Project Report: ASL Glove

11/30/2021

Team Members:
Haritha Patil, Indraja Venigandla, Sangam Shrestha, and Sumati Gupta

Table of Contents

Introduction **4**
- Problem Description 4
- Project Description 4

Background **4**
- Theory of voltage divider and Operation of Flex Sensors 4
- Theory of UART 5
- Theory of Bluetooth 6
- Theory of I2C and SPI 6
- Operation of HC-05 BT Module 7
- Theory of DMA 8

Individual aspects of the project **8**
- PIT 8
- ADC and DMA triggered ADC 8
- UART and HC-05 connections 9
- I2C, SPI and LCD Screen 9
- 16 - segment Display 11
- Calibration and Decode Software 12

Overall Block Diagram and Flowchart **14**

Results **17**
- Performance 17
- Limitations 17

Alternatives and/or future work **17**
- Parallel ADC readings 17
- Increase accuracy 17
- Currently, we take readings from only flex sensors and while this works well most of the time, there are some inaccuracies due to two factors. 17
- PCB 18
- Since using breadboards is bulky and the external wires can lead to loose connections, we would design and solder a printed circuit board for a more sturdy platform on which the glove operates. This would also lead to a more robust product, capable of being used in an easier manner. 18
- LCD Screen 18
- Power supply 18

Video Links showing different Stages of progress **18**

Summary **19**

Reference List (APA Citations) **19**

Appendices **20**
- Appendix A 20

Code for PIT source file 20
Code for PIT header file 21
Code for ADC source file 21
Code for ADC header file 23
Code for DMA source file 24
Code for DMA header file 26
Code for UART TX source file 26
Code for UART TX header file 31
Code for Mode detect source file 32
Code for Mode detect header file 33
Code for main source file on Transmission MCU 34
Code for main header file on Transmission MCU 35
Code for UART RX source file 35
Code for UART RX header file 39
Code for calibration and letter decode source file 39
Code for calibration and letter decode header file 54
Code for 16 segment source file 55
Code for 16 segment header file 60
Code for main source file on Reception MCU 60
Code for main header file on Reception MCU 61
Appendix B 62
Segments of MCU reference Manual 62
HC-05 Bluetooth Module datasheet segments 63
16-segment display datasheet segments 64
Appendix C 64
Information on data formats and slave devices for I2C. 65

Introduction

Problem Description

Most people have a hard time understanding American Sign Language (ASL). When they come across a situation where they need to communicate or understand sign language, they require a human or electronic translator. This leads to alienation and miscommunication between two groups of people.

To reduce such alienation, we propose a child-friendly educational tool which will help people get acclimated to and learn sign language.

Project Description

The aim of this project is to design a glove that can be used to teach the ASL alphabet. This glove will have flex sensors integrated into it, which will detect and keep track of various gestures for alphabets used in ASL and display them on a 16-segment display.

Background

Theory of voltage divider and Operation of Flex Sensors

When a voltage source is applied across two resistors in series (where one is connected directly to the source and the other is connected to ground), there is a voltage drop that can be measured across the resistor connected to ground. This voltage drop is the divided voltage and is often referred to as the V_{out} . The formula used V_{out} to calculate is:

$$\mathbf{V_{out} = V_{in} * \frac{R_2}{R_1 + R_2}}$$

, Given the setup in Figure 1

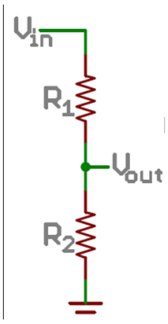


Figure 1: General Voltage Divider circuit *[1]*

The flex sensor is a resistor that changes resistance based on the bend of the sensor. We connect it in a similar manner to the voltage divider circuit *[2]*. The V_{out} is measured using the ADC peripheral of the Board. R1 is the flex sensor, R2 is 47kΩ or 51kΩ based on Table 1 and 2, Vin is 5 V output from the MCU and GND is GND from the board

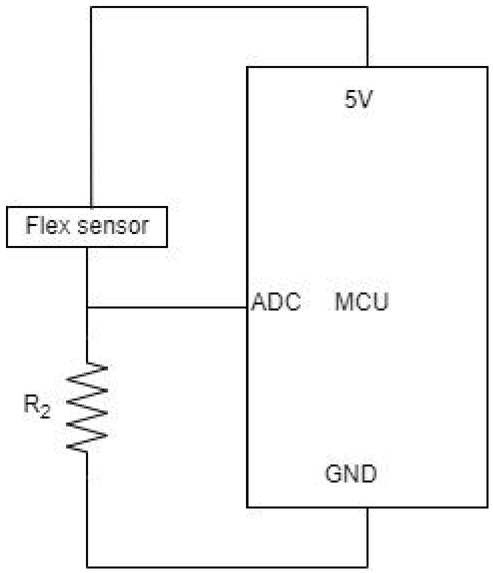


Figure 2: Flex sensor circuit

Resitor value	Standing up ADC value	Completely bent value	Opposite bend	Reason to not use/use
47k	A57x	65xx	B1xx	range gives us 16,480 values
1k	09Ax	357		range to small
no resistor	FCAx	F77x		range to small
2,2k	133x	06Bx	17Bx	
100k	C6Ax	94xx	D1xx	range gives us 809 values
51k	A33x	42xx	B1xx	range gives us 24,864 values
68k	B33x	53xx	C06x	ranges gives us 24,608 values

Table 1: The values from different resistors for small flex sensors. The highlighted one was determined as the best one to use. Here "x" means, it's fluctuating. Empty cells represent a range that was too small to take into account.

Resitor value	Standing up ADC value	Completely bent value	Opposite bend	Reason to not use/use
47k	C3xx	019x	D2Bx	range gives us 49,530 values
no resistor	FCxx	FCxx		range to small
30k	AACx	00Ax	BFxx	range gives us 43,552 values
51k	C0xx	89xx	D6xx	range gives us 14,080 values

Table 2: The values from different resistors for long flex sensors. The highlighted one was determined as the best one to use. Here "x" means, it's fluctuating. Empty cells represent a range that was too small to take into account

Theory of UART

UART stands for Universal Asynchronous Receiver Transmitter. It is a peripheral used for serial communication. The data format and baud rate is configurable using registers on the MCU. Based on configuration, it can be configured to be only a transmitter, only a receiver or both. The transmitter uses a shift register to convert input data to serial data and the receiver uses a shift register to convert serial data into parallel data.

Baud rate refers to the rate at which data is being transferred in bits per second. It is set using the formula, **Baud rate = bus clock frequency/(SBR * (OSR +1))**, where SBR can be set in BDH and BDL registers and OSR refers to the oversampling rate.

The oversampling rate is used to estimate the middle points of transmitted bits and retrieve these points accordingly *[3].* This means that each incoming bit is sampled OSR times and a majority vote is taken.

In our case, the format of data being sent is one start bit, one byte of data, one stop bit

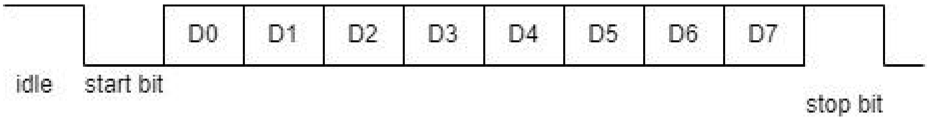


Figure 3: Data format being transmitted and received

Theory of Bluetooth

Bluetooth is a short-range wireless technology standard that is used for exchanging data . A Bluetooth device works by using radio waves instead of wires or cables. *[5]*

Theory of I2C and SPI

I^{2}C (Inter-Integrated circuit) bus and SPI (Serial peripheral interface) are used for synchronous serial communication. While I^{2}C can be used with one master or multiple masters and a single or multiple slave devices, SPI can work with only one master and one or multiple slave devices.

I^{2}C uses serial data (SDA) signal and serial clock (SCL) signal. A slave may not transmit data unless it has been addressed by the master. Each device on the I2C bus has a specific device address. In our case, since we attempted to work with only one LCD screen as the slave device, we did not need to configure it for multiple slave devices.

In our case, we tried to send data to a slave device (LCD screen and data format is as follows):

SINGLE-BYTE WRITE									
MASTER	START		SLAVE ADDRESS + WRITE		REGISTER ADDRESS		DATA		STOP
SLAVE				ACK		ACK		ACK	

Figure 4: Data format that was being sent to screen *[6]*

In SPI, The MCU is the master and peripheral devices are slaves. In our case, our slave device is the LCD controller. SPI communication between a master and slaves uses three signals(clock and two data signals), a select signal for each slave and ground.
- The clock signal (known as SCK).
- The MOSI data signal is the master output and slave input.
- The MISO data signal is the master input and slave output.
- The SS signal - Slave Select signal.

Baud rate refers to the rate at which data is being transferred in bits per second. It is set using the formula, **SPI baud rate = bus clock frequency/((SPPR+1)*2SPR+1)**, where SPPR and SPR can be set in the SPIx_BR register and **I2C baud rate = bus speed/(mulx SCL divider)** which can be sent in F register using Table 38-41 in reference Manual (See Appendix B)

Operation of HC-05 BT Module

The HC-05 has EN, VCC, GND, TX, RX, and STATE pins. The HC-05 Bluetooth (BT) Module has two modes “ AT Command " mode and operation mode. HC-05 Module’s are slaves by default and able to recieve data at the default baud rate and settings.

However to configure the module, we need to go into AT Command mode and configure details of the BT Module. To do this, we require an FTDI reader and the tera term software. We connected the FTDI reader (Micro USB to TTL Serial Converter Adapter) to the computer and once it was detected by the computer’s COM ports. After that, we connected the Rx pin of the FTDI reader to the Tx pin of the Module and the Tx pin to the Rx pin. After which we shorted the Vcc and EN pins of the HC-05 and then we connected the Vcc of the reader to BT Module. The LED on the Module blinked every two seconds indicating it was in AT mode *[4]*.

Once it was in AT mode, we used the tera term software to send data to the HC-05 and using the commands in the HC-05 Datasheet, we set the following configurations.

	HC-05 (Transmitter)	**HC-05 (Receiver)**
Role	Master	Slave
Baud rate	9600	9600
Stop bit	1	1
parity bit	0	0
Bind to	98D3:61:F5D8FF (address of slave)	slave can't bind to anything

Table 3: Configurations of HC-05 BT Module’s, Note: Master has to bind to slave address to ensure it always connects to that particular slave device. The address of the slave device was found using an AT command: AT+ADDR?

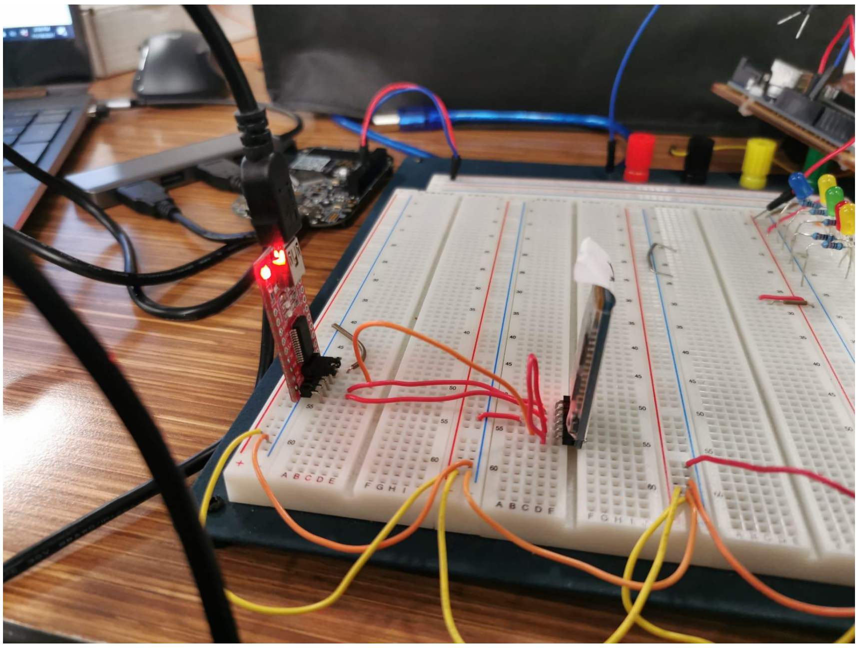


Figure 4: Connection between HC-05 and FTDI reader

Figure 5: Picture of Tera Term software

This process is how we ensure that the HC-05 Modules don't connect to any other bluetooth devices in the vicinity.

Theory of DMA

DMA stands for Direct Memory Access. This peripheral can directly access the MCU's memory bus directly instead of relying on store and load instructions. Different peripherals can trigger a DMA request. In our case, each ADC conversion triggers a DMA request.

Individual aspects of the project

PIT

The timer PIT is initialized such that 2 channels are utilized. Channel 0 is used for polling adc values every 10 ms by loading 5DBF (10ms*24Mhz - 1 = 23999 cycles or 5DBF) in the LDVAL register for channel 0, and calling a function that starts ADC conversion in the IRQHandler.

Channel 1 is used for updating the flag that indicates mode of operation of the program based on a switch input. Mode of operation refers to calibration mode or performance mode. Once the switch is pressed, it will update every 2 seconds until calibration is over. This is done by loading 2DC6BFF (2*24Mhz - 1 = 47999999 cycles or 2DC6BFF) in LDVAL register for channel 1.

Neither channel uses chaining. The IRQHandler uses the TIF flag in the TFLG register to check which channel triggered the interrupt. The timer restarts when calibration mode is set to avoid unsynchronized data transmission. See Appendix A for code.

ADC and DMA triggered ADC

The ADC is initialized such that there are 5 ADC pins and it is DMA enabled. It uses a 16-bit mode and uses a calibration function after each conversion for accuracy purposes. Once an ADC conversion is over, it triggers the DMA and the DMA transfers the ADC value into an array called GloveInputs. This is done by setting the DMA source to ADC, setting the Source address register in DMA to the result register of the ADC and the Destination address register to the array. We use DMA in circular buffer mode. We increment the destination address, making sure it stores the values sequentially in the correct index value of GloveInputs. Each time a transfer of 16 bits is over, the DMA interrupt occurs, indicating it's done. We have a counter to keep track of this. See Appendix A for code.

UART and HC-05 connections

We used AT mode as described in "Operation of HC-05" in the Background Section.We use UART0 and UART1. We connected to the UART0 TX pin of the Transmitter MCU (reading glove inputs) to RX of master HC-05 and the UART0 RX pin of the Receiver MCU (displaying the values) to the UART0 TX of the slave HC-05.The UART1 TX pin of the Transmitter MCU is connected to the UART1 RX pin of the Receiver MCU.

We configured both baud rates to 9600 in 8 bit mode. There is one stop and no parity bit.

The Transmitter MCU sends five 16 bit values of ADC and one calibration flag 8 bits at a time using polling once the DMA finishes populating all the elements in GloveInputs. After the values in the array are transmitted, transmission stops until DMA is triggered again. UART1 transmits "1" when UART0 is transmitting valid data and "0" when not.

The receiver MCU interrupts whenever it receives a UART1 transmission, if the value is not 0, it will consider UART0 values as valid inputs and populate an array called "receivedADCread" and decodes the calibration flag to find out mode of operation. The way it populates this array is, it combines two 8 bit values in one 16 bit value and places it an appropriate index (index indicates which finger). See Appendix A for code.

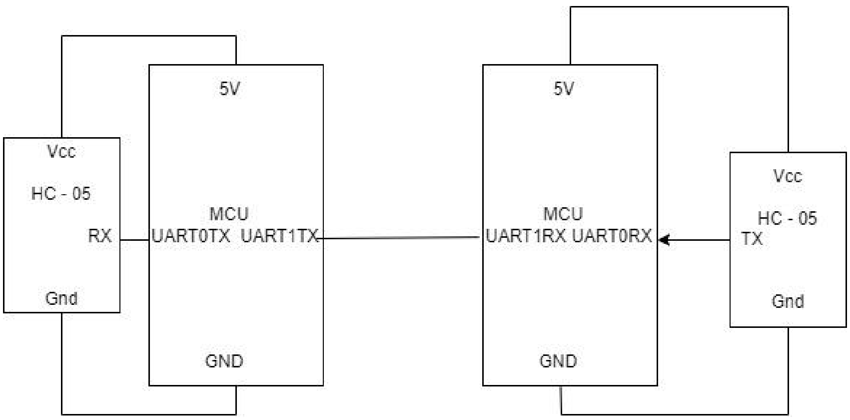


Figure 6: UART schematic

I2C, SPI and LCD Screen

The SPI is initialized and verified using the SPI loopback test where we figured out the SPI is initialized and the MOSI is sending the data. The figure 5 shows the circuit connection for the SPI initialization. Later, we tried initializing the LCD using the LCD datasheet but couldn't initialize the LCD. The SPI was not sending data to the SPIx_D register. After various consultations with course staff, we moved on to an I^2C LCD screen.

