Table of contents:

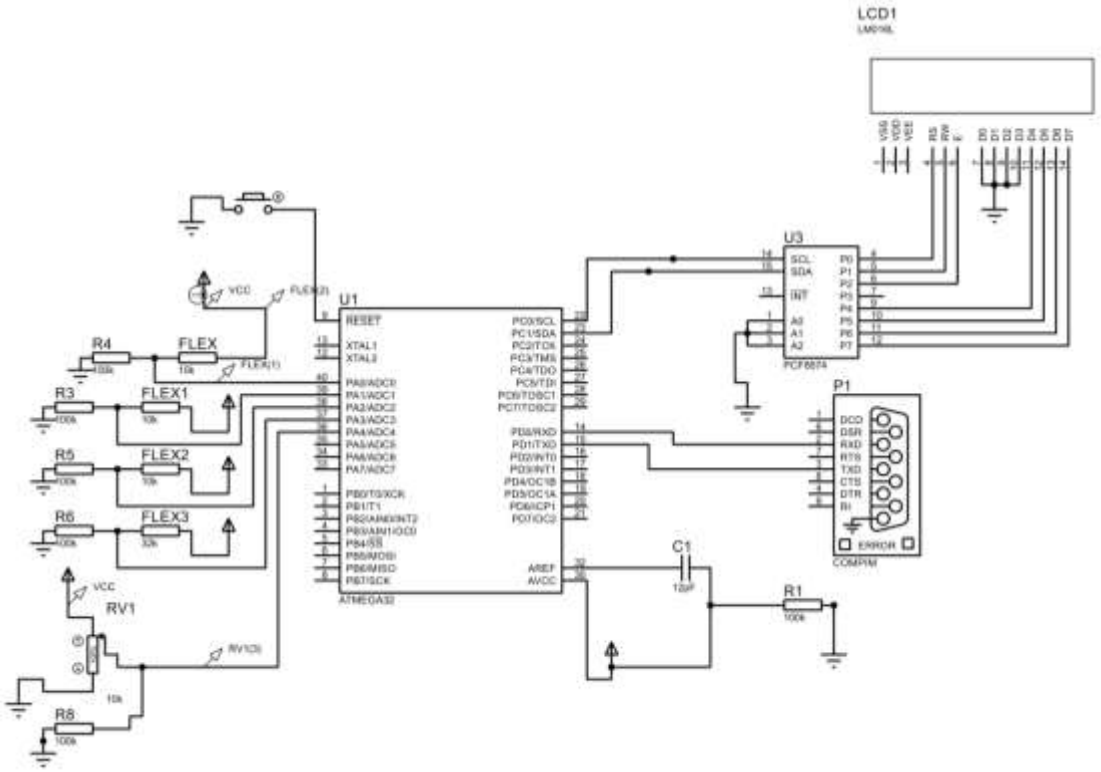
Abstract:

This project aims to convert flex sensor installed on fingers signs Into binary values that will be translated based on customized dictionary to strings that are pronounced by peripheral device.

Components:

- Flex sensors x5
- Atmega32 x1
- I2c LCD x1
- PL2303 USB TTL x1

Connections



Project Objectives (high detailed)

Project aims to produce text and voice based on hand fingers movement

System Design

Design Goals

- User-Friendly Interface
- Efficient Data Management
- Real-time respone

Proposed Software Architecture

Overview

For this project, a suitable architecture would be the Layered Architecture because in layered arch. It's easy to modify system components without affecting the whole system.
i.g. : if another LCD was chosen its code can be updated separately without affecting the whole system

1- HAL Layer:

- FLEX sensor
- LCD

2- MCAL Layer:

