Figure 54 - AVGS Register Settings

The conversions are controlled with the ADC0_SC1A register.

Bit	31 - 8	7	6	5	4	3	2	1	0
ADC0_SC1A	Reserved	COCO	AIEN	DIFF	ADCH				
Value	all bits 0	RO	0	0	See Figure 56				

Figure 55 - ADC0_SC1A Register

The COCO bit is a read only bit that indicates if the conversion is complete. Once this bit is set to a one (1), the result of the conversion can be read in the Data Results Register ADC0_RA.

The AIEN bit should be set to 0 to disable interrupts.

The DIFF bit should be set to 0 to set the ADC to operate in single ended mode.

Finally the ADCH bits select which channel the ADC should do a conversion on.

ADCH value (binary)	**ADCH value (hex)**	**ADC Input**
00000b thru 10111b	0x00 thru 0x17	AD0 thru AD23
11010b	0x1A	Internal Temperature Sensor
11011b	0x1B	Internal 1V Bandgap Reference
11101b	0x1D	VREFSH
11110b	0x1E	VREFSL

Figure 56- ADCH Bit Definitions

The results of the conversion are returned in the ADC data results register ADC0_RA. The number of bits used in this register depends on the resolution set in the MODE field. For single ended conversion the results are unsigned, right justified.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16-bit Single ended	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
12-bit Single ended	0	0	0	0	D	D	D	D	D	D	D	D	D	D	D	D
10-bit Single ended	0	0	0	0	0	0	D	D	D	D	D	D	D	D	D	D
8-bit Single ended	0	0	0	0	0	0	0	0	D	D	D	D	D	D	D	D

Figure 57 - ADC0_RA Data Register Format

The results from the data register can be converted to the equivalent voltage with the following formula:

$V_{ADC} = DATA * (V_{REFSH} – V_{REFSL}) / 2^N$

Where N is the resolution set with the ADCH field, V_{REFSH} is 3.3V and V_{REFSL} is 0V.

<u>Procedure</u>

3.7.1 Part 1 – Bandgap Reference Measurement

Load the code shown in Figure 58. The code starts by running the adc_init function which turns on the ADC clock in the SIM_SCGC6 register and then setups up the ADC0_CFG1 and ADC0_SC3 registers for 16-bit conversions without averaging. Adc_init also enables the bandgap reference by setting the BGBE bit in the PMC_REGSC register.

Next the adc_read function is called. This function expects R0 to contain the appropriate setting for the ADCH bits in the SC1A register that select which ADC channel to read. In the code, three equates have been defined to select the AD0, TEMP, or BANDGAP sources which will be used in this lab.