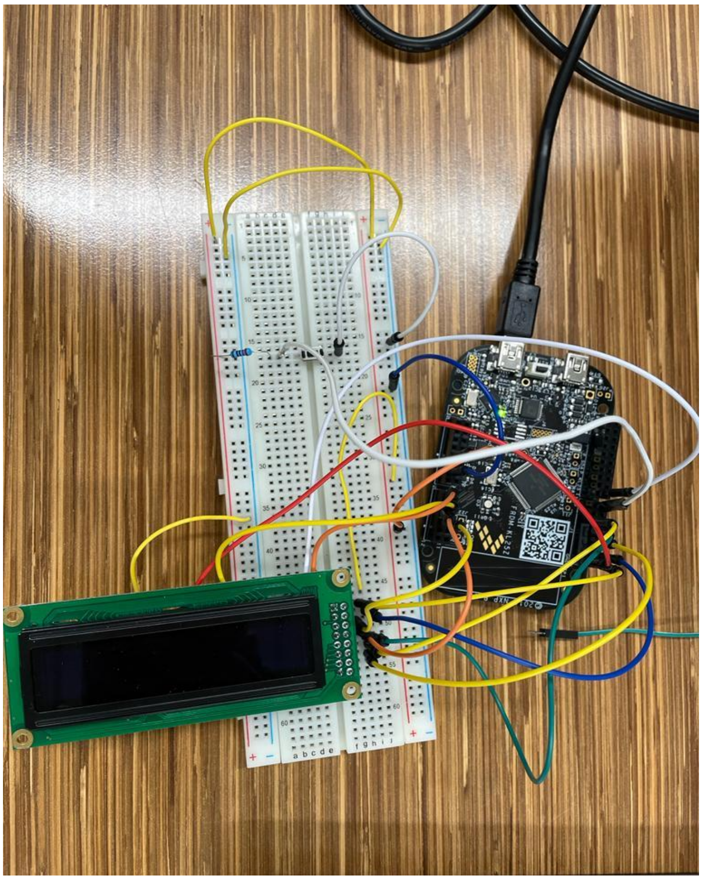


Figure 7: Circuit connection for SPI Initialization

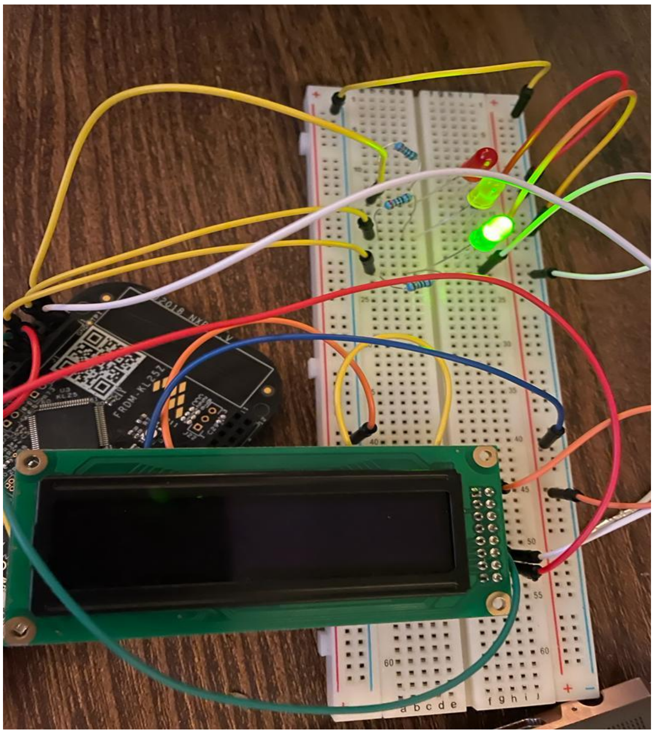


Figure 8: Circuit connection for SPI loopback test

The I2C is initialized and performs the LCD initialization. During LCD initialization, the I2Cx_D register is getting updated but the display doesn't initialize . The figure 9 shows the circuit connection for I2C initialization. Attached is the debug screen shot for the reference as the figure 10.

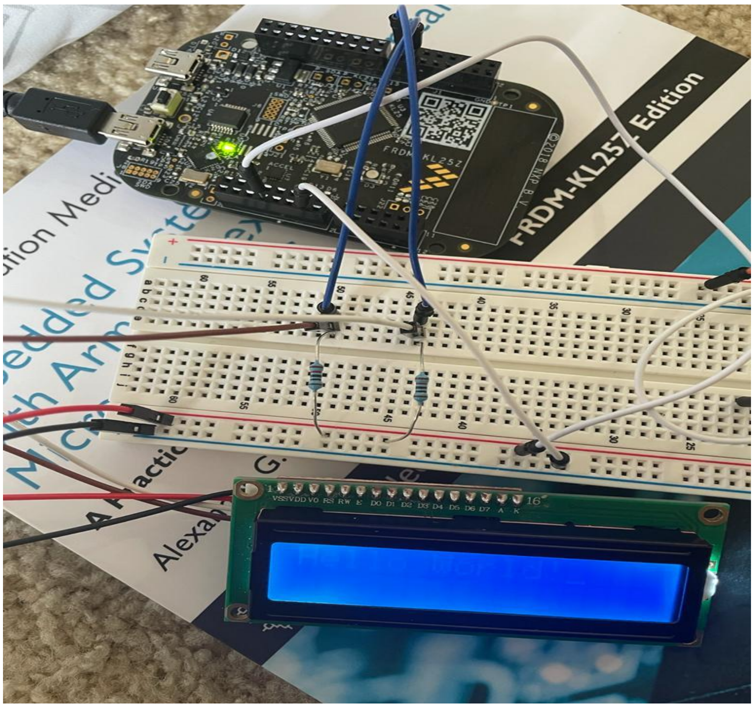


Figure 9: Circuit connection for Initialization of I2C

I2C0

Property	Value
A1	0
F	0x11
C1	0xB0
S	0x85
D	0xFF
DATA	0xFF
C2	0x20
FLT	0x40
RA	0
SMB	0
A2	0xC2
SLTH	0
SLTL	0

Figure 10: Debug screen showing D register getting updated

Unfortunately within the scope of our course (the time available), we are unable to figure out the issue. So, we moved on to a 16 segment display.

16 - segment Display

The 16 segment display works with the GPIO. The 16 segments of LED are designated as a,b,c,d,e,f,g,h,k,m,n,p,s,r,t,u and see the figure 11 for reference. We connected all the GPIO ports of the MCU configured as outputs. The segment display receives the input from the decode function and activates the corresponding GPIO ports and displays the output according to that. See Appendix A for code.

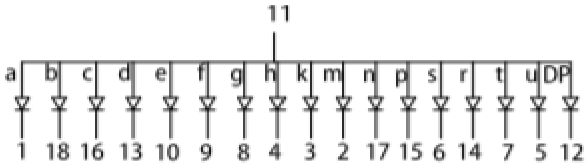


Figure 11: 16 Segments of LED

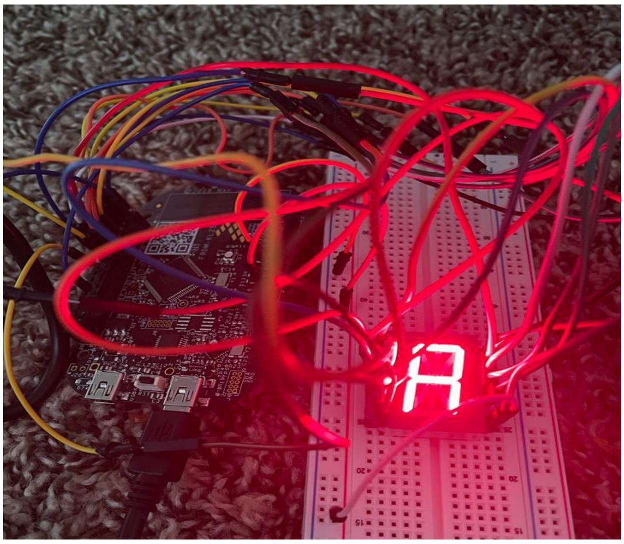


Figure 12: Circuit connection for 16 Segment display

Calibration and Decode Software

The calibration process is a way to ensure that each set of values read by the ADC can be reliably, and repeatedly mapped to the different letters in the ASL. After the calibration is done, any values read by the ADC needs to be decoded so that the glove knows what letter to display. Both the calibration and decoding procedures are explained in this section:

Calibration

Since the system needs to be able to repeatedly display the same letters given a certain combination of flex sensor readings, it is important to first establish a reference with respect to which we can measure the flexion of the flex sensors. We anticipated that there are three base references that we must establish: the range of values which corresponds to the flex sensor being in the "standing up" position, the range of values which corresponds to the flex sensors being in the "bent" position, and the range of values which corresponds to the flex sensors being in a position that is intermediate to the previous two. This position is referred to as the “middle” position.

This process has a user interface. When the green LED on the freedom board lights up, this means that the MCU is calibrating for the “standing” position, and so the user should keep their hand as straight as possible, so that there is minimal flexion impressed on the flex sensors. After some time, the LED turns red which signifies that the MCU is calibrating for the “bent” position. In this stage, the user should bend his/her fingers so that maximum flexion is impressed on the sensors. After some time, the LED turns off and this means that the user should return to the standing position.The “middle” position is taken as the arithmetic average of the standing and bent positions.

The generated values above are from the user. However, to determine a range of values over which the flex sensor is considered to be in one of standing, middle or bent positions, we need another set of values to which we compare the values generated by the user following the above procedure. This new set of values has been embedded into gloves itself at manufacturing time. We have individually tested each flex sensor and recorded what values they read when configured in the standing, middle and bent positions.

The ranges for the standing, middle and bent positions, then, can be calculated as the difference of the user-generated values and the embedded values. In this way, whenever the ADC detects a particular set of values, it will fall in one of these ranges and we can then use a look-up table (explained below) to determine which letter corresponds to the detected set of values.

Decoding

The look-up table is the data structure which helps in the decoding process. This table is a 2-dimensional array (it contains 26 by 5 elements) which takes the 26 letters of the alphabet and mandates what combination of 5 ADC/flex sensor values (one from each finger) should correspond to that letter.

Once this array is established, then any set of values read by the ADC can be corresponded to a letter and will be displayed on the 16-segment display.

The code to execute the calibration and decoding processes are given the appendix.