- TIMERS
- I2C PROTOCL
- UART PROTOCOL
- GI
- DIO
- ADC

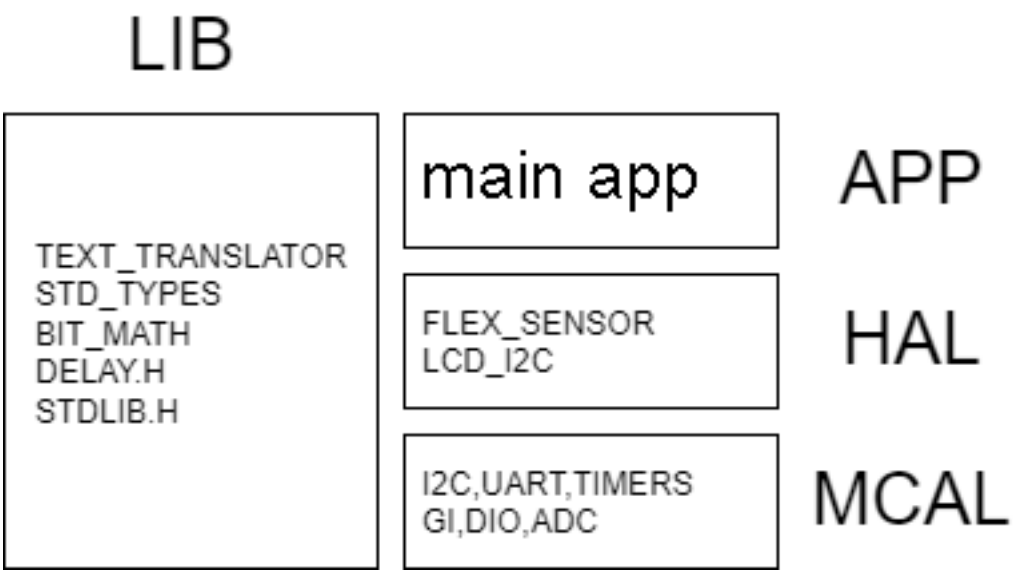
3- LIB Layer:

- STD TYPES
- BIT MATH
- TEXT TRANSLATOR

4- APP Layer:

- MAIN APPLICATION

Layers:

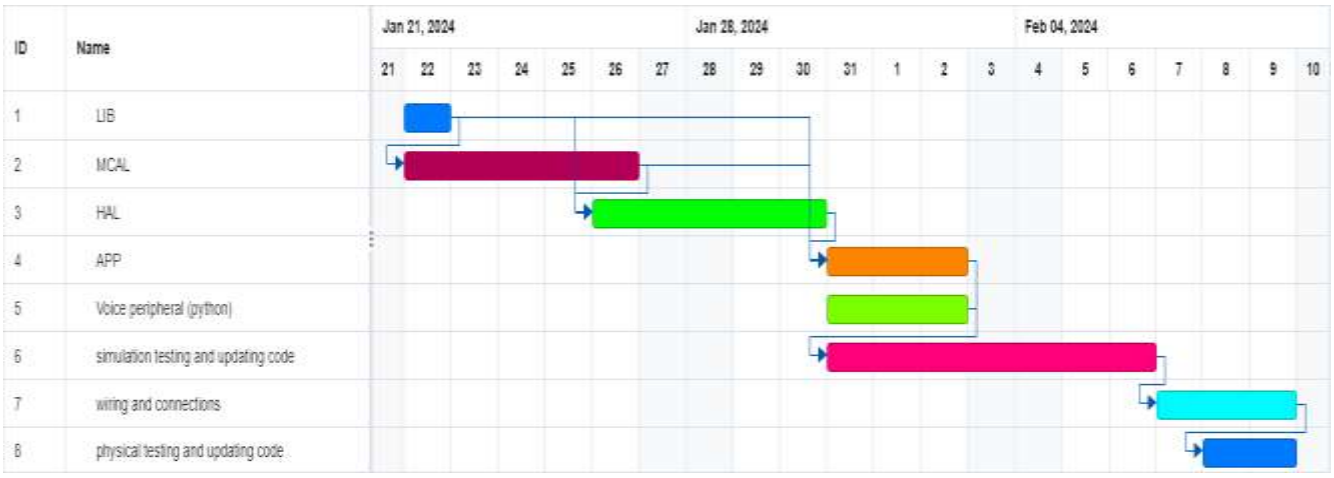


Proposed Software Model