adc_read or’s the ADCH value contained in R0 with the default value for the other bits SC1A to then stores this value into the SC1A register. This store starts the ADC conversion. The ADC indicates that the conversion is complete when the COCO bit is set. adc_read polls the AC1A, waiting for the COCO bit to be set then once this condition is met, reads the RA register which contains the result of the conversion. adc_read returns the ADC conversion result in R0.

```
SIM_SCGC6   EQU  0x4004803C

ADC0_SC1A      EQU       0x4003B000
ADC0_CFG1      EQU       0x4003B008
ADC0_RA        EQU       0x4003B010
ADC0_SC3       EQU       0x4003B024
PMC_REGSC      EQU       0x4007D002

SC1A_DEFAULTS  EQU      0x00
ADCH_AD0       EQU      0x00
ADCH_TEMP      EQU      0x1A
ADCH_BANDGAP   EQU      0x1B
COCO_FLAG_MASK EQU      0x00000080

SIM_SCGC5   EQU  0x40048038          ;SIM_SCGC5 address

PORTB_PCR0  EQU  0x4004A000 + 4 * 0   ;PORTB_PCR0 address

GPIOB_PSOR  EQU  0x400FF044          ;GPIOB_PSOR address
GPIOB_PCOR  EQU  0x400FF048          ;GPIOB_PCOR address
GPIOB_PTOR  EQU  0x400FF04C          ;GPIOB_PTOR address
GPIOB_PDDR  EQU  0x400FF054          ;GPIOB_PDDR address

PTB0_MASK  EQU 0x00000001

    AREA    asm_area, CODE, READONLY
    EXPORT  asm_main

asm_main  ;assembly entry point for C function, do not delete
; Add program code here
     BL   adc_init
     BL   gpio_init
adc_loop
     ;Load R0 with the channel to read
     LDR  R0,= ADCH_BANDGAP
     BL   ptb0_on          ;turn gpio on
     BL   adc_read         ;do adc conversion
     BL   ptb0_off         ;turn gpio off
     B    adc_loop

adc_init
     ; SIM_SCGC6[ADC0] = 1
     LDR  R0,=SIM_SCGC6
     LDR  R1,[R0]
     LDR  R2,=0x08000000
     ORRS R1,R2
     STR  R1,[R0]
```

```
    ; Set ADC0_CFG1[MODE] = 11b for 16-bit results
    LDR  R0,=ADC0_CFG1
    LDR  R1,=0x0000000C
    STR  R1,[R0]

    ; Set ADC0_SC3[AVGE] = 0b to disable averaging
    ; Set ADC0_SC3[AVGS] = 00b for 4 sample averages
    LDR  R0,= ADC0_SC3
    LDR  R1,=0x00000000
    STR  R1,[R0]

    ; Set PMC_REGSC[BGBE] = 1b to enable 1V bandgap reference
    LDR  R0,=PMC_REGSC
    LDR  R1,=0x01
    STRB R1,[R0]

    BX   LR

;When called, R0 contains SC1A_ADCH value
;Returns ADC value in R0
```

```
adc_read
     LDR  R1,=SC1A_DEFAULTS
     ORRS R0,R1
     LDR  R1,=ADC0_SC1A
     STR  R0,[R1]
     LDR  R2,=COCO_FLAG_MASK
adc_read_wait
     LDR  R0,[R1]
     TST  R0,R2
     BEQ  adc_read_wait
     LDR  R1,=ADC0_RA
     LDR  R0,[R1]
     BX   LR

ptb0_on
     LDR R2,=GPIOB_PSOR  ;Load address of GPIOB_PCOR to R0
     LDR R1,=PTB0_MASK    ;Load value to R1
     STR R1,[R2]         ;Put value into GPIOB_PCOR
     BX LR

ptb0_off
     LDR R2,=GPIOB_PCOR  ;Load address of GPIOB_PSOR to R0
     LDR R1,=PTB0_MASK    ;Load value to R1
     STR R1,[R2]         ;Put value into GPIOB_PSOR
     BX LR


gpio_init
     ; Turns on clocks for all ports
     LDR R0,=SIM_SCGC5    ;Load address of SIM_SCGC5 to R0
     LDR R1,[R0]          ;Get original value of SIM_SCGC5
     LDR R2,=0x00003E00   ;Load mask for bits to set to R2
     ORRS R1,R2           ;Set bits with OR of orig val and
mask
     STR R1,[R0]          ;Put new value back into SIM_SCGC5

     ; Setup PORTB Pin 0 to be output
     LDR R0,=PORTB_PCR0   ;Load address of PORTB_PCR0 to R0
     LDR R1,=0x00000100    ;Load new value to R1
     STR R1,[R0]           ;Put value into PORTB_PCR0

     ;Set bits in DDR register to enable outputs to drive LED's
     LDR R2,=PTB0_MASK
     LDR R0,=GPIOB_PDDR   ;Load address of GPIOB_PDDR to R0
     LDR R1,[R0]          ;Get original value of GPIOB_PDDR
     ORRS R1,R2           ;Set bits with OR of orig val and
mask
```

```
     STR R1,[R0]          ;Put new value back into GPIOB_PDDR
     BX LR

     AREA data_area, DATA, READWRITE
; Put variables here

     END
```

Figure 58 – ADC Code

For the first part of this lab, run the code as shown with ADCH source set to the internal 1V bandgap. Insert a breakpoint at the line “B adc_loop”. Run the code, when the breakpoint is reached record the ADC result returned in R0.