Overall Block Diagram and Flowchart

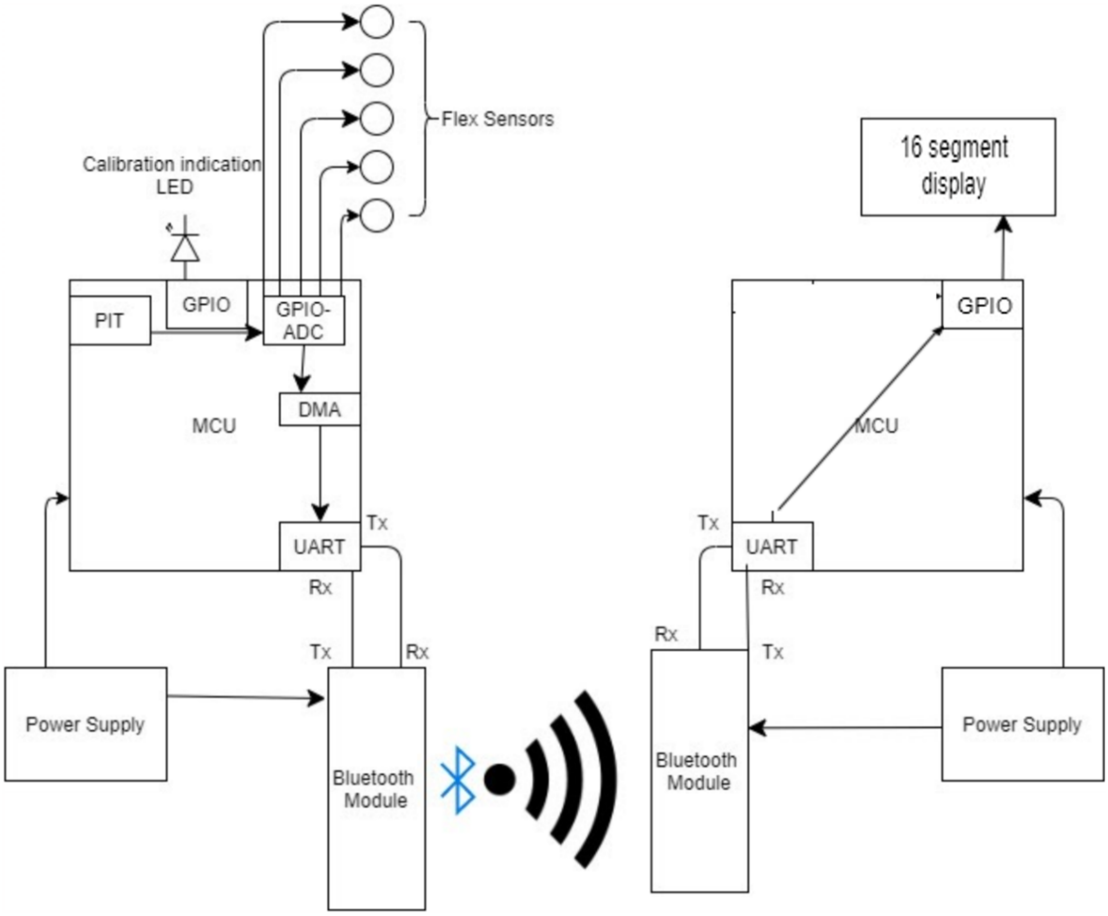


Figure 13: Block Diagram for overall product

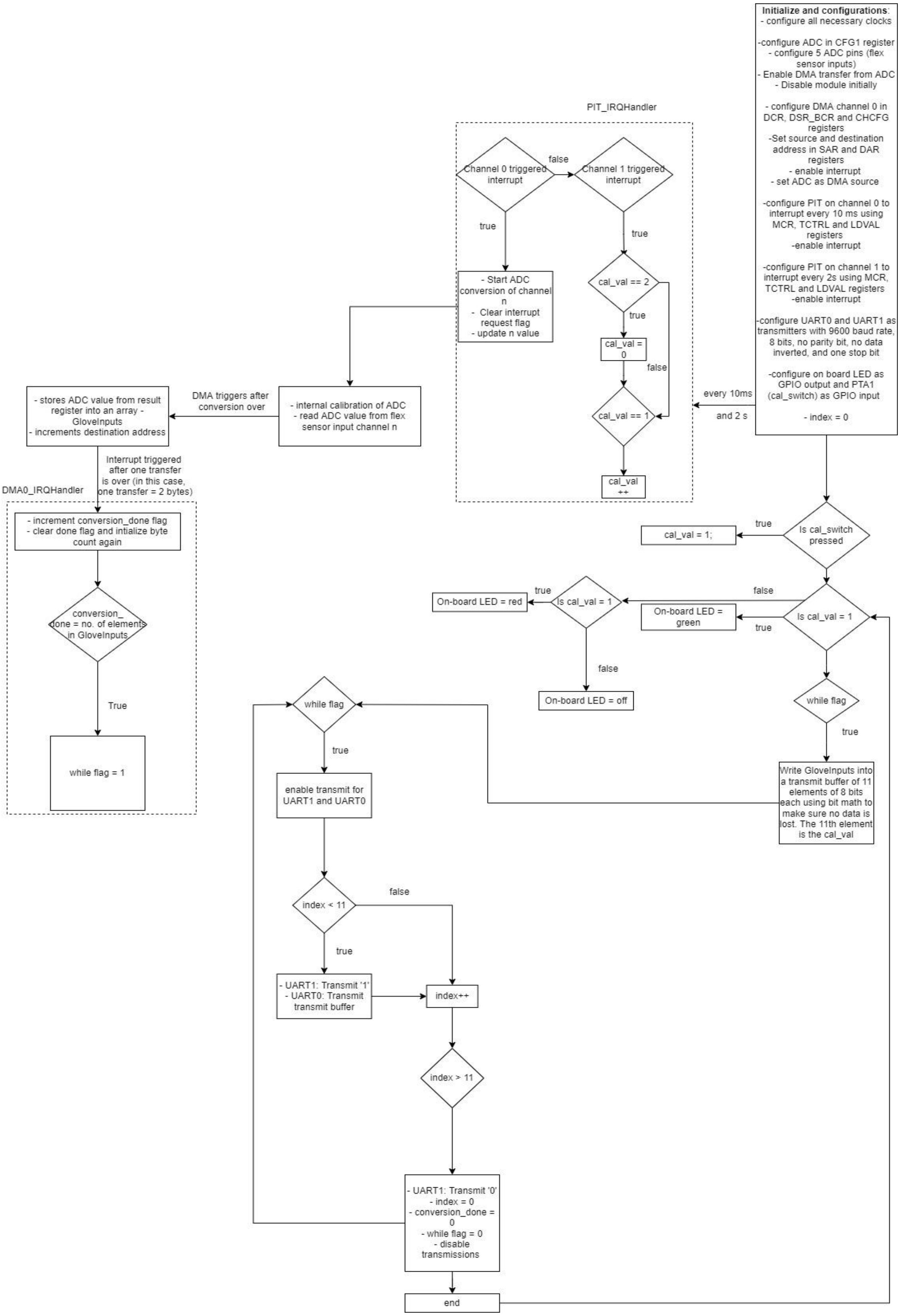


Figure 14: Overall Flowchart of Transmitter MCU

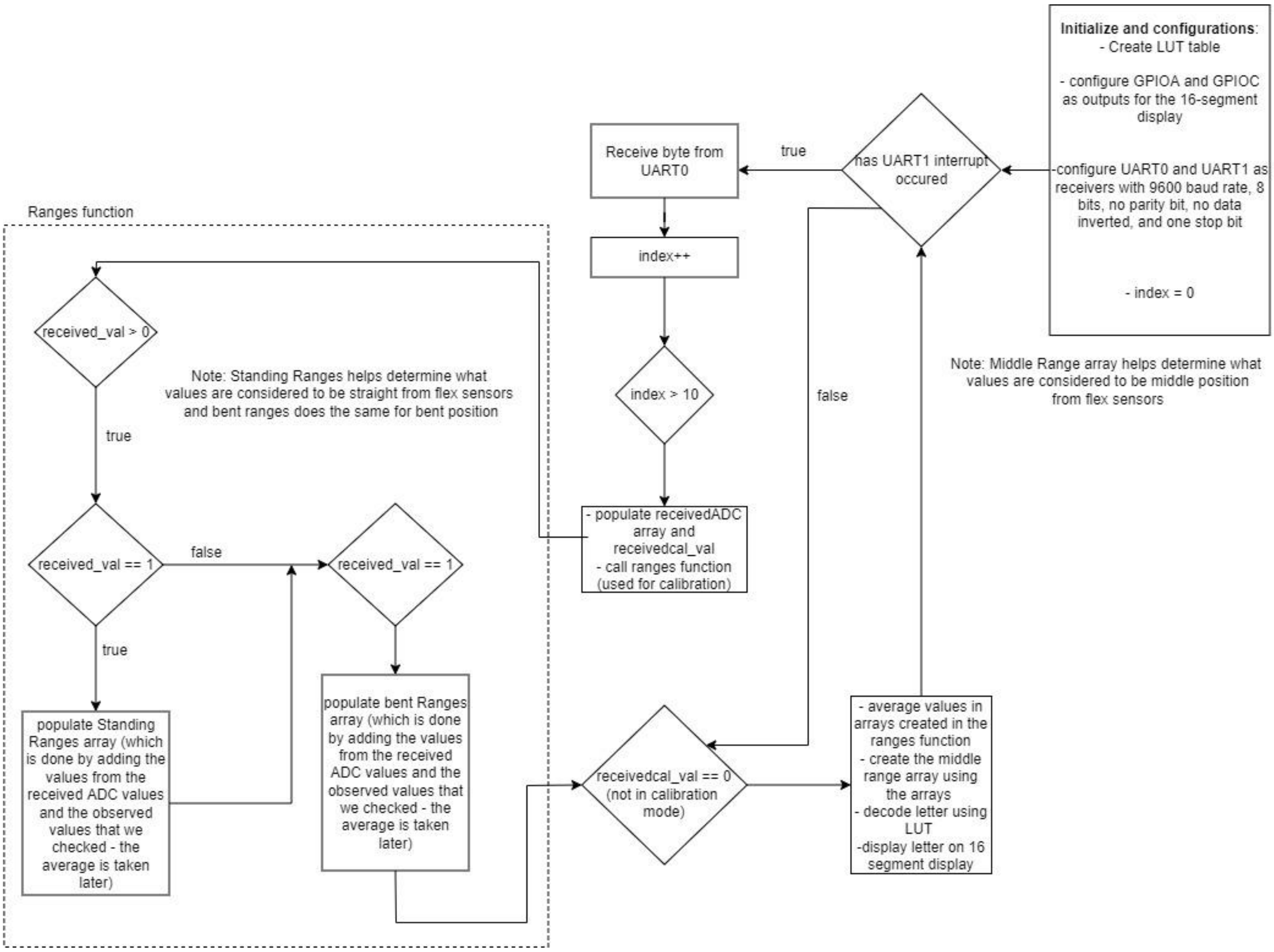


Figure 15: Overall Flowchart Receiver MCU

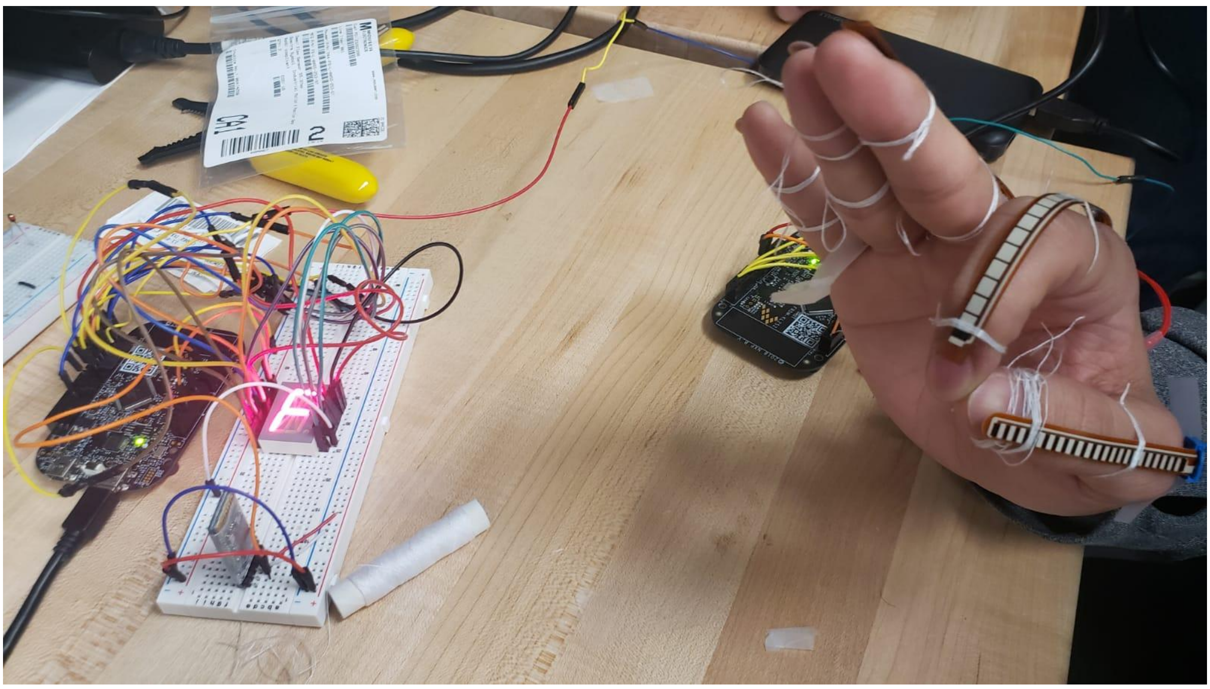


Figure 16: Photo of project in use

Results

Performance

The timers, ADC, DMA, UART, Calibration and letter decoding functions work as expected and the project displays the letter on the segment as expected. When we sign a letter with the flex sensors attached, it is reflected on the screen.

Limitations

1. The flex sensors aren't stabilized when they are attached to the hand which leads to fluctuating values and the need to be precise in hand positions..
2. Since we can't determine movement in the fingers, some letters are hard to differentiate. For example the signs for "i" and "j" are the same, except that there is "J" movement with the pinky for "j".

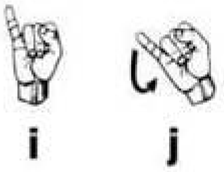


Figure 17: ASL Letters for i and j

Alternatives and/or future work

Parallel ADC readings

Currently, we sequentially read the ADC values from various pins instead of in a parallel manner. This decision was made after discussion with course staff. However, in the future, it would be efficient if we were able to read it in parallel.

Increase accuracy

Currently, we take readings from only flex sensors and while this works well most of the time, there are some inaccuracies due to two factors.

One factor is that the flex sensors aren't stabilized when they are attached to the hand and the second factor is that certain ASL letters don't depend on only the bend angle of the fingers but also movement of fingers. In order to pinpoint the exact location of the hand, we would need to include a gyroscope or an accelerometer.

PCB

Since using breadboards is bulky and the external wires can lead to loose connections, we would design and solder a printed circuit board for a more sturdy platform on which the glove operates. This would also lead to a more robust product, capable of being used in an easier manner.

LCD Screen

We are currently using a 16-segment display. We would like to incorporate an LCD Screen in the future in order to reduce the amount of wires/connections necessary. An LCD screen utilizing the I2C communication protocol only requires 4 wires to power up and use, compared to the 16 connections required for a 16-segment display. Another benefit of the LCD screen is that it is more scalable compared to the 16-segment display. An immediate expansion to the product is the incorporation of words and phrases. An LCD screen would easily allow the user to see what is being said when a certain gesture is made.

Power supply

We are currently using power banks to power the MCUs. We would want to switch to battery packs to make it more compatible with the current market. Since, the 5 V output from the MCU only works when it is connected via USB, this would require a voltage regulator set-up to work with a battery pack.

Video Links showing different Stages of progress

---

1. DMA working with ADC inputs - https://youtu.be/g7CzWs6c_rA
2. Configuring HC-05 in AT mode - https://youtu.be/fyI2CLJY3aA
3. UART working with HC-05 transmitting one byte - https://youtu.be/caIVNzS3L14
4. UART working with HC-05 transmitting the whole buffer. - https://youtu.be/u_HvwvnnR60
5. Calibration process using the glove - https://youtu.be/7QInGuYTodQ
6. Letter "E" displayed based on flex sensors- https://youtu.be/5Dz7n6EHO3E
7. Letter “Z” displayed based on flex sensors - https://youtu.be/-eKPcRNBViQ
8. Letter "I" displayed based on finger positions - https://youtu.be/3a8Pn2bjF5I
9. Letter “G” and “H” fluctuating based on finger positions - https://youtu.be/s1mzhiopUj8
10. Letter "L" displayed based on finger positions - https://youtu.be/M8MpVGRHP30
11. Letter "D" displayed based on finger positions - https://youtu.be/m7DSW6J3_eg
12. Letters "A", "B", and "C" displayed based on finger positions - https://youtu.be/_WVLcP_ZgN8

Summary

The ASL glove is a product designed to help interested individuals in learning the basics of ASL. This device makes use of several peripherals found in the MKL25Z128VLK4 microcontroller. These peripherals were: the UART0 and UART 1 modules, the ADC, the DMA, the PIT, and the GPIOs.

The overall program flow of the device is as follows:

The PIT triggers an interrupt every 10 ms which signals the ADC to read the values from the GPIO pins. Upon the completion of every conversion, the ADC requests the DMA to save the value into an array in memory. Once the array is filled with all the sampled values from all the GPIO pins, the UART1 module sends a random character to the receiver. Once the receiver receives the character, it triggers an interrupt which signals the UART0 module to start polling for the array. The transmitter transmits the array to the receiver, and the receiver uses its look-up table to decode and display the proper letter to the 16-segment display.

Reference List (APA Citations)

1. *Voltage Dividers*. Sparkfun. (n.d.). Retrieved November 30, 2021, from https://learn.sparkfun.com/tutorials/voltage-dividers/all
2. *Flex Sensor Hookup Guide*. Sparkfun. (n.d.). Retrieved November 30, 2021, from https://learn.sparkfun.com/tutorials/flex-sensor-hookup-guide/all#example-circuit
3. *UART (universal asyncronous receiver & transmitter)*. Weebly. (n.d.). Retrieved November 30, 2021, from https://robo-tronix.weebly.com/uploads/2/3/2/1/23219916/uart_design_doc.pdf
4. sayem2603, & Instructables. (2017, October 14). *At command mode of HC-05 and HC-06 Bluetooth module*. Instructables. Retrieved November 30, 2021, from https://www.instructables.com/AT-command-mode-of-HC-05-Bluetooth-module/
5. *How does a bluetooth headset work?* Jabra. (n.d.). Retrieved November 30, 2021, from https://www.jabra.com/fq/how-does-bluetooth-headset-work#:~:text=A%20Bluetooth%C2%AE%20device%20works,cell%20phone%2C%20smartphone%20or%20computer.&text=A%20single%20Bluetooth%20headset%20can,bother%20of%20wires%20or%20cords
6. Noel, P. A. (n. d.). Lecture on Serial Communications Chapter 8 by Dr Alexander G Dean. Personal Collection of P. A. Noel II, Oakland University, Rochester Hills, MI.

Appendices

Appendix A

Code for PIT source file

```c
#include "pit.h"

int counter = 0;

void pit_init(void)
{
  // Enable PIT clock
  SIM->SCGC6 |= SIM_SCGC6_PIT_MASK;


  // Turn on PIT - clear MDIS bit in MCR reg
  PIT->MCR &= ~PIT_MCR_MDIS_MASK;
  //freeze timer in debug mode
  PIT->MCR |= PIT_MCR_FRZ_MASK;

  // Configure PIT to produce an interrupt every 1s on Timer 0
  PIT->CHANNEL[0].LDVAL  = PIT_LDVAL_TSV(0x5DBF); // 1ms*24Mhz - 1 = 23999 cycles or 5DBF
  // Enable interrupt and enable timer
  PIT->CHANNEL[0].TCTRL  |= PIT_TCTRL_TIE_MASK | PIT_TCTRL_TEN_MASK;
  //set "no chaining"
  PIT->CHANNEL[0].TCTRL &= ~PIT_TCTRL_CHN_MASK;

  // Configure PIT to produce an interrupt every 2s on Timer 1
  PIT->CHANNEL[1].LDVAL  = PIT_LDVAL_TSV(0x2DC6BFF);  // 2*24Mhz - 1 = 47999999 cycles or 2DC6BFF
  // Enable interrupt and enable timer
  PIT->CHANNEL[1].TCTRL  |= PIT_TCTRL_TIE_MASK | PIT_TCTRL_TEN_MASK;
  //set "no chaining"
  PIT->CHANNEL[1].TCTRL &= ~PIT_TCTRL_CHN_MASK;


  // Enable interrupt
  NVIC_SetPriority(PIT_IRQn, 0);
  NVIC_ClearPendingIRQ(PIT_IRQn);
  NVIC_EnableIRQ(PIT_IRQn);
  __enable_irq();

}

void PIT_IRQHandler(void)
{
  //Can chose to check channel int flag
  if(PIT->CHANNEL[0].TFLG & PIT_TFLG_TIF_MASK){
      adc_read(counter);
      counter += 1;
      if (counter == 8){
```

```
            counter = 0;
        }
        //Clear interrupt request flag for channel by writing one to it - slide 12 of chapter 7
        PIT->CHANNEL[0].TFLG |= PIT_TFLG_TIF_MASK;
    }

    else if(PIT->CHANNEL[1].TFLG & PIT_TFLG_TIF_MASK)
    {
        if (cal_val == 2)
            cal_val = 0;
        else if (cal_val == 1)
            cal_val++;

      //Clear interrupt request flag for channel by writing one to it - slide 12 of chapter 7
        PIT->CHANNEL[1].TFLG |= PIT_TFLG_TIF_MASK;
    }

}

```

Code for PIT header file

```
#include <MKL25Z4.h>
#include "adc.h"
#include "read.h"

extern int counter;

void pit_init(void);
void PIT_IRQHandler(void);

```

Code for ADC source file

```
#include "adc.h"

int CH_idx[8];

void adc_init(void)
{
  // Enable clocks
  SIM->SCGC6 |= SIM_SCGC6_ADC0_MASK;  // ADC0 clock
  SIM->SCGC5 |= SIM_SCGC5_PORTB_MASK; // PortB clock
  SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK; // Port E clock


  // Configure ADC
  ADC0->CFG1 = 0; // Reset register
  ADC0->CFG1 |= (ADC_CFG1_MODE(3)  |       // 16 bits mode
                 ADC_CFG1_ADICLK(0)|   // Input Bus Clock (20-25 MHz out of reset (FEI mode))
                 ADC_CFG1_ADIV(0)) |   // Clock divide by 1
                   ADC_CFG1_ADLPC_MASK //low power configuration
                   ; // short sample time

```

```
 //Select analog pin for 5 fingers
 PORTB->PCR[ADCpinky] &= ~PORT_PCR_MUX_MASK;
 PORTB->PCR[ADCpinky] |= PORT_PCR_MUX(0);
 PORTB->PCR[ADCring] &= ~PORT_PCR_MUX_MASK;
 PORTB->PCR[ADCring] |= PORT_PCR_MUX(0);
 PORTB->PCR[ADCmiddle] &= ~PORT_PCR_MUX_MASK;
 PORTB->PCR[ADCmiddle] |= PORT_PCR_MUX(0);
 PORTB->PCR[ADCindex] &= ~PORT_PCR_MUX_MASK;
 PORTB->PCR[ADCindex] |= PORT_PCR_MUX(0);
 PORTE->PCR[ADCthumb] &= ~PORT_PCR_MUX_MASK;
 PORTE->PCR[ADCthumb] |= PORT_PCR_MUX(0);

 ADC0->SC2 |= ADC_SC2_DMAEN_MASK;    // DMA Enable

 ADC0_SC3 = 0; // Reset SC3

 ADC0_SC1A |= ADC_SC1_ADCH(31); // Disable module initially. Change the channel value once we start using it.

 CH_idx[0] = ADCpinky_CH;
 CH_idx[1] = ADCring_CH;
 CH_idx[2] = ADCmiddle_CH;
 CH_idx[3] = ADCindex_CH;
 CH_idx[4] = ADCthumb_CH;
 CH_idx[5] = 0x10;
 CH_idx[6] = 0x10;
 CH_idx[7] = 0x10;

}

//Refer to page 494 to 495 for instructions on calibration procedure

int internal_calibration_adc(void)
{
 uint16_t cal_var = 0; // calibration variable

 ADC0_CFG1 |= (ADC_CFG1_MODE(3)  |   // 16 bit mode
             ADC_CFG1_ADICLK(1)| // Input Bus Clock divided by 2 (20-25 MHz out of reset (FEI mode) / 2)
             ADC_CFG1_ADIV(2)) ; // Clock divide by 4 (2.5-3 MHz)

 ADC0_SC3 |= ADC_SC3_AVGE_MASK |      // Enable HW average
             ADC_SC3_AVGS(3)   |      // Set HW average of 32 samples
             ADC_SC3_CAL_MASK;        // Start calibration process

 while(ADC0_SC3 & ADC_SC3_CAL_MASK); // Wait for calibration to end

 if(ADC0_SC3 & ADC_SC3_CALF_MASK)    // Check for successful calibration
     return 1;

 //Do the following procedure for both plus and minus sides
 cal_var += ADC0_CLPS + ADC0_CLP4 + ADC0_CLP3 +
          ADC0_CLP2 + ADC0_CLP1 + ADC0_CLP0;
```

```
cal_var = cal_var >> 2;
cal_var |= ADC_set_MSB;      // Set MSB
ADC0_PG = cal_var;
cal_var = 0;
cal_var += ADC0_CLMS + ADC0_CLM4 + ADC0_CLM3 +
         ADC0_CLM2 + ADC0_CLM1 + ADC0_CLM0;
cal_var = cal_var >> 2;
cal_var |= ADC_set_MSB; // Set MSB
ADC0_MG = cal_var;

return 0;
}

void adc_read(int i){

//clear coco to start conversion on for all channels - bits 4 - 0 in ADCx_SC1n selects the input channel
// call the calibration function before each conversion

internal_calibration_adc();
 ADC0->SC1[0] = CH_idx[i];
ADC0->SC1[0] &= ~ADC_SC1_COCO_MASK;

}

```

Code for ADC header file

```
#include <MKL25Z4.h>

#define ADCpinky               (0) //port B - ADC0_SE8, Channel: AD8, ADC Channel:01000
#define ADCring                (1) //port B - ADC0_SE9, Channel: AD9, ADC Channel:01001
#define ADCmiddle              (2) //port B - ADC0_SE12, Channel: AD12, ADC Channel:01100
#define ADCindex               (3) //port B - ADC0_SE13, Channel: AD13, ADC Channel:01101
#define ADCthumb               (20) //port E - ADC0_DP0/SE0, Channel: DAD0, ADC Channel:00000

#define ADCpinky_CH      (0x8)
#define ADCring_CH       (0x9)
#define ADCmiddle_CH     (0xc)
#define ADCindex_CH      (0xd)
#define ADCthumb_CH      (0x0)

#define ADC_set_MSB      (0x8000)

extern int CH_idx[8];

void adc_init(void);
void adc_read(int i);
int internal_calibration_adc(void);
```

Code for DMA source file

```
#include "dma.h"

//variable definitions
uint32_t  GloveInputs[8];
volatile int conversion_done = 0;
int while_flag = 0;

//test
#define CONVERSION_LED (5) //on port A
#define MASK(x) (1UL << (x))

void dma_init(void)
{
 // Enable clocks
 SIM->SCGC6 |= SIM_SCGC6_DMAMUX_MASK;
 SIM->SCGC7 |= SIM_SCGC7_DMA_MASK;

 // Config DMA Mux for ADC operation
 // Disable DMA Mux channel first
 DMAMUX0->CHCFG[0] = 0x00;

 //configure DMA_SAR to read the ADC_Rn
 //configure DMA_DAR to write in array
 DMA0->DMA[0].SAR = DMA_SAR_SAR((uint32_t *)(&(ADC0->R[0])));
 DMA0->DMA[0].DAR = DMA_DAR_DAR((uint32_t *)GloveInputs);


 // initialize byte count - 2 bytes at a time - 16 bits. 16 bits selected since ADC in 16 bit mode
 DMA0->DMA[0].DSR_BCR = DMA_DSR_BCR_BCR(2);

 //if we want to force one read at a time - this means two bytes is one read
 DMA0->DMA[0].DCR |= DMA_DCR_CS_MASK;

 DMA0->DMA[0].DCR |= (DMA_DCR_EINT_MASK|      // Enable interrupt each time transfer is over
               DMA_DCR_ERQ_MASK |      // Enable peripheral request
               DMA_DCR_SSIZE(2) |      // Set source size to 16 bits
               DMA_DCR_DINC_MASK|      // Set increments to destination address
               DMA_DCR_DMOD(2)  |      // Destination address modulo of 16 bytes
               DMA_DCR_DSIZE(2));      // Set destination size of 16 bits
               //set bit 24 to 0 (a requirement)
 DMA0->DMA[0].DCR &= ~DMA_DCR_SINC_MASK; //clears source increment

 DMA0->DMA[0].DCR |= DMA_DCR_EADREQ_MASK; // changed in office hours

```

```

// Enable interrupt
NVIC_SetPriority(DMA0_IRQn, 2);
NVIC_ClearPendingIRQ(DMA0_IRQn);
NVIC_EnableIRQ(DMA0_IRQn);


// Enable DMA channel and source - ADC0 = 40 based on Table 3-20: DMA request sources
DMAMUX0->CHCFG[0] |= DMAMUX_CHCFG_SOURCE(40); // Enable DMA channel and set ADC0 as source
DMAMUX0->CHCFG[0] |= DMAMUX_CHCFG_ENBL_MASK;

/*if ((DMA0->DMA[0].DAR >= GloveInputs) && (DMA0->DMA[0].DAR <= (GloveInputs + 5))){
        DMAMUX0->CHCFG[0] |= DMAMUX_CHCFG_ENBL_MASK;
}
else{
            //DMAMUX0->CHCFG[0] = 0x00;
            DMA0->DMA[0].DAR = DMA_DAR_DAR((uint32_t *)GloveInputs);
            DMAMUX0->CHCFG[0] |= DMAMUX_CHCFG_ENBL_MASK;
}*/

DMA0->DMA[0].DSR_BCR &= ~DMA_DSR_BCR_DONE_MASK;


}

void DMA0_IRQHandler(void)
{
conversion_done += 1;

//everytime conversion completes, clears BCR to 0, so we need to clear done flag and initialize byte count again
DMA0->DMA[0].DSR_BCR |= DMA_DSR_BCR_DONE_MASK; // Clear Done Flag - write to it to clear it
DMA0->DMA[0].DSR_BCR = DMA_DSR_BCR_BCR(2);  //initialize byte count again


if (conversion_done == 8)
{
    //UART stuff
    /*TransmitBufferCreate();
    while_flag = 1;
    UART0->C2 |= UART0_C2_TE_MASK;
    while(while_flag)
    {
        //UART0_Transmit_DMA(index);
        index++;
      if(index > 11) //if you change 11 to a 10, the uart might be on the cusp of sending/not sending
```

```
                                    //11 gives the uart plenty of time to reset the index, etc.
            {
                UART0->C2 &= ~UART0_C2_TE_MASK; //disable interrupt once all values are transferred
                index = 0;
                while_flag = 0; //stops while loop
            }
        }*/
        while_flag = 1;
        //conversion_done = 0;
    }


}

```

Code for DMA header file

```
#ifndef DMA_H
#define DMA_H

#include <MKL25Z4.h>
#include "UART.h"
#include "adc.h"
#include "pit.h"

#define pinky_index (0)
#define ring_index (1)
#define middle_index (2)
#define index_index (3)
#define thumb_index (4)

//variable declarations
extern volatile int conversion_done;
extern uint32_t  GloveInputs[8];
extern int while_flag;
extern volatile int conversion_done;
void dma_init(void);
void DMA0_IRQHandler(void);

#endif

//Note: added ifndef and endif to get rid of "include nested too deeply"
//error
```

Code for UART TX source file

```
#include "UART.h"

//UART TX
int shiftedpinky = 0;
int shiftedring = 0;
int shiftedmiddle = 0;
int shiftedindex = 0;
```

```
int shiftedthumb = 0;
//uint8_t cal_val = 0;
uint8_t transmitbuffer[11];
//uint32_t  GloveInputs[8];

//Debug
unsigned char Transmit_Done = 0;


void Init_UART(void)
{
 // Let's use UART0 and PTB
 // Enable clock gating for UART0 and Port B

 uint16_t sbr;

 //We use transmit data to LCD screen and maybe eventually receive (for error check)
 //Since the UART can only take 8 bits, and we have 5 important 16 bit values that
 //need to be sent, we would have to fill D reg with each half of an array element
 //making it a total of 5*2 times D needs to be updated.
 SIM->SCGC4 |= SIM_SCGC4_UART0_MASK;
 SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK;

 // Make sure transmitter and receiver are disabled before configuring
 UART0->C2 &= ~UART0_C2_TE_MASK & ~UART0_C2_RE_MASK;

 // UART 1 has to use Bus Clock, clock can't be configured
 // according to table 5-2 (page 122) of reference manual
 // and absence of macors
 // but UART0 clock needs to be set - to 48 MHz clock in this case
 SIM->SOPT2 |= SIM_SOPT2_UART0SRC(1);
 //SIM->SOPT2 |= SIM_SOPT2_PLLFLLSEL_MASK;

 // Set pins to UART0 Rx and Tx
 PORTA->PCR[TX_PIN] |= PORT_PCR_MUX(2); // Tx


 // Set baud rate and oversampling ratio
 sbr = (uint16_t)((CLOCK)/(BAUD_RATE * OVERSAMPLE_RATE)) - 32 ;
 UART0->BDH &= ~UART0_BDH_SBR_MASK;
 UART0->BDH |= UART0_BDH_SBR(0x1F & (sbr>>8));
 UART0->BDL = UART0_BDL_SBR(sbr);
 UART0->C4 |= UART0_C4_OSR(OVERSAMPLE_RATE - 1); //UARTLP_C4_OSR(x) is same as UART0_C4_OSR(x) and since C4 reg all UART is same, it should work

 //LIN break is high whenever a break is detected.
 //I believe this occurs when there is empty space in a transmission or recieve line
 //this will set LBKDIF bit in UARTx_S2 as a high.
 //setting LBKDIE bit in BDH reg will cause an interrupt if LBKDIF is high
```

```
UART0->BDH |= UART0_BDH_LBKDIE(0);  // for now let's disable it
UART0->BDH |= UART0_BDH_RXEDGIE(0); // Disable interrupts for RX active edge
UART0->BDH |= UART0_BDH_SBNS(0);   // one stop-bit - only two options 1 or 2
                      // Its logic level is the same as the signal's idle state, i.e., logic high
UART0->C1 |= UART0_C1_M(0); //Data bit mode - 8 bits - In 8-bit data mode, the shift register holds a
                     //start bit, eight data bits, and a stop bit.
 UART0->C1 |= UART0_C1_LOOPS(0); //0: Normal operation - RxD and TxD use separate pins.
UART0->C1 |= UART0_C1_PE(0);    //Don't use parity bit
                        //We can use parity bit but will need to change bit mode to 9 - The 9-bit data mode is typically used with parity to allow eight bits of data plus the parity in the ninth bit
                        //When parity is enabled, the bit immediately before the stop bit is treated as the parity bit
                        // a little confused about D being only 8 bits long
                        // will also need to set PT bit in C1 to parity odd or even and enable interrupt for parity (PEIE) in C3

// Don't enable interrupts for errors
UART0->C3 = UART0_C3_ORIE(0)| UART0_C3_NEIE(0) | UART0_C3_FEIE(0) | UART0_C3_PEIE(0);

// Clear error flags
UART0->S1 |= UART0_S1_OR(1) | UART0_S1_NF(1) | UART0_S1_FE(1) | UART0_S1_PF(1); //To clear these flags write logic one to them

    UART0->S2 = UART0_S2_MSBF(0); //do not invert received data
UART0 -> C3 = UART0_C3_TXINV(0); //Send LSB first, do not invert transmitted data


//POSSIBLE NEED IN CASE OF RECEIVER
//If Receive Data Register Full Flag (RDRF bit) in UARTx_S1 is high and
//setting RIE bit in C2 reg will cause an interrupt - clear flag initially at end of init (based on textbook)
//UART0->C2 |= UART_C2_RIE_MASK;

UART0->C2 |= UART0_C2_TE(1); // Enable UART transmitter
}

void Init_UART1(void)
{
// Let's use UART0 and PTB
// Enable clock gating for UART0 and Port B

uint16_t sbr;

//We use transmit data to LCD screen and maybe eventually receive (for error check)
```

```
//Since the UART can only take 8 bits, and we have 5 important 16 bit values that
//need to be sent, we would have to fill D reg with each half of an array element
//making it a total of 5*2 times D needs to be updated.
SIM->SCGC4 |= SIM_SCGC4_UART1_MASK;
SIM->SCGC5 |= SIM_SCGC5_PORTC_MASK;

    // Make sure transmitter and receiver are disabled before configuring
    UART1->C2 &= ~UART_C2_TE_MASK & ~UART_C2_RE_MASK;

    // UART 1 has to use Bus Clock, clock can't be configured
    // according to table 5-2 (page 122) of reference manual
    // and absence of macors
    // but UART0 clock needs to be set - to 48 MHz clock in this case
    //SIM->SOPT2 |= SIM_SOPT2_UART0SRC(1);
    //SIM->SOPT2 |= SIM_SOPT2_PLLFLLSEL_MASK;

    // Set pins to UART0 Rx and Tx
    PORTC->PCR[4] |= PORT_PCR_MUX(3); // Tx


    // Set baud rate and oversampling ratio
    sbr = (uint16_t)((CLOCK)/(BAUD_RATE * 16)) - 32 ;
    UART1->BDH &= ~UART_BDH_SBR_MASK;
    UART1->BDH |= UART_BDH_SBR(0x1F & (sbr>>8));
    UART1->BDL = UART_BDL_SBR(sbr);
    //UART1->C4 |= UARTLP_C4_OSR(OVERSAMPLE_RATE - 1); //UARTLP_C4_OSR(x) is same as UART0_C4_OSR(x) and since C4 reg all UART is same, it should work

    //LIN break is high whenever a break is detected.
    //I believe this occurs when there is empty space in a transmission or recieve line
    //this will set LBKDIF bit in UARTx_S2 as a high.
    //setting LBKDIE bit in BDH reg will cause an interrupt if LBKDIF is high
    UART1->BDH |= UART_BDH_LBKDIE(0);  // for now let's disable it
    UART1->BDH |= UART_BDH_RXEDGIE(0); // Disable interrupts for RX active edge
    UART1->BDH |= UART_BDH_SBNS(0);   // one stop-bit - only two options 1 or 2
                      // Its logic level is the same as the signal's idle state, i.e., logic high
    UART1->C1 |= UART_C1_M(0); //Data bit mode - 8 bits - In 8-bit data mode, the shift register holds a
                     //start bit, eight data bits, and a stop bit.
  UART1->C1 |= UART_C1_LOOPS(0); //0: Normal operation - RxD and TxD use separate pins.
    UART1->C1 |= UART_C1_PE(0); //Don't use parity bit
                        //We can use parity bit but will need to change bit mode to 9 - The 9-bit data mode is typically used with parity to allow eight bits of data plus the parity in the ninth bit
```

```
                        //When parity is enabled, the bit immediately before the stop bit is treated as the parity bit
                        // a little confused about D being only 8 bits long
                        // will also need to set PT bit in C1 to parity odd or even and enable interrupt for parity (PEIE) in C3

    // Don't enable interrupts for errors
    UART1->C3 = UART_C3_ORIE(0)| UART_C3_NEIE(0) | UART_C3_FEIE(0) | UART_C3_PEIE(0);

    // Clear error flags
    //UART1->S1 |= UART_S1_OR(1) | UART_S1_NF(1) | UART_S1_FE(1) | UART_S1_PF(1); //To clear these flags write logic one to them

    UART1->S2 &= ~UARTLP_S2_MSBF_MASK; //do not invert recieved data
    UART1 -> C3 = UART_C3_TXINV(0); //Send LSB first, do not invert transmitted data


    //POSSIBLE NEED IN CASE OF RECEIVER
    //If Receive Data Register Full Flag (RDRF bit) in UARTx_S1 is high and
    //setting RIE bit in C2 reg will cause an interrupt - clear flag initially at end of init (based on textbook)
    //UART0->C2 |= UART_C2_RIE_MASK;

    UART1->C2 |= UART_C2_TE(1); // Enable UART transmitter
}

void UART0_Transmit_DMA(int index) {

        while (!(UART0->S1 & UART0_S1_TDRE_MASK))
            ;
        UART0->D = (uint8_t)(transmitbuffer[index]);

        if (!(UART0->S1 & UART0_S1_TDRE_MASK))
            Transmit_Done++;
}

void UART1_Transmit(int value)
{
    while (!(UART1->S1 & UART_S1_TDRE_MASK))
            ;
        UART1->D = (uint8_t)(value);
}

void UART0_Transmit_OneByte() {

        while (!(UART0->S1 & UART0_S1_TDRE_MASK))
            ;
        UART0->D = (uint8_t)(0x11);