For each of the configurations show in Table 2, run 10 conversions and record the results of the ADC conversion returned in R0 and the conversion time.

The conversion time can be measured by observing pin PTB0 with an oscilloscope. Note this pin is configured to be an output using the gpio_init function and the ptb0_on function is called before the adc_read and ptb0_off is called after the adc_read function. Therefore, PTB0 will be on for the duration that the ADC is doing its conversion.

For the various configurations change the adc_init code to control the averaging with the AGVE and AVGS fields in the ADC0_SC3 register and ADC resolution using the MODE bits in the ADC0_CFG1 register.

Table 2 - Lab7 Part 1 Test Summary

Case	Resolution	Averaging	Number of runs
1	16-bits	None	10
2	16-bits	8 averages per conversion	10
3	16-bits	32 averages per conversion	10
4	12-bits	None	10
5	10-bits	None	10
6	8-bits	None	10

3.7.2 Part 2 – Temperature Sensor Measurement

For the this part of the lab, run the code as shown with ADCH source set to the temperature sensor. Setup adc_init to run the ADC in 16-bit mode with 32 averages.

Insert a breakpoint at the line "B adc_loop". Run the code, when the breakpoint is reached, record the ADC result returned in R0. Run a single conversion and record the results.

Calculate the internal temperature of the processor with the following equation:

$TEMP = 25 - ((V_{TEMP} - V_{TEMP25})/m)$

where **V_{TEMP} is the voltage** reported by the ADC for the temperature sensor, V_{TEMP25} is 719 mV and m is 1.715 mV/C.

3.7.3 Part 3 – External Voltage Measurement

For the this part of the lab, run the code as shown with ADCH source set to input AD0. Setup adc_init to run the ADC in 16-bit mode with 32 averages.

Connect a 5k or 10k potentiometer between 3.3V, PTE20 (note ADC input AD0 is on PTE20) and GND as shown in Figure 59. Connect a voltmeter between AD0 and GND and setup the voltmeter to measure DC voltage.