```

```
        if (!(UART0->S1 & UART0_S1_TDRE_MASK))
            Transmit_Done++;


}

//function will be in UARTTX
void TransmitBufferCreate(){

    shiftedpinky = GLOVEMASK(GloveInputs[pinky_index]);
    shiftedring = GLOVEMASK(GloveInputs[ring_index]);
    shiftedmiddle = GLOVEMASK(GloveInputs[middle_index]);
    shiftedindex = GLOVEMASK(GloveInputs[index_index]);
    shiftedthumb = GLOVEMASK(GloveInputs[thumb_index]);

    transmitbuffer[0] = LSB(shiftedpinky);
    transmitbuffer[1] = MSB(shiftedpinky);
    transmitbuffer[2] = LSB(shiftedring);
    transmitbuffer[3] = MSB(shiftedring);
    transmitbuffer[4] = LSB(shiftedmiddle);
    transmitbuffer[5] = MSB(shiftedmiddle);
    transmitbuffer[6] = LSB(shiftedindex);
    transmitbuffer[7] = MSB(shiftedindex);
    transmitbuffer[8] = LSB(shiftedthumb);
    transmitbuffer[9] = MSB(shiftedthumb);
    transmitbuffer[10] = cal_val;
}

//Note: took out unneeded variables in .h file and .c file that was
//meant for only RX
```

Code for UART TX header file

```
#ifndef UART_H
#define UART_H

#include <MKL25Z4.h>
#include "dma.h"
#include "read.h"

//declarations in UART RX

#define BAUD_RATE        (9600)
#define OVERSAMPLE_RATE  (25)
#define CLOCK                (24000000)

#define TX_PIN           (2) //on port A - ALT 2 is UART0_Tx

#define LSB(x)              (uint8_t)(x & 0x00FF)
#define MSB(x)              (uint8_t)((x & 0xFF00) >> 8)
#define GLOVEMASK(x)        ((x) >> 16)
#define COMBINE(MSB, LSB)  (uint16_t)(((uint16_t)MSB << 8) | LSB)
```

```
void Init_UART(void);
void UART0_Transmit_DMA(int);
void TransmitBufferCreate(void);
void UART1_Transmit(int value);
void UART0_Transmit_OneByte(void);
void Init_UART1(void);


extern uint8_t transmitbuffer[11];
extern int shiftedpinky;
extern int shiftedring;
extern int shiftedmiddle;
extern int shiftedindex;
extern int shiftedthumb;
//extern uint8_t cal_val;

#endif

//Note: added ifndef and endif to get rid of "include nested too deeply"
//error

```

Code for Mode detect source file

```
#include"read.h"

uint8_t cal_val;
//cal_val = 0 indicates performance mode
//cal_val = 1 indicates calibration of standing up
//cal_val = 2 indicates calibration of bent position

void Init_CAL_GPIO(void){

  // Test code to see if ADC works
  // Enable clocks
  SIM->SCGC5 |= SIM_SCGC5_PORTA_MASK | SIM_SCGC5_PORTB_MASK ;


  //Select GPIO pin for 6 LED outputs
  PORTB->PCR[CAL_LED_1] &= ~PORT_PCR_MUX_MASK;
  PORTB->PCR[CAL_LED_1] |= PORT_PCR_MUX(1);
  PORTB->PCR[CAL_LED_2] &= ~PORT_PCR_MUX_MASK;
  PORTB->PCR[CAL_LED_2] |= PORT_PCR_MUX(1);
  PORTA->PCR[CAL_SW] &= ~PORT_PCR_MUX_MASK;
  PORTA->PCR[CAL_SW] |= PORT_PCR_MUX(1);

  //set LED bits as outputs
  PTB->PDDR |= MASK(CAL_LED_1) | MASK(CAL_LED_2);

  //Set sw as input
  PTA->PDDR &= ~MASK(CAL_SW);

  //Switch off all LEDS intially
```

```
 PTB->PSOR = MASK(CAL_LED_1);
 PTB->PSOR = MASK(CAL_LED_2);

}

void Read_Switches(void){
 if(PTA->PDIR & MASK(CAL_SW))
 {
     //restart timer
     PIT->MCR |= PIT_MCR_MDIS_MASK;
     PIT->MCR &= ~PIT_MCR_MDIS_MASK;
     cal_val = 1;
 }
}

void Light_LED(void){
 if(cal_val > 0)
 {
     if(cal_val == 1)
     {
         PTB->PSOR = MASK(CAL_LED_2);
         PTB->PCOR = MASK(CAL_LED_1);

     }
     if(cal_val == 2)
     {
         PTB->PSOR = MASK(CAL_LED_1);
         PTB->PCOR = MASK(CAL_LED_2);
     }
 }
 else
 {
     PTB->PSOR = MASK(CAL_LED_1);
     PTB->PSOR = MASK(CAL_LED_2);
 }
}
```

Code for Mode detect header file

```
#include <MKL25Z4.h>

#define CAL_LED_1 (19) //on port B
#define CAL_LED_2 (18) //on port B
#define CAL_SW      (1) //on port A
#define MASK(x)     (1UL << (x))

extern uint8_t cal_val;

void Init_CAL_GPIO(void);
void Read_Switches(void);
void Light_LED(void);
void delay(unsigned int length_ms);
```

Code for main source file on Transmission MCU

```
#include "main.h"


int index = 0;

int main(void){

    adc_init();
    Init_UART();
    dma_init();
    pit_init();
    Init_CAL_GPIO();
    Init_UART1();

    while(1)
    {
        //UART0_Transmit_OneByte();
        Read_Switches();
        Light_LED();
        if(while_flag == 1)
        {
            TransmitBufferCreate();

            while(while_flag)
            {

                    UART0->C2 |= UART0_C2_TE_MASK;
                  UART1->C2 |= UART_C2_TE_MASK;
                    if (index < 11)
                    {
                        UART1_Transmit(1);
                        UART0_Transmit_DMA(index);
                    }
                    index++;
                    if(index > 11) //if you change 11 to a 10, the uart might be on the cusp of sending/not sending
                                             //11 gives the uart plenty of time to reset the index, etc.
                    {
                        UART1_Transmit(0);
                        UART0->C2 &= ~UART0_C2_TE_MASK; //disable interrupt once all values are transferred
                        UART1->C2 &= ~UART_C2_TE_MASK;
                        index = 0;
                        while_flag = 0; //stops while loop
                        conversion_done = 0;
                    }
             }
        }
    }
```

```
}
```

Code for main header file on Transmission MCU

```
#include "pit.h"
#include "dma.h"
#include "adc.h"
#include "read.h"
#include <MKL25Z4.h>
```

Code for UART RX source file

```
#include "UART.h"

//UART RX
uint8_t receivedbuffer[11];
uint16_t receivedADCread[5];
uint8_t receivedcal_val;
volatile int received_go_signal = 0;
volatile int raise_this_flag = 0;




void Init_UART(void)
{
 // Let's use UART0 and PTA
 // Enable clock gating for UART0 and Port A

 uint16_t sbr;

 //We use Receive data from glove and maybe eventually Transmit (for error check)
 //Since the UART can only take 8 bits, and we have 5 important 16 bit values that
 //need to be sent, we would have to fill D reg with each half of an array element
 //making it a total of 5*2 times D needs to be updated.
 SIM->SCGC4 |= SIM_SCGC4_UART0_MASK;
 SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK;

 // Make sure transmitter and receiver are disabled before configuring
 UART0->C2 &= ~UART_C2_TE_MASK & ~UART_C2_RE_MASK;

 // UART 1 has to use Bus Clock, clock can't be configured
 // according to table 5-2 (page 122) of reference manual
 // and absence of macors
  // but UART0 clock needs to be set - to 48 MHz clock in this case
 SIM->SOPT2 |= SIM_SOPT2_UART0SRC(1);
 //SIM->SOPT2 |= SIM_SOPT2_PLLFLLSEL_MASK;

```

```c
// Set pins to UART1 Rx and Tx
PORTE->PCR[RX_PIN] |= PORT_PCR_MUX(4); // Rx

// Set baud rate and oversampling ratio
sbr = (uint16_t)((CLOCK)/(BAUD_RATE * OVERSAMPLE_RATE)) - 32;
UART0->BDH &= ~UART0_BDH_SBR_MASK;
UART0->BDH |= UART0_BDH_SBR(0x1F & (sbr>>8));
UART0->BDL = UART0_BDL_SBR(sbr);
UART0->C4 |= UART0_C4_OSR(OVERSAMPLE_RATE-1); //UARTLP_C4_OSR(x) is same as UART1_C4_OSR(x) and since C4 reg all UART is same, it should work

//LIN break is high whenever a break is detected.
//I believe this occurs when there is empty space in a transmission or recieve line
//this will set LBKDIF bit in UARTx_S2 as a high.
//setting LBKDIE bit in BDH reg will cause an interrupt if LBKDIF is high
UART0->BDH |= UART0_BDH_LBKDIE(0);  // for now let's disable it
UART0->BDH |= UART0_BDH_RXEDGIE(0); // Disable interrupts for RX active edge
UART0->BDH |= UART0_BDH_SBNS(0);   // one stop-bit - only two options 1 or 2
                                                                                   // Its logic level is the same as the signal's idle state, i.e., logic high
UART0->C1 |= UART0_C1_M(0); //Data bit mode - 8 bits - In 8-bit data mode, the shift register holds a
                                                                     //start bit, eight data bits, and a stop bit.
 UART0->C1 |= UART0_C1_LOOPS(0); //0: Normal operation - RxD and TxD use separate pins.
UART0->C1   |= UART0_C1_PE(0);  //Don't use parity bit
                                                                          //We can use parity bit but will need to change bit mode to 9 - The 9-bit data mode is typically used with parity to allow eight bits of data plus the parity in the ninth bit
                                                                          //When parity is enabled, the bit immediately before the stop bit is treated as the parity bit
                                                                          // a little confused about D being only 8 bits long
                                                                          // will also need to set PT bit in C1 to parity odd or even and enable interrupt for parity (PEIE) in C3

// Don't enable interrupts for errors
UART0->C3 = UART0_C3_ORIE(0)| UART0_C3_NEIE(0) | UART0_C3_FEIE(0) | UART0_C3_PEIE(0);

// Clear error flags
UART0->S1 |= UART0_S1_OR(1) | UART0_S1_NF(1) | UART0_S1_FE(1) | UART0_S1_PF(1); //To clear these flags write logic one to them

 UART0->S2 = UART0_S2_MSBF(0) | UART0_S2_RXINV(0); //Send LSB first, do not invert received data
```

```
 UART0->C2 |= UART_C2_RE(1); // Enable UART receiver


}


void Init_UART1(void)
{
 // rx pin (PTC3)

 uint16_t sbr;

 SIM->SCGC4 |= SIM_SCGC4_UART1_MASK;
 SIM->SCGC5 |= SIM_SCGC5_PORTC_MASK;

 UART1->C2 &= ~UART_C2_TE_MASK & ~UART_C2_RE_MASK;

 PORTC->PCR[3] |= PORT_PCR_MUX(3); //3 is the RX pin for UART1; alternative 3 is the selection option for UART1 rx

 sbr = (uint16_t)((CLOCK)/(BAUD_RATE * 16)) - 32;
 UART1->BDH &= ~UART_BDH_SBR_MASK;
 UART1->BDH |= UART_BDH_SBR(0x1F & (sbr>>8));
 UART1->BDL = UART_BDL_SBR(sbr);
 //UART1->C4 |= UARTLP_C4_OSR(OVERSAMPLE_RATE-1);

 UART1->BDH |= UART_BDH_LBKDIE(0);
 UART1->BDH |= UART_BDH_RXEDGIE(0);
 UART1->BDH |= UART_BDH_SBNS(0);


     UART1->C1 |= UART_C1_M(0);
   UART1->C1 |= UART_C1_LOOPS(0);
     UART1->C1   |= UART_C1_PE(0);

     UART1->C3 = UART_C3_ORIE(0)| UART_C3_NEIE(0) | UART_C3_FEIE(0) | UART_C3_PEIE(0);

     UART1->S2 &= ~UARTLP_S2_MSBF_MASK;
     UART1 -> S2 = UART_S2_RXINV(0);
     UART1->C2 |= UART_C2_RE(1) | UART_C2_RIE(1);

     NVIC_SetPriority(UART1_IRQn, 0);
     NVIC_ClearPendingIRQ(UART1_IRQn);
     NVIC_EnableIRQ(UART1_IRQn);
     __enable_irq();
}



void UART1_IRQHandler(void)
{
```

```

    if((UART1 -> S1) & (UART_S1_RDRF_MASK))
    {
        received_go_signal = UART1 -> D;
        if(received_go_signal)
            raise_this_flag = 1;
        else
            raise_this_flag = 0;
    }

}






void UART0_Receive_DMA(int index) {


        while (!(UART0->S1 & UART0_S1_RDRF_MASK));

        receivedbuffer[index] = UART0->D;

}

void CombineReceiveBuffer(){

    receivedADCread[pinky_index] =  COMBINE(receivedbuffer[1], receivedbuffer[0]);
    receivedADCread[ring_index] =   COMBINE(receivedbuffer[3], receivedbuffer[2]);
    receivedADCread[middle_index] = COMBINE(receivedbuffer[5], receivedbuffer[4]);
    receivedADCread[index_index] =  COMBINE(receivedbuffer[7], receivedbuffer[6]);
    receivedADCread[thumb_index] =  COMBINE(receivedbuffer[9], receivedbuffer[8]);
    receivedcal_val = receivedbuffer[10];
}


void UART0_Receive_char (void)
{
    while (!(UART0->S1 & UART0_S1_RDRF_MASK));
        //Receive_Done = UART0->D;
}
```

Code for UART RX header file

```
#include <MKL25Z4.h>
#include "dma.h"


```

```
#define BAUD_RATE (9600)
#define OVERSAMPLE_RATE (25)
#define CLOCK (24000000)


#define RX_PIN 21 //on port E - ALT 4 is UART0_Rx

#define LSB(x) (uint8_t)(x & 0x00FF)
#define MSB(x) (uint8_t)((x & 0xFF00) >> 8)
#define COMBINE(MSB, LSB) (uint16_t)(((uint16_t)MSB << 8) | LSB)

void Init_UART(void);
void UART0_Receive_DMA(int);
void CombineReceiveBuffer(void);
void UART0_Receive_char (void);
void Init_UART1(void);

extern uint8_t receivedbuffer[11];
extern uint16_t receivedADCread[5];
extern uint8_t receivedcal_val;
extern volatile int received_go_signal;
extern volatile int raise_this_flag;
```

<u>Code for calibration and letter decode source file</u>

```
#include "cal.h"
#include "UART.h"
//uint16_t receivedADCread[5];
//uint8_t  receivedcal_val;
uint16_t Observed_Standing_values[5] = {0xFFFF, 0xF800, 0xFFFF, 0xFFFF, 0xCB00};
uint16_t Observed_Bent_values[5] = {0x5100, 0x0120, 0xAE00, 0xAF00, 0x6800};
uint32_t Standing_Ranges[10] = {0,0,0,0,0,0,0,0,0,0};
uint32_t Bent_Ranges[10] = {0,0,0,0,0,0,0,0,0,0};
uint32_t Middle_Ranges[10] = {0,0,0,0,0,0,0,0,0,0};
uint16_t LUT[26][5];
uint16_t counterstand = 0;
uint16_t counterbend = 0;
int pinky_flag, ring_flag, middle_flag, index_flag, thumb_flag;
//flag = 0: standing
//flag = 1: middle
//flag = 2: bent
//cal_val = 0 indicates performance mode
//cal_val = 1 indicates calibration of standing up
//cal_val = 2 indicates calibration of bent position

void ranges(void)
{
 if (receivedcal_val > 0)
 {
     if (receivedcal_val == 1)
     {
         Standing_Ranges[pinky_LB]  += receivedADCread[pinky_index];
```

```
        Standing_Ranges[pinky_UB]  = Observed_Standing_values[pinky_index];
        Standing_Ranges[ring_LB]   += receivedADCread[ring_index];
        Standing_Ranges[ring_UB]   = Observed_Standing_values[ring_index];
        Standing_Ranges[middle_LB] += receivedADCread[middle_index];
        Standing_Ranges[middle_UB] = Observed_Standing_values[middle_index];
        Standing_Ranges[index_LB]  += receivedADCread[index_index]; //lol
        Standing_Ranges[index_UB]  = Observed_Standing_values[index_index];
        Standing_Ranges[thumb_LB]  += receivedADCread[thumb_index];
        Standing_Ranges[thumb_UB]  = Observed_Standing_values[thumb_index];
        counterstand++;
    }
    if (receivedcal_val == 2)
    {
        Bent_Ranges[pinky_LB]  += receivedADCread[pinky_index];
        Bent_Ranges[pinky_UB]  = Observed_Bent_values[pinky_index];
        Bent_Ranges[ring_LB]   += receivedADCread[ring_index];
        Bent_Ranges[ring_UB]   = Observed_Bent_values[ring_index];
        Bent_Ranges[middle_LB] += receivedADCread[middle_index];
        Bent_Ranges[middle_UB] = Observed_Bent_values[middle_index];
        Bent_Ranges[index_LB]  += receivedADCread[index_index]; //lol
        Bent_Ranges[index_UB]  = Observed_Bent_values[index_index];
        Bent_Ranges[thumb_LB]  += receivedADCread[thumb_index];
        Bent_Ranges[thumb_UB]  = Observed_Bent_values[thumb_index];
        counterbend++;
    }
    /*
    for (int i = 0; i < 10; i++)
    {
        Standing_Ranges[i] = Standing_Ranges[i]/counterstand;
        Bent_Ranges[i] = Bent_Ranges[i]/counterbend;
    }
    counterstand = 0;
    counterbend = 0;
    for (int i = 0; i < 10; i=i+2)
    {
        uint16_t min = Min(Standing_Ranges[i], Standing_Ranges[i+1]);
        uint16_t max = Max(Standing_Ranges[i], Standing_Ranges[i+1]);
        Standing_Ranges[i] = min;
        Standing_Ranges[i+1] = max;
        min = Min(Bent_Ranges[i], Bent_Ranges[i+1]);
        max = Max(Bent_Ranges[i], Bent_Ranges[i+1]);
        Bent_Ranges[i] = min;
        Bent_Ranges[i+1] = max;
    }

    Middle_Ranges[pinky_LB]  = (Standing_Ranges[pinky_LB] + Bent_Ranges[pinky_LB])/2;
```

```
        Middle_Ranges[pinky_UB]   = (Standing_Ranges[pinky_UB] + Bent_Ranges[pinky_UB])/2;
        Middle_Ranges[ring_LB]    = (Standing_Ranges[ring_LB] + Bent_Ranges[ring_LB])/2;
        Middle_Ranges[ring_UB]    = (Standing_Ranges[ring_UB] + Bent_Ranges[ring_UB])/2;
        Middle_Ranges[middle_LB] = (Standing_Ranges[middle_LB] + Bent_Ranges[middle_LB])/2;
        Middle_Ranges[middle_UB] = (Standing_Ranges[middle_UB] + Bent_Ranges[middle_UB])/2;
        Middle_Ranges[index_LB]   = (Standing_Ranges[index_LB] + Bent_Ranges[index_LB])/2;
        Middle_Ranges[index_UB]   = (Standing_Ranges[index_UB] + Bent_Ranges[index_UB])/2;
        Middle_Ranges[thumb_LB]   = (Standing_Ranges[thumb_LB] + Bent_Ranges[thumb_LB])/2;
        Middle_Ranges[thumb_UB]   = (Standing_Ranges[thumb_UB] + Bent_Ranges[thumb_UB])/2;*/
  }
}

void AVG()
{
        for (int i = 0; i < 10; i= i+2)
        {
            if(counterstand != 0)
                Standing_Ranges[i] = Standing_Ranges[i]/counterstand;
            if(counterbend!=0)
                Bent_Ranges[i] = Bent_Ranges[i]/counterbend;
        }
        counterbend = 0;
        counterstand = 0;
        for (int i = 0; i < 10; i=i+2)
        {
            uint16_t min = Min(Standing_Ranges[i], Standing_Ranges[i+1]);
                uint16_t max = Max(Standing_Ranges[i], Standing_Ranges[i+1]);
                Standing_Ranges[i] = min;
                Standing_Ranges[i+1] = max;
                min = Min(Bent_Ranges[i], Bent_Ranges[i+1]);
                max = Max(Bent_Ranges[i], Bent_Ranges[i+1]);
                Bent_Ranges[i] = min;
                Bent_Ranges[i+1] = max;
            }

            Middle_Ranges[pinky_LB]   = (Standing_Ranges[pinky_LB] + Bent_Ranges[pinky_LB])/2;
            Middle_Ranges[pinky_UB]   = (Standing_Ranges[pinky_UB] + Bent_Ranges[pinky_UB])/2;
            Middle_Ranges[ring_LB]    = (Standing_Ranges[ring_LB] + Bent_Ranges[ring_LB])/2;
            Middle_Ranges[ring_UB]    = (Standing_Ranges[ring_UB] + Bent_Ranges[ring_UB])/2;
```

```
        Middle_Ranges[middle_LB] = (Standing_Ranges[middle_LB] + Bent_Ranges[middle_LB])/2;
        Middle_Ranges[middle_UB] = (Standing_Ranges[middle_UB] + Bent_Ranges[middle_UB])/2;
        Middle_Ranges[index_LB]  = (Standing_Ranges[index_LB] + Bent_Ranges[index_LB])/2;
        Middle_Ranges[index_UB]  = (Standing_Ranges[index_UB] + Bent_Ranges[index_UB])/2;
        Middle_Ranges[thumb_LB]  = (Standing_Ranges[thumb_LB] + Bent_Ranges[thumb_LB])/2;
        Middle_Ranges[thumb_UB]  = (Standing_Ranges[thumb_UB] + Bent_Ranges[thumb_UB])/2;
    }




void LUT_func(void){

    /*
    LUT[A][0][pinky_index] = Bent_Ranges[0];
    LUT[A][1][pinky_index] = Bent_Ranges[1];
    LUT[A][0][ring_index] = Bent_Ranges[1];
    LUT[A][1][ring_index] = Bent_Ranges[2];
    LUT[A][0][middle_index] = Bent_Ranges[3];
    LUT[A][1][middle_index] = Bent_Ranges[4];
    LUT[A][0][index_index] = Bent_Ranges[5];
    LUT[A][1][index_index] = Bent_Ranges[6];
    LUT[A][0][thumb_index] = Standing_Ranges[7];
    LUT[A][1][thumb_index] = Standing_Ranges[8];
    */
    //Determing positions of each finger - what flag to set for each finger
    //pinky


    //LUT

    //Letter A - SAME AS T - use fsr.
    LUT[A][pinky_index]  = bent;
    LUT[A][ring_index]   = bent;
    LUT[A][middle_index] = bent;
    LUT[A][index_index]  = bent;
    LUT[A][thumb_index]  = standing;

    //Letter B
    LUT[B][pinky_index]  = standing;
    LUT[B][ring_index]   = standing;
    LUT[B][middle_index] = standing;
    LUT[B][index_index]  = standing;
    LUT[B][thumb_index]  = bent;

    //Letter C
```

```
    LUT[C][pinky_index]  = middle;
    LUT[C][ring_index]   = middle;
    LUT[C][middle_index] = middle;
    LUT[C][index_index]  = middle;
    LUT[C][thumb_index]  = middle;

    //Letter D
    LUT[D][pinky_index]  = middle;
    LUT[D][ring_index]   = middle;
    LUT[D][middle_index] = middle;
    LUT[D][index_index]  = standing;
    LUT[D][thumb_index]  = middle;

    //Letter E
    LUT[E][pinky_index]  = bent;
    LUT[E][ring_index]   = bent;
    LUT[E][middle_index] = bent;
    LUT[E][index_index]  = bent;
    LUT[E][thumb_index]  = bent;

    //Letter F
    LUT[F][pinky_index]  = standing;
    LUT[F][ring_index]   = standing;
    LUT[F][middle_index] = standing;
    LUT[F][index_index]  = middle;
    LUT[F][thumb_index]  = middle;

    //Letter G
    //Note: thumb can be interpreted as middle or standing,
    //in our implementation it's middle
    LUT[G][pinky_index]  = bent;
    LUT[G][ring_index]   = bent;
    LUT[G][middle_index] = bent;
    LUT[G][index_index]  = standing;
    LUT[G][thumb_index]  = middle;

    //Letter H
    LUT[H][pinky_index]  = bent;
    LUT[H][ring_index]   = bent;
    LUT[H][middle_index] = standing;
    LUT[H][index_index]  = standing;
    LUT[H][thumb_index]  = middle;

    //Letter I
    LUT[I][pinky_index]  = standing;
    LUT[I][ring_index]   = bent;
    LUT[I][middle_index] = bent;
    LUT[I][index_index]  = bent;
    LUT[I][thumb_index]  = bent;

    //Letter J
    LUT[J][pinky_index]  = standing;
    LUT[J][ring_index]   = bent;
```

```
    LUT[J][middle_index] = bent;
    LUT[J][index_index]  = bent;
    LUT[J][thumb_index]  = standing;

    //Letter K //SAME AS U
    LUT[K][pinky_index]  = bent;
    LUT[K][ring_index]   = bent;
    LUT[K][middle_index] = standing;
    LUT[K][index_index]  = standing;
    LUT[K][thumb_index]  = standing;

    //Letter L
    LUT[L][pinky_index]  = bent;
    LUT[L][ring_index]   = bent;
    LUT[L][middle_index] = bent;
    LUT[L][index_index]  = standing;
    LUT[L][thumb_index]  = standing;

    //Letter M //SAME AS E!!!!!!!!
    LUT[M][pinky_index]  = bent;
    LUT[M][ring_index]   = bent;
    LUT[M][middle_index] = bent;
    LUT[M][index_index]  = bent;
    LUT[M][thumb_index]  = bent;

    //Letter N //SAME AS S!!!!!!!!
    LUT[N][pinky_index]  = bent;
    LUT[N][ring_index]   = bent;
    LUT[N][middle_index] = bent;
    LUT[N][index_index]  = bent;
    LUT[N][thumb_index]  = middle;

    //Letter O
    LUT[O][pinky_index]  = middle;
    LUT[O][ring_index]   = middle;
    LUT[O][middle_index] = middle;
    LUT[O][index_index]  = middle;
    LUT[O][thumb_index]  = middle;

    //Letter P
    LUT[P][pinky_index]  = bent;
    LUT[P][ring_index]   = bent;
    LUT[P][middle_index] = middle;
    LUT[P][index_index]  = standing;
    LUT[P][thumb_index]  = standing;

    //Letter Q
    LUT[Q][pinky_index]  = bent;
    LUT[Q][ring_index]   = bent;
    LUT[Q][middle_index] = bent;
    LUT[Q][index_index]  = standing;
    LUT[Q][thumb_index]  = standing;
```

```
    //Letter R
    LUT[R][pinky_index]  = bent;
    LUT[R][ring_index]   = bent;
    LUT[R][middle_index] = standing;
    LUT[R][index_index]  = standing;
    LUT[R][thumb_index]  = bent;

    //Letter S
    LUT[S][pinky_index]  = bent;
    LUT[S][ring_index]   = bent;
    LUT[S][middle_index] = bent;
    LUT[S][index_index]  = bent;
    LUT[S][thumb_index]  = middle;

    //Letter T
    LUT[T][pinky_index]  = bent;
    LUT[T][ring_index]   = bent;
    LUT[T][middle_index] = bent;
    LUT[T][index_index]  = bent;
    LUT[T][thumb_index]  = standing;

    //Letter U //SAME AS K!!!
    LUT[U][pinky_index]  = bent;
    LUT[U][ring_index]   = bent;
    LUT[U][middle_index] = standing;
    LUT[U][index_index]  = standing;
    LUT[U][thumb_index]  = standing;

    //Letter V //SAME AS K!!!!
    LUT[V][pinky_index]  = bent;
    LUT[V][ring_index]   = bent;
    LUT[V][middle_index] = bent;
    LUT[V][index_index]  = bent;
    LUT[V][thumb_index]  = standing;

    //Letter W
    LUT[W][pinky_index]  = bent;
    LUT[W][ring_index]   = standing;
    LUT[W][middle_index] = standing;
    LUT[W][index_index]  = standing;
    LUT[W][thumb_index]  = bent;

    //Letter X
    LUT[X][pinky_index]  = bent;
    LUT[X][ring_index]   = bent;
    LUT[X][middle_index] = bent;
    LUT[X][index_index]  = bent;
    LUT[X][thumb_index]  = standing;

    //Letter Y
    LUT[Y][pinky_index]  = bent;
    LUT[Y][ring_index]   = bent;
    LUT[Y][middle_index] = bent;
```

```
    LUT[Y][index_index]  = middle;
    LUT[Y][thumb_index]  = bent;

    //Letter Z
    LUT[Z][pinky_index]  = bent;
    LUT[Z][ring_index]   = bent;
    LUT[Z][middle_index] = bent;
    LUT[Z][index_index]  = standing;
    LUT[Z][thumb_index]  = bent;

}

//this function returns 0-25, indicating A-Z which will be used as input to screen
int letter_decode(void)
{
    int bent_middle_mid_point;
    int stand_middle_mid_point;
    if(receivedADCread[pinky_index] >= Bent_Ranges[pinky_LB] && receivedADCread[pinky_index] <= Bent_Ranges[pinky_UB])
            pinky_flag = bent;
    else if(receivedADCread[pinky_index] >= Middle_Ranges[pinky_LB] && receivedADCread[pinky_index] <= Middle_Ranges[pinky_UB])
            pinky_flag = middle;
    else if(receivedADCread[pinky_index] >= Standing_Ranges[pinky_LB] && receivedADCread[pinky_index] <= Standing_Ranges[pinky_UB])
            pinky_flag = standing;
    else if (receivedADCread[pinky_index]  < Bent_Ranges[pinky_LB])
            pinky_flag = bent;
    else if (receivedADCread[pinky_index]  > Standing_Ranges[pinky_UB])
            pinky_flag = standing;
    else
    {
        bent_middle_mid_point =  (Bent_Ranges[pinky_UB] + Middle_Ranges[pinky_LB])/2;
        stand_middle_mid_point = (Middle_Ranges[pinky_UB] + Standing_Ranges[pinky_LB])/2;
        if (receivedADCread[pinky_index]  < Standing_Ranges[pinky_LB] && receivedADCread[pinky_index]  > stand_middle_mid_point)
            pinky_flag = standing;
        if (receivedADCread[pinky_index]  > Middle_Ranges[pinky_LB] && receivedADCread[pinky_index]  < stand_middle_mid_point)
            pinky_flag = middle;
        if (receivedADCread[pinky_index]  < Middle_Ranges[pinky_LB] && receivedADCread[pinky_index]  > bent_middle_mid_point)
            pinky_flag = middle;
        if (receivedADCread[pinky_index]  > Bent_Ranges[pinky_UB] && receivedADCread[pinky_index]  < bent_middle_mid_point)
            pinky_flag = bent;
    }

    //ring
```

```
    if(receivedADCread[ring_index] >= Bent_Ranges[ring_LB] && receivedADCread[ring_index] <= Bent_Ranges[ring_UB])
            ring_flag = bent;
    else if(receivedADCread[ring_index] >= Middle_Ranges[ring_LB] && receivedADCread[ring_index] <= Middle_Ranges[ring_UB])
            ring_flag = middle;
    else if(receivedADCread[ring_index] >= Standing_Ranges[ring_LB] && receivedADCread[ring_index] <= Standing_Ranges[ring_UB])
            ring_flag = standing;
    else if (receivedADCread[ring_index]  < Bent_Ranges[ring_LB])
            ring_flag = bent;
    else if (receivedADCread[ring_index]  > Standing_Ranges[ring_UB])
            ring_flag = standing;
    else
    {
        bent_middle_mid_point =  (Bent_Ranges[ring_UB] + Middle_Ranges[ring_LB])/2;
        stand_middle_mid_point = (Middle_Ranges[ring_UB] + Standing_Ranges[ring_LB])/2;
        if (receivedADCread[ring_index]  < Standing_Ranges[ring_LB] && receivedADCread[ring_index]  > stand_middle_mid_point)
            ring_flag = standing;
        if (receivedADCread[ring_index]  > Middle_Ranges[ring_LB] && receivedADCread[ring_index]  < stand_middle_mid_point)
            ring_flag = middle;
        if (receivedADCread[ring_index]  < Middle_Ranges[ring_LB] && receivedADCread[ring_index]  > bent_middle_mid_point)
            ring_flag = middle;
        if (receivedADCread[ring_index]  > Bent_Ranges[ring_UB] && receivedADCread[ring_index]  < bent_middle_mid_point)
            ring_flag = bent;
    }

    //middle
    if(receivedADCread[middle_index] >= Bent_Ranges[middle_LB] && receivedADCread[middle_index] <= Bent_Ranges[middle_UB])
            middle_flag = bent;
    else if(receivedADCread[middle_index] >= Middle_Ranges[middle_LB] && receivedADCread[middle_index] <= Middle_Ranges[middle_UB])
            middle_flag = middle;
    else if(receivedADCread[middle_index] >= Standing_Ranges[middle_LB] && receivedADCread[middle_index] <= Standing_Ranges[middle_UB])
            middle_flag = standing;
    else if (receivedADCread[middle_index]  < Bent_Ranges[middle_LB])
            middle_flag = bent;
    else if (receivedADCread[middle_index]  > Standing_Ranges[middle_UB])
            middle_flag = standing;
    else
    {
        bent_middle_mid_point =  (Bent_Ranges[middle_UB] + Middle_Ranges[middle_LB])/2;
```

```
        stand_middle_mid_point = (Middle_Ranges[middle_UB] + Standing_Ranges[middle_LB])/2;
        if (receivedADCread[middle_index]  < Standing_Ranges[middle_LB] && receivedADCread[middle_index]  > stand_middle_mid_point)
            middle_flag = standing;
        if (receivedADCread[middle_index]  > Middle_Ranges[middle_LB] && receivedADCread[middle_index]  < stand_middle_mid_point)
            middle_flag = middle;
        if (receivedADCread[middle_index]  < Middle_Ranges[middle_LB] && receivedADCread[middle_index]  > bent_middle_mid_point)
            middle_flag = middle;
        if (receivedADCread[middle_index]  > Bent_Ranges[middle_UB] && receivedADCread[middle_index]  < bent_middle_mid_point)
            middle_flag = bent;
    }

    //index
    if(receivedADCread[index_index] >= Bent_Ranges[index_LB] && receivedADCread[index_index] <= Bent_Ranges[index_UB])
            index_flag = bent;
    else if(receivedADCread[index_index] >= Middle_Ranges[index_LB] && receivedADCread[index_index] <= Middle_Ranges[index_UB])
            index_flag = middle;
    else if(receivedADCread[index_index] >= Standing_Ranges[index_LB] && receivedADCread[index_index] <= Standing_Ranges[index_UB])
            index_flag = standing;
    else if (receivedADCread[index_index]  < Bent_Ranges[index_LB])
            index_flag = bent;
    else if (receivedADCread[index_index]  > Standing_Ranges[index_UB])
            index_flag = standing;
    else
    {
        bent_middle_mid_point =  (Bent_Ranges[index_UB] + Middle_Ranges[index_LB])/2;
        stand_middle_mid_point = (Middle_Ranges[index_UB] + Standing_Ranges[index_LB])/2;
        if (receivedADCread[index_index]  < Standing_Ranges[index_LB] && receivedADCread[index_index]  > stand_middle_mid_point)
            index_flag = standing;
        if (receivedADCread[index_index]  > Middle_Ranges[index_LB] && receivedADCread[index_index]  < stand_middle_mid_point)
            index_flag = middle;
        if (receivedADCread[index_index]  < Middle_Ranges[index_LB] && receivedADCread[index_index]  > bent_middle_mid_point)
            index_flag = middle;
        if (receivedADCread[index_index]  > Bent_Ranges[index_UB] && receivedADCread[index_index]  < bent_middle_mid_point)
            index_flag = bent;
    }

    //thumb
    if(receivedADCread[thumb_index] >= Bent_Ranges[thumb_LB] && receivedADCread[thumb_index] <= Bent_Ranges[thumb_UB])
```

```
            thumb_flag = bent;
    else if(receivedADCread[thumb_index] >= Middle_Ranges[thumb_LB] && receivedADCread[thumb_index] <= Middle_Ranges[thumb_UB])
            thumb_flag = middle;
    else if(receivedADCread[thumb_index] >= Standing_Ranges[thumb_LB] && receivedADCread[thumb_index] <= Standing_Ranges[thumb_UB])
            thumb_flag = standing;
    else if (receivedADCread[thumb_index]  < Bent_Ranges[thumb_LB])
            thumb_flag = bent;
    else if (receivedADCread[thumb_index]  > Standing_Ranges[thumb_UB])
            thumb_flag = standing;
    else
    {
        bent_middle_mid_point =  (Bent_Ranges[thumb_UB] + Middle_Ranges[thumb_LB])/2;
        stand_middle_mid_point = (Middle_Ranges[thumb_UB] + Standing_Ranges[thumb_LB])/2;
        if (receivedADCread[thumb_index]  < Standing_Ranges[thumb_LB] && receivedADCread[thumb_index]  > stand_middle_mid_point)
            thumb_flag = standing;
        if (receivedADCread[thumb_index]  > Middle_Ranges[thumb_LB] && receivedADCread[thumb_index]  < stand_middle_mid_point)
            thumb_flag = middle;
        if (receivedADCread[thumb_index]  < Middle_Ranges[thumb_LB] && receivedADCread[thumb_index]  > bent_middle_mid_point)
            thumb_flag = middle;
        if (receivedADCread[thumb_index]  > Bent_Ranges[thumb_UB] && receivedADCread[thumb_index]  < bent_middle_mid_point)
            thumb_flag = bent;
    }


    //Letter A - 0
    if ((pinky_flag == LUT[A][pinky_index]) && (ring_flag == LUT[A][ring_index]) && (middle_flag == LUT[A][middle_index]) && (index_flag == LUT[A][index_index]) && (thumb_flag == LUT[A][thumb_index]))
    {
        return A;
    }

    //Letter B - 1
    if ((pinky_flag == LUT[B][pinky_index]) && (ring_flag == LUT[B][ring_index]) && (middle_flag == LUT[B][middle_index]) && (index_flag == LUT[B][index_index]) && (thumb_flag == LUT[B][thumb_index]))
    {
        return B;
    }

    //Letter C - 2
    if ((pinky_flag == LUT[C][pinky_index]) && (ring_flag == LUT[C][ring_index]) && (middle_flag == LUT[C][middle_index]) && (index_flag == LUT[C][index_index]) && (thumb_flag == LUT[C][thumb_index]))
    {
```

```
        return C;
    }

    //Letter D - 3
    if ((pinky_flag == LUT[D][pinky_index]) && (ring_flag == LUT[D][ring_index]) && (middle_flag == LUT[D][middle_index]) && (index_flag == LUT[D][index_index]) && (thumb_flag == LUT[D][thumb_index]))
    {
        return D;
    }

    //Letter E - 4
    if ((pinky_flag == LUT[E][pinky_index]) && (ring_flag == LUT[E][ring_index]) && (middle_flag == LUT[E][middle_index]) && (index_flag == LUT[E][index_index]) && (thumb_flag == LUT[E][thumb_index]))
    {
        return E;
    }

    //Letter F - 5
    if ((pinky_flag == LUT[F][pinky_index]) && (ring_flag == LUT[F][ring_index]) && (middle_flag == LUT[F][middle_index]) && (index_flag == LUT[F][index_index]) && (thumb_flag == LUT[F][thumb_index]))
    {
        return F;
    }

    //Letter G - 6
    if ((pinky_flag == LUT[G][pinky_index]) && (ring_flag == LUT[G][ring_index]) && (middle_flag == LUT[G][middle_index]) && (index_flag == LUT[G][index_index]) && (thumb_flag == LUT[G][thumb_index]))
    {
        return G;
    }

    //Letter H - 7
    if ((pinky_flag == LUT[H][pinky_index]) && (ring_flag == LUT[H][ring_index]) && (middle_flag == LUT[H][middle_index]) && (index_flag == LUT[H][index_index]) && (thumb_flag == LUT[H][thumb_index]))
    {
        return H;
    }

    //Letter I - 8
    if ((pinky_flag == LUT[I][pinky_index]) && (ring_flag == LUT[I][ring_index]) && (middle_flag == LUT[I][middle_index]) && (index_flag == LUT[I][index_index]) && (thumb_flag == LUT[I][thumb_index]))
    {
        return I;
    }

    //Letter J - 9
```

```
    if ((pinky_flag == LUT[J][pinky_index]) && (ring_flag == LUT[J][ring_index]) && (middle_flag == LUT[J][middle_index]) && (index_flag == LUT[J][index_index]) && (thumb_flag == LUT[J][thumb_index]))
    {
        return J;
    }

    //Letter K - 10
    if ((pinky_flag == LUT[K][pinky_index]) && (ring_flag == LUT[K][ring_index]) && (middle_flag == LUT[K][middle_index]) && (index_flag == LUT[K][index_index]) && (thumb_flag == LUT[K][thumb_index]))
    {
        return K;
    }

    //Letter L - 11
    if ((pinky_flag == LUT[L][pinky_index]) && (ring_flag == LUT[L][ring_index]) && (middle_flag == LUT[L][middle_index]) && (index_flag == LUT[L][index_index]) && (thumb_flag == LUT[L][thumb_index]))
    {
        return L;
    }

    //Letter M - 12
    if ((pinky_flag == LUT[M][pinky_index]) && (ring_flag == LUT[M][ring_index]) && (middle_flag == LUT[M][middle_index]) && (index_flag == LUT[M][index_index]) && (thumb_flag == LUT[M][thumb_index]))
    {
        return M;
    }

    //Letter N - 13
    if ((pinky_flag == LUT[N][pinky_index]) && (ring_flag == LUT[N][ring_index]) && (middle_flag == LUT[N][middle_index]) && (index_flag == LUT[N][index_index]) && (thumb_flag == LUT[N][thumb_index]))
    {
        return N;
    }

    //Letter O - 14
    if ((pinky_flag == LUT[O][pinky_index]) && (ring_flag == LUT[O][ring_index]) && (middle_flag == LUT[O][middle_index]) && (index_flag == LUT[O][index_index]) && (thumb_flag == LUT[O][thumb_index]))
    {
        return O;
    }

    //Letter P - 15
    if ((pinky_flag == LUT[P][pinky_index]) && (ring_flag == LUT[P][ring_index]) && (middle_flag == LUT[P][middle_index]) && (index_flag == LUT[P][index_index]) && (thumb_flag == LUT[P][thumb_index]))
    {
        return P;
```

```
    }

    //Letter Q - 16
    if ((pinky_flag == LUT[Q][pinky_index]) && (ring_flag == LUT[Q][ring_index]) && (middle_flag == LUT[Q][middle_index]) && (index_flag == LUT[Q][index_index]) && (thumb_flag == LUT[Q][thumb_index]))
    {
        return Q;
    }

    //Letter R - 17
    if ((pinky_flag == LUT[R][pinky_index]) && (ring_flag == LUT[R][ring_index]) && (middle_flag == LUT[R][middle_index]) && (index_flag == LUT[R][index_index]) && (thumb_flag == LUT[R][thumb_index]))
    {
        return R;
    }

    //Letter S - 18
    if ((pinky_flag == LUT[S][pinky_index]) && (ring_flag == LUT[S][ring_index]) && (middle_flag == LUT[S][middle_index]) && (index_flag == LUT[S][index_index]) && (thumb_flag == LUT[S][thumb_index]))
    {
        return S;
    }

    //Letter T - 19
    if ((pinky_flag == LUT[T][pinky_index]) && (ring_flag == LUT[T][ring_index]) && (middle_flag == LUT[T][middle_index]) && (index_flag == LUT[T][index_index]) && (thumb_flag == LUT[T][thumb_index]))
    {
        return T;
    }

    //Letter U - 20
    if ((pinky_flag == LUT[U][pinky_index]) && (ring_flag == LUT[U][ring_index]) && (middle_flag == LUT[U][middle_index]) && (index_flag == LUT[U][index_index]) && (thumb_flag == LUT[U][thumb_index]))
    {
        return U;
    }

    //Letter V - 21
    if ((pinky_flag == LUT[V][pinky_index]) && (ring_flag == LUT[V][ring_index]) && (middle_flag == LUT[V][middle_index]) && (index_flag == LUT[V][index_index]) && (thumb_flag == LUT[V][thumb_index]))
    {
        return V;
    }

    //Letter W - 22
```

```
    if ((pinky_flag == LUT[W][pinky_index]) && (ring_flag == LUT[W][ring_index]) && (middle_flag == LUT[W][middle_index]) && (index_flag == LUT[W][index_index]) && (thumb_flag == LUT[W][thumb_index]))
    {
        return W;
    }

    //Letter X - 23
    if ((pinky_flag == LUT[X][pinky_index]) && (ring_flag == LUT[X][ring_index]) && (middle_flag == LUT[X][middle_index]) && (index_flag == LUT[X][index_index]) && (thumb_flag == LUT[X][thumb_index]))
    {
        return X;
    }

    //Letter Y - 24
    if ((pinky_flag == LUT[Y][pinky_index]) && (ring_flag == LUT[Y][ring_index]) && (middle_flag == LUT[Y][middle_index]) && (index_flag == LUT[Y][index_index]) && (thumb_flag == LUT[Y][thumb_index]))
    {
        return Y;
    }

    //Letter Z - 25
    if ((pinky_flag == LUT[Z][pinky_index]) && (ring_flag == LUT[Z][ring_index]) && (middle_flag == LUT[Z][middle_index]) && (index_flag == LUT[Z][index_index]) && (thumb_flag == LUT[Z][thumb_index]))
    {
        return Z;
    }

return 26;
}

uint32_t Max(uint32_t num1, uint32_t num2)
{
    if (num1 >= num2)
        return num1;
    else
        return num2;
}

uint32_t Min(uint32_t num1, uint32_t num2)
{
    if (num1 <= num2)
        return num1;
    else
        return num2;
}
```