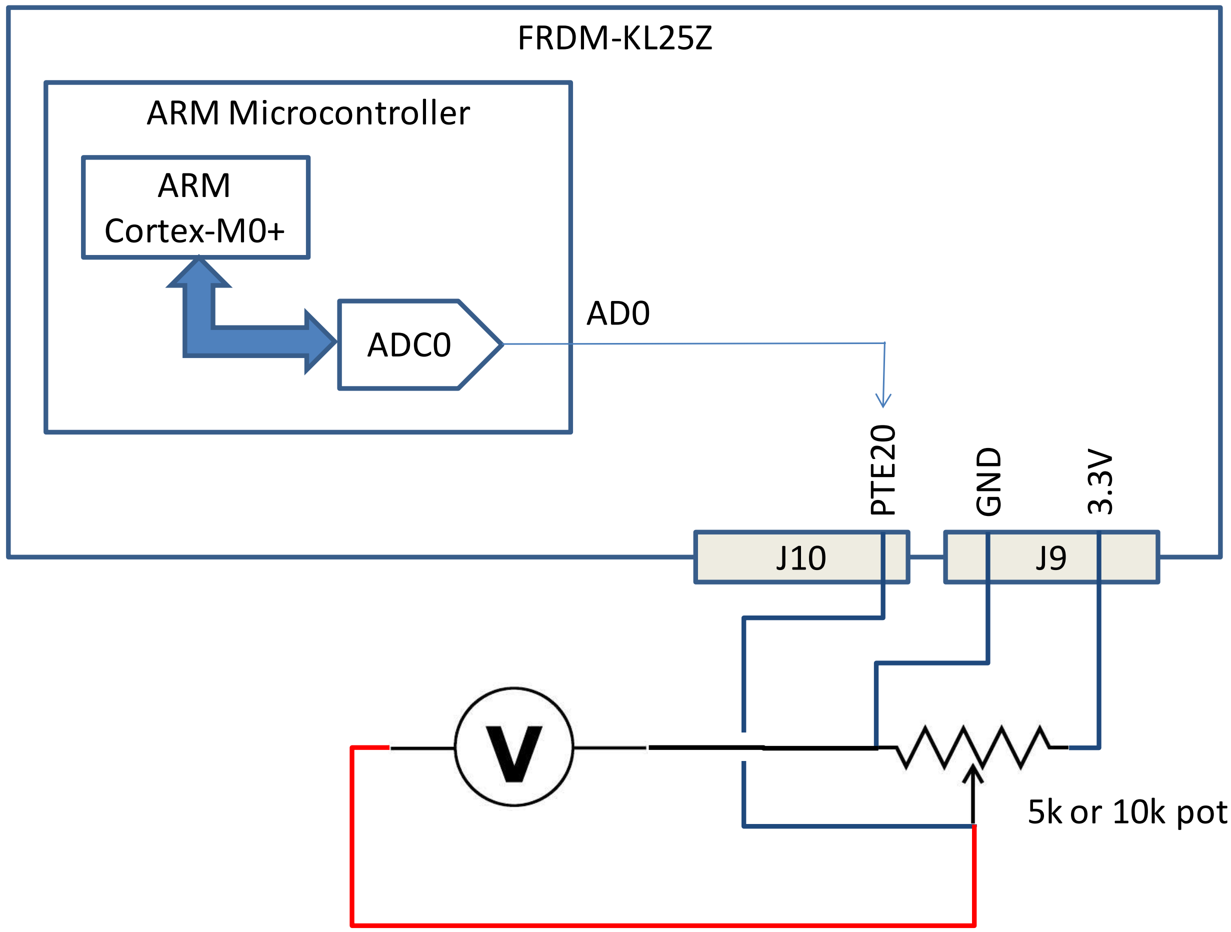


Figure 59 - Lab7, Part 3 setup

Adjust the pot to one end of the range. Insert a breakpoint at the line "B adc_loop". Run the code, when the breakpoint is reached, record the ADC result returned in R0 and the voltage on the voltmeter. Turn the pot slightly. Run another conversion and record the results. Perform this process about 10 times until the full range of the pot is covered.

3.7.4 Lab 7, Lab report

For the lab report on this lab, be sure to include:

A description of how the lab was performed.

For Part 1, include a table of each of the sets of measured data (conversion values and conversion times) and the corresponding voltages calculated from the data.

For Part 2, include the measured data and the corresponding calculated temperature.

For Part 3, include the measured data and the corresponding calculated voltage. Plot the calculated voltage vs the voltmeter voltage.

3.8 *Lab 8 – Interrupts and Exceptions*

Lab Objectives

- To learn how to setup and use interrupts and exceptions in the MCU

Introduction

During normal operation, a microprocessor executes instructions in a sequential fashion only changing the order of execution when a branch occurs or a subroutine is called. Interrupts and excerptions offer an alternate means to change what code a processor is executing based on events that occur. The events can be an external condition changing, such as an input GPIO changing state, reception of data on the serial port or the ADC completing a conversion. When an event such as these occurs, the processor can be programmed to trip an interrupt which causes the processor to execute code at a particular address. This code is known as an interrupt hander and behaves similar to a subroutine.

3.8.1 Part 1: SysTick Timer Interrupt

The Cortex-M0+ processor provides a simple hardware timer, known as the SysTick Timer, intended for scheduling periodic events in an operating system. The SysTick timer is a 24-bit register, known current value register or CVR, which counts down. When the timer reaches zero, an interrupt occurs and the SysTick counter is automatically reloaded with a "reload value" contained in the reload value register or RVR.