<u>Code for calibration and letter decode header file</u>

```
#include <MKL25Z4.h>
```

```c
#include "UART.h"
#include "adc.h"
#include "dma.h"

#define A (0)
#define B (1)
#define C (2)
#define D (3)
#define E (4)
#define F (5)
#define G (6)
#define H (7)
#define I (8)
#define J (9)
#define K (10)
#define L (11)
#define M (12)
#define N (13)
#define O (14)
#define P (15)
#define Q (16)
#define R (17)
#define S (18)
#define T (19)
#define U (20)
#define V (21)
#define W (22)
#define X (23)
#define Y (24)
#define Z (25)

#define standing (0)
#define middle   (1)
#define bent     (2)

#define pinky_LB  (0)
#define pinky_UB  (1)
#define ring_LB   (2)
#define ring_UB   (3)
#define middle_LB (4)
#define middle_UB (5)
#define index_LB  (6)
#define index_UB  (7)
#define thumb_LB  (8)
#define thumb_UB  (9)

extern uint16_t Observed_Standing_values[5];
extern uint16_t Observed_Bent_values[5];
extern uint32_t Standing_Ranges[10];
extern uint32_t Bent_Ranges[10];
extern uint32_t Middle_Ranges[10];
extern uint16_t LUT[26][5];
extern int pinky_flag, ring_flag, middle_flag, index_flag, thumb_flag;
```

```
extern uint16_t receivedADCread[5];
extern uint8_t receivedcal_val;
extern uint16_t counterstand;
extern uint16_t counterbend;
extern int in;

void ranges(void);
uint32_t Max(uint32_t num1, uint32_t num2);
uint32_t Min(uint32_t num1, uint32_t num2);
void LUT_func(void);
int letter_decode(void);
void AVG(void);
```