The code in Figure 60 shows an example of how the SysTick timer can be used to trigger an interrupt that flashes the on-board LED green. Note this code is done in C, not assembly.

The code in the GPIO_init function initializes the three GPIO's used to drive the on-board tri-color LED (see Lab 2 for details).

Next is a basic interrupt handler called SysTick_Handler. The name of this function is carefully chosen to match a pre-defined name that is in the vendor supplied board support header, MKL25Z4.H. By using the same name, the modified code for the interrupt handler is used instead of the default code (which does nothing). SysTick_Handler toggles the green drive line for the LED and increments a counter.

Last is the normal main routine. The main routine calls a function SysTick_Config, which initializes the SysTick timer. The value passed to the function is the number of processor "ticks" that are counted between each

time the interrupt is triggered. Next the GPIO_Init function is called. Then the code enters a while(1) loop that runs for ever.

Load and run the code shown in Figure 60. Notice the green LED on the board flashes, even though the main code is stuck in the while(1) loop.

Add a breakpoint to the first line of the SysTick_Handler (i.e. PTB->PTOR…). Repeatedly run the code (press F5). Notice how the LED toggles and the counters increments. Navigate to the SysTick registers (Peripherals >> System Viewer >> SysTick). Answer the questions in the Lab 8 worksheet.

Next modify the code to toggle an additional GPIO line available on the expansion headers (for instance PTB0). Connect an oscilloscope to the line and run the code without a breakpoint. A square wave should be observed. Use the oscilloscope to measure the square wave frequency.

Calculate the tick frequency taking the measured square wave frequency and multiplying it by 2x the number of ticks setup in the SysTick_config call (Note the 2x is needed due because the interrupt triggers twice for a full cycle). Include the result in the lab worksheet.

Repeat this for the alternate SysTick_config shown on the worksheet.

3.8.2 Part 2: Hard Faults

Exceptions occur when the processor encounters a condition that it cannot handle in a normal way. When an exception occurs, an exception handler is called in much the same way as an interrupt handler. The Cortex-M0+ has somewhat limited diagnostic information to determine but it is still possible to track down the cause of many hard faults.

Create a new project and load the code shown in Figure 61. Note this is assembly code so it should be loaded into asm_main. Step through the code. Note the LDR R1,[R0] triggers an exception and calls the HardFault_Handler. This is the default routine that locks the processor into a never ending loop. (You've probably experienced this multiple times throughout the course when you've had an error in your code.)

A more advanced fault hander could produce additional debug info. For instance, the most of the registers contain their values from when the fault occurred. One important piece of information that is somewhat hidden is the address of the code that executing when the fault occurred. This has been pushed to the stack along with some of the registers and can be found by viewing SP+0x18 in a memory window. The first 4 bytes displayed is the address of the code that cause the error. Note the syntax can be entered as shown into the memory window. SP is the stack pointer (R13) and 0x18 is

the offset from the current stack pointer position to the address of the bad instruction.

Restart the debug session. In the disassembly window, note the address of the LDR R1, [R0] instruction before running the code. Run the code until the exception occurs and then find the address pushed to the stack. Compare the two addresses (remember the values pushed to the stack are in little endian!)

```c
#include <MKL25Z4.H>

#define GREEN_MASK 0x00080000
#define RED_MASK 0x00040000
#define BLUE_MASK 0x00000002

void GPIO_Init(void){
	SIM->SCGC5 |= 0x00003E00;
	// Setup output pin control registers
	PORTB->PCR[18] = 0x00000100;
	PORTB->PCR[19] = 0x00000100;
	PORTD->PCR[1] = 0x00000100;

	// Setup output data direction registers
	PTB->PDDR |= GREEN_MASK;
	PTB->PDDR |= RED_MASK;
	PTD->PDDR |= BLUE_MASK;

	//Force LED's off
	PTB->PSOR |= GREEN_MASK;
	PTB->PSOR |= RED_MASK;
	PTD->PSOR |= BLUE_MASK;
}

uint32_t counter;

void SysTick_Handler(void){
	PTB->PTOR = GREEN_MASK;  //Toggle green LED
	counter++;
}

int main (void) {
	SysTick_Config(2000000);

	GPIO_Init();

	while(1){
	}
}
```

Figure 60 – SysTick Example Code

```
     AREA asm_area, CODE, READONLY
     EXPORT  asm_main

asm_main  ;assembly entry point for C function, do not delete
; Add program code here

     LDR R0, =0x30000000   ;this is not a valid address
     LDR R1,[R0]           ;trying to copy from an invalid
                           ;address causes hard fault

loop
     B    loop


; Put constants here

 AREA data_area, DATA, READWRITE
; Put variables here

 END
```

Figure 61 – Code to Trigger Hard Fault Exception

4 Troubleshooting

Message: Error #5 Cannot open source input file “MKL25Z4.h” **Cause:** Kinetis KL25Z support pack not loaded after installing KEIL uVision 5. **Fix:** Follow procedure in section 2.1 to install the Keil::Kinetis_KLxx_DFP support pack. **Note:** Older versions of uVision may require downloading and installing the support pack manually. For instance, many of the lab PC’s have Keil 5.1.0 installed. Go to www.keil.com/dd2/pack and download Kinetis_KLxx_DFP.1.0.1.pack. Then install the pack manually using File>>Install in the Pack Installer in uVision.
Message: Error #A1163E Unknown opcode code XXXX, expected opcode or macro **Cause:** Label not in first character position in line of code. **Fix:** Make sure all labels are in the first column.
Message: startup_MKL25Z4.s: error: A1023E: File "startup_MKL25Z4.s" could not be opened: No such file or directory **Cause:** Caused by opening the project directly from the project zip file. **Fix:** Extract the zip file, then open the project.
Message: error: #5: cannot open source input file "core_cm0plus.h": No such file or directory **Cause:** **Fix:** Open Options for Target (Alt+F7). Add path to missing file (C:\Keil_v5\ARM\Pack\ARM\CMSIS\4.2.0\CMSIS\Include) to C/C++ “Include Paths” field. Note the exact location to the missing file may change with the version of

5 Document Change History

Version	Date	By	Changes
3.00	1/14/2014	DJH	Initial release for comment on sections 1, 2 and 3.1.
3.01	1/20/2014	DJH	Completed lab 1 and added course objectives.
3.02	1/29/2014	DJH	Added lab 2 and modified initial install instructions for uVision 5.
3.03	1/31/2014	DJH	Minor corrections to code in lab 1 and additions to troubleshooting section.
3.04	2/10/2014	DJH	Added lab 3.
3.05	3/26/2014	DJH	Added lab 4.
3.06	8/31/2014	DJH	Added lab 5. Added some discussion to lab 1 and added info on receiver error flags to lab 4.
3.07	11/17/2014	DJH	Added lab 6. Minor corrections to labs 4 and 5.
3.08	11/17/2014	DJH	Fixed errors in lab 6 Vout and time delay calculations.
3.09	11/23/2014	DJH	Added Lab 7.
3.10	12/1/2014	DJH	Corrections to Lab 7: enabled bandgap in code, fixed code typo.
3.11	1/13/2015	DJH	Revised lab 2.
3.12	4/27/2015	DJH	Minor corrections to labs 6 and 7.
3.13	12/1/2015	DJH	Minor corrections to lab 1. Minor corrections to lab 7 Added GPIO output code to lab 7 for measure the ADC conversion time. Added lab 8.
3.14	4/25/2016	DJH	Minor corrections to labs 7 and 8. Added measurement of ADC conversion time to lab 7.
3.15	8/30/2016	DJH	Minor corrections to lab 8. Updated the Keil tool installation and KL25Z firmware load procedures.
3.16	1/17/2017	DJH	Minor edits to lab 4.