Code for 16 segment source file

```
#include "16Segment.h"
#include "cal.h"

int in;
void Init_14Segment(void){
//Enable Clock to port A and C
SIM_SCGC5 |= (SIM_SCGC5_PORTA_MASK | SIM_SCGC5_PORTC_MASK);
  PORTA->PCR[LED_A] &= ~PORT_PCR_MUX_MASK;
  PORTA->PCR[LED_A] |= PORT_PCR_MUX(1);
  PORTA->PCR[LED_B] &= ~PORT_PCR_MUX_MASK;
  PORTA->PCR[LED_B] |= PORT_PCR_MUX(1);
  PORTA->PCR[LED_C] &= ~PORT_PCR_MUX_MASK;
  PORTA->PCR[LED_C] |= PORT_PCR_MUX(1);
  PORTA->PCR[LED_D] &= ~PORT_PCR_MUX_MASK;
  PORTA->PCR[LED_D] |= PORT_PCR_MUX(1);
  PORTA->PCR[LED_E] &= ~PORT_PCR_MUX_MASK;
  PORTA->PCR[LED_E] |= PORT_PCR_MUX(1);
  PORTA->PCR[LED_F] &= ~PORT_PCR_MUX_MASK;
  PORTA->PCR[LED_F] |= PORT_PCR_MUX(1);
  PORTA->PCR[LED_G] &= ~PORT_PCR_MUX_MASK;
  PORTA->PCR[LED_G] |= PORT_PCR_MUX(1);
  PORTC->PCR[LED_H] &= ~PORT_PCR_MUX_MASK;
  PORTC->PCR[LED_H] |= PORT_PCR_MUX(1);
  PORTC->PCR[LED_K] &= ~PORT_PCR_MUX_MASK;
  PORTC->PCR[LED_K] |= PORT_PCR_MUX(1);
  PORTC->PCR[LED_M] &= ~PORT_PCR_MUX_MASK;
  PORTC->PCR[LED_M] |= PORT_PCR_MUX(1);
  PORTC->PCR[LED_N] &= ~PORT_PCR_MUX_MASK;
  PORTC->PCR[LED_N] |= PORT_PCR_MUX(1);
  PORTC->PCR[LED_P] &= ~PORT_PCR_MUX_MASK;
  PORTC->PCR[LED_P] |= PORT_PCR_MUX(1);
  PORTC->PCR[LED_S] &= ~PORT_PCR_MUX_MASK;
  PORTC->PCR[LED_S] |= PORT_PCR_MUX(1);
  PORTC->PCR[LED_R] &= ~PORT_PCR_MUX_MASK;
  PORTC->PCR[LED_R] |= PORT_PCR_MUX(1);
  PORTC->PCR[LED_T] &= ~PORT_PCR_MUX_MASK;
  PORTC->PCR[LED_T] |= PORT_PCR_MUX(1);
  PORTC->PCR[LED_U] &= ~PORT_PCR_MUX_MASK;
  PORTC->PCR[LED_U] |= PORT_PCR_MUX(1);
```

```
      //Set LED bits to outputs
  PTA->PDDR |= MASK(LED_A) | MASK(LED_B) | MASK(LED_C) | MASK(LED_D) | MASK(LED_E) | MASK(LED_F)| MASK(LED_G);
   PTC->PDDR |= MASK(LED_H) | MASK(LED_K) | MASK(LED_M) | MASK(LED_N) | MASK(LED_P) | MASK(LED_S)| MASK(LED_R) | MASK(LED_T)| MASK(LED_U);

  //Turn off all LED's off initially
  PTA->PSOR=MASK(LED_A);
  PTA->PSOR=MASK(LED_B);
  PTA->PSOR=MASK(LED_C);
  PTA->PSOR=MASK(LED_D);
  PTA->PSOR=MASK(LED_E);
  PTA->PSOR=MASK(LED_F);
  PTA->PSOR=MASK(LED_G);
  PTC->PSOR=MASK(LED_H);
  PTC->PSOR=MASK(LED_K);
  PTC->PSOR=MASK(LED_M);
  PTC->PSOR=MASK(LED_N);
  PTC->PSOR=MASK(LED_P);
  PTC->PSOR=MASK(LED_S);
  PTC->PSOR=MASK(LED_R);
  PTC->PSOR=MASK(LED_T);
  PTC->PSOR=MASK(LED_U);
}

void control_LEDs(int A_on, int B_on, int C_on, int D_on, int E_on, int F_on, int G_on,int H_on, int K_on, int M_on, int N_on, int P_on, int S_on, int R_on, int T_on, int U_on)
{
  if (A_on)
      PTA->PCOR = MASK(LED_A);
  else
      PTA->PSOR = MASK(LED_A);
  if (B_on)
      PTA->PCOR = MASK(LED_B);
  else
      PTA->PSOR = MASK(LED_B);
  if (C_on)
      PTA->PCOR = MASK(LED_C);
  else
      PTA->PSOR = MASK(LED_C);
  if (D_on)
      PTA->PCOR = MASK(LED_D);
  else
      PTA->PSOR = MASK(LED_D);
  if (E_on)
      PTA->PCOR = MASK(LED_E);
  else
      PTA->PSOR = MASK(LED_E);
  if (F_on)
      PTA->PCOR = MASK(LED_F);
  else
```

```
        PTA->PSOR = MASK(LED_F);
    if (G_on)
        PTA->PCOR = MASK(LED_G);
    else
        PTA->PSOR = MASK(LED_G);
     if (H_on)
        PTC->PCOR = MASK(LED_H);
    else
        PTC->PSOR = MASK(LED_H);
    if (K_on)
        PTC->PCOR = MASK(LED_K);
        else
            PTC->PSOR = MASK(LED_K);
        if (M_on)
            PTC->PCOR = MASK(LED_M);
        else
            PTC->PSOR = MASK(LED_M);
        if (N_on)
            PTC->PCOR = MASK(LED_N);
        else
            PTC->PSOR = MASK(LED_N);
        if (P_on)
            PTC->PCOR = MASK(LED_P);
        else
            PTC->PSOR = MASK(LED_P);
        if (S_on)
            PTC->PCOR = MASK(LED_S);
        else
            PTC->PSOR = MASK(LED_S);
        if (R_on)
            PTC->PCOR = MASK(LED_R);
        else
            PTC->PSOR = MASK(LED_R);
        if (T_on)
            PTC->PCOR = MASK(LED_T);
        else
            PTC->PSOR = MASK(LED_T);
        if (U_on)
            PTC->PCOR = MASK(LED_U);
        else
            PTC->PSOR = MASK(LED_U);

}
void Display_Alphabet(int input){
    in = input;
    if(input == 0)
    {
        control_LEDs(1,1,1,1,0,0,1,1,0,0,0,1,0,0,0,1);
    }
    else if(input == 1)
    {
        control_LEDs(1,1,1,1,1,1,0,0,0,1,0,1,0,1,0,0);
    }
```

```
      else if(input == 2)
      {
          control_LEDs(1,1,0,0,1,1,1,1,0,0,0,0,0,0,0,0);
      }
      else if(input == 3)
      {
          control_LEDs(1,1,1,1,1,1,0,0,0,1,0,0,0,1,0,0);
      }
      else if(input == 4)
      {
          control_LEDs(1,1,0,0,1,1,1,1,0,0,0,1,0,0,0,1);
      }
      else if(input == 5)
      {
          control_LEDs(1,1,0,0,0,0,1,1,0,0,0,1,0,0,0,1);
      }
      else if(input == 6)
      {
          control_LEDs(1,1,0,1,1,1,1,1,0,0,0,1,0,0,0,0);
      }
      else if(input == 7)
      {
          control_LEDs(0,0,1,1,0,0,1,1,0,0,0,1,0,0,0,1);
      }
      else if(input == 8)
      {
          control_LEDs(1,1,0,0,1,1,0,0,0,1,0,0,0,1,0,0);
    }
    else if(input == 9)
    {
      control_LEDs(1,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0);
    }
    else if(input == 10)
    {
      control_LEDs(0,0,0,0,0,0,1,1,0,0,1,0,1,0,0,1);
    }
    else if(input == 11)
    {
          control_LEDs(0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0);
      }
      else if(input == 12)
      {
          control_LEDs(0,0,1,1,0,0,1,1,1,0,1,0,0,0,0,0);
      }
      else if(input == 13)
      {
          control_LEDs(0,0,1,1,0,0,1,1,1,0,0,0,1,0,0,0);
      }
      else if(input == 14)
      {
          control_LEDs(1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0);
      }
      else if(input == 15)
```

```
    {
        control_LEDs(1,1,1,0,0,0,1,1,0,0,0,1,0,0,0,1);
    }
    else if(input == 16)
    {
        control_LEDs(1,1,1,1,1,1,1,1,0,0,0,0,1,0,0,0);
    }
    else if(input == 17)
    {
        control_LEDs(1,1,1,0,0,0,1,1,0,0,0,1,1,0,0,1);
    }
    else if(input == 18)
    {
        control_LEDs(1,1,0,1,1,1,0,1,0,0,0,1,0,0,0,1);
    }
    else if(input == 19)
    {
        control_LEDs(1,1,0,0,0,0,0,0,0,1,0,0,0,1,0,0);
    }
    else if(input == 20)
    {
        control_LEDs(0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0);
    }
    else if(input == 21)
    {
        control_LEDs(0,0,0,0,0,0,1,1,0,0,1,0,0,0,1,0);
    }
    else if(input == 22)
    {
        control_LEDs(0,0,1,1,0,0,1,1,0,0,0,0,1,0,1,0);
    }
    else if(input == 23)
    {
        control_LEDs(0,0,0,0,0,0,0,0,1,0,1,0,1,0,1,0);
    }
    else if(input == 24)
    {
        control_LEDs(0,0,1,1,1,1,0,1,0,0,0,1,0,0,0,1);
    }
    else if(input == 25)
    {
        control_LEDs(1,1,0,0,1,1,0,0,0,0,1,0,0,0,1,0);
    }
    else
    {
        control_LEDs(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1);
    }
}
```

Code for 16 segment header file

```
#include <MKL25Z4.h>

#define MASK(x) (1UL << (x))
```

```
#define LED_A (1) //on port A
#define LED_B (2) //on port A
#define LED_C (4) //on port A
#define LED_D (5) //on port A
#define LED_E (12) //on port A
#define LED_F (13) //on port A
#define LED_G (16) //on port A

#define LED_H (7)//on port C
#define LED_K (0)//on port C
#define LED_M (11)//on port C
#define LED_N (4) //on port C
#define LED_P (5)//on port C
#define LED_R (6) //on port C
#define LED_S (10)//on port C
#define LED_T (8) //on port C
#define LED_U (9)//on port C

void Init_14Segment(void);
void control_LEDs(int A_on, int B_on, int C_on, int D_on, int E_on, int F_on, int G_on,int H_on, int K_on, int M_on, int N_on, int P_on, int R_on, int S_on, int T_on, int U_on);
void Display_Alphabet(int input);

```

Code for main source file on Reception MCU

```
#include "main.h"

int ADC_is_reading_cal1 = 1000; //this flag is high till we are in cal = 1 mode
int ADC_is_reading_cal2 = 1000;
static int index = 0;
//uint16_t receivedADCread[5]= {0x0000,0x0002,0x0004,0x0006,0x0008};

int main (void){
  LUT_func();
  Init_14Segment();
  Init_UART();
  Init_UART1();

  while(1)
  {
      if(received_go_signal)
      {
          UART0_Receive_DMA(index);
          index++;

          if(index > 10)
          {
              index = 0;
              CombineReceiveBuffer();
              ranges();
          }
```

```
        }
        if (receivedcal_val == 0)
        {
            AVG();
            int input = letter_decode();
            //in = 23;
            Display_Alphabet(input);
        }

    }
}
```

Code for main header file on Reception MCU

```
#include <MKL25Z4.h>
#include "dma.h"
#include "cal.h"
#include "UART.h"
#include "adc.h"
#include "16Segment.h"

extern int screen_input;
```

Appendix B

Segments of MCU reference Manual

Table 38-41. I2C divider and hold values

ICR (hex)	SCL divider	SDA hold value	SCL hold (start) value	SCL hold (stop) value	ICR (hex)	SCL divider (clocks)	SDA hold (clocks)	SCL hold (start) value	SCL hold (stop) value
00	20	7	6	11	20	160	17	78	81
01	22	7	7	12	21	192	17	94	97
02	24	8	8	13	22	224	33	110	113
03	26	8	9	14	23	256	33	126	129
04	28	9	10	15	24	288	49	142	145
05	30	9	11	16	25	320	49	158	161
06	34	10	13	18	26	384	65	190	193
07	40	10	16	21	27	480	65	238	241
08	28	7	10	15	28	320	33	158	161
09	32	7	12	17	29	384	33	190	193
0A	36	9	14	19	2A	448	65	222	225
0B	40	9	16	21	2B	512	65	254	257
0C	44	11	18	23	2C	576	97	286	289
0D	48	11	20	25	2D	640	97	318	321
0E	56	13	24	29	2E	768	129	382	385
0F	68	13	30	35	2F	960	129	478	481
10	48	9	18	25	30	640	65	318	321
11	56	9	22	29	31	768	65	382	385
12	64	13	26	33	32	896	129	446	449
13	72	13	30	37	33	1024	129	510	513

Table continues on the next page...

Table 38-41. I2C divider and hold values (continued)

ICR (hex)	SCL divider	SDA hold value	SCL hold (start) value	SCL hold (stop) value	ICR (hex)	SCL divider (clocks)	SDA hold (clocks)	SCL hold (start) value	SCL hold (stop) value
14	80	17	34	41	34	1152	193	574	577
15	88	17	38	45	35	1280	193	638	641
16	104	21	46	53	36	1536	257	766	769
17	128	21	58	65	37	1920	257	958	961
18	80	9	38	41	38	1280	129	638	641
19	96	9	46	49	39	1536	129	766	769
1A	112	17	54	57	3A	1792	257	894	897
1B	128	17	62	65	3B	2048	257	1022	1025
1C	144	25	70	73	3C	2304	385	1150	1153
1D	160	25	78	81	3D	2560	385	1278	1281
1E	192	33	94	97	3E	3072	513	1534	1537
1F	240	33	118	121	3F	3840	513	1918	1921

UART memory map

Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
4006_A000	UART Baud Rate Register High (UART0_BDH)	8	R/W	00h	39.2.1/725
4006_A001	UART Baud Rate Register Low (UART0_BDL)	8	R/W	04h	39.2.2/726
4006_A002	UART Control Register 1 (UART0_C1)	8	R/W	00h	39.2.3/726
4006_A003	UART Control Register 2 (UART0_C2)	8	R/W	00h	39.2.4/728
4006_A004	UART Status Register 1 (UART0_S1)	8	R/W	C0h	39.2.5/729

Table continues on the next page...

UART memory map (continued)

Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
4006_A005	UART Status Register 2 (UART0_S2)	8	R/W	00h	39.2.6/731
4006_A006	UART Control Register 3 (UART0_C3)	8	R/W	00h	39.2.7/733
4006_A007	UART Data Register (UART0_D)	8	R/W	00h	39.2.8/734
4006_A008	UART Match Address Registers 1 (UART0_MA1)	8	R/W	00h	39.2.9/735
4006_A009	UART Match Address Registers 2 (UART0_MA2)	8	R/W	00h	39.2.10/ 736
4006_A00A	UART Control Register 4 (UART0_C4)	8	R/W	0Fh	39.2.11/ 736
4006_A00B	UART Control Register 5 (UART0_C5)	8	R/W	00h	39.2.12/ 737

HC-05 Bluetooth Module datasheet segments

AT command Default:

How to set the mode to server (master):

1. Connect PIO11 to high level.
2. Power on, module into command state.
3. Using baud rate 38400, sent the "AT+ROLE=1\r\n" to module, with "OK\r\n" means setting successes.
4. Connect the PIO11 to low level, repower the module, the module work as server (master).

AT commands: (all end with \r\n)

1. Test command:

Command	Respond	Parameter
AT	OK	-

8. Set/Check module mode:

Command	Respond	Parameter
AT+ROLE=<Param>	OK	Param:
AT+ ROLE?	+ROLE:<Param>	0- Slave

13. Set/Check serial parameter:

Command	Respond	Parameter
AT+UART=<Param>,<Param2>,<Param3>	OK	Param1: Baud Param2: Stop bit Param3: Parity
AT+ UART?	+UART=<Param>,<Param2>, <Param3> OK	

Example:

AT+UART=115200， 1,2,\r\n

OK

AT+UART?

+UART:115200,1,2

OK

16-segment display datasheet segments

20.32 mm (0.8 inch) 16 Segment Single Digit Alphanumeric Display

DESCRIPTIONS

- The Super Bright Red source color devices are made with Gallium Aluminum Arsenide Red Light Emitting Diode
- Electrostatic discharge and power surge could damage the LEDs
- It is recommended to use a wrist band or anti-electrostatic glove when handling the LEDs
- All devices, equipments and machineries must be electrically grounded

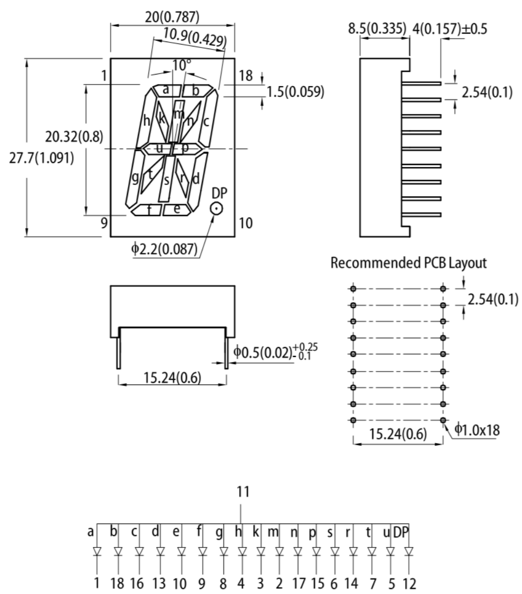
FEATURES

- 0.8 inch character height
- Low current operation
- High contrast and light output
- Common cathode and common anode available
- Easy mounting on P.C. boards or sockets
- Mechanically rugged
- Standard: gray face, white segment
- RoHS compliant

APPLICATIONS

- Home and smart appliances
- Display time and digital combination
- Industrial and instrumental applications
- Numeric status

PACKAGE DIMENSIONS



Appendix C

Information on data formats and slave devices for I2C.
A device can have one or multiple registers where data is stored, written, or read. Data transfer may be initiated only when the bus is idle. A bus is considered idle if both SDA and SCL lines are high after a STOP condition.

The general procedure for a master to access a slave device is the following:

1. A master wants to send data to a slave:
- Master-transmitter sends a START condition and addresses the slave-receiver
- Master-transmitter sends data to slave-receiver
- Master-transmitter terminates the transfer with a STOP condition

2. A master wants to receive/read data from a slave:
- Master-receiver sends a START condition and addresses the slave-transmitter
- Master-receiver sends the requested register to read to slave-transmitter
- Master-receiver receives data from the slave-transmitter
- Master-receiver terminates the transfer with a STOP condition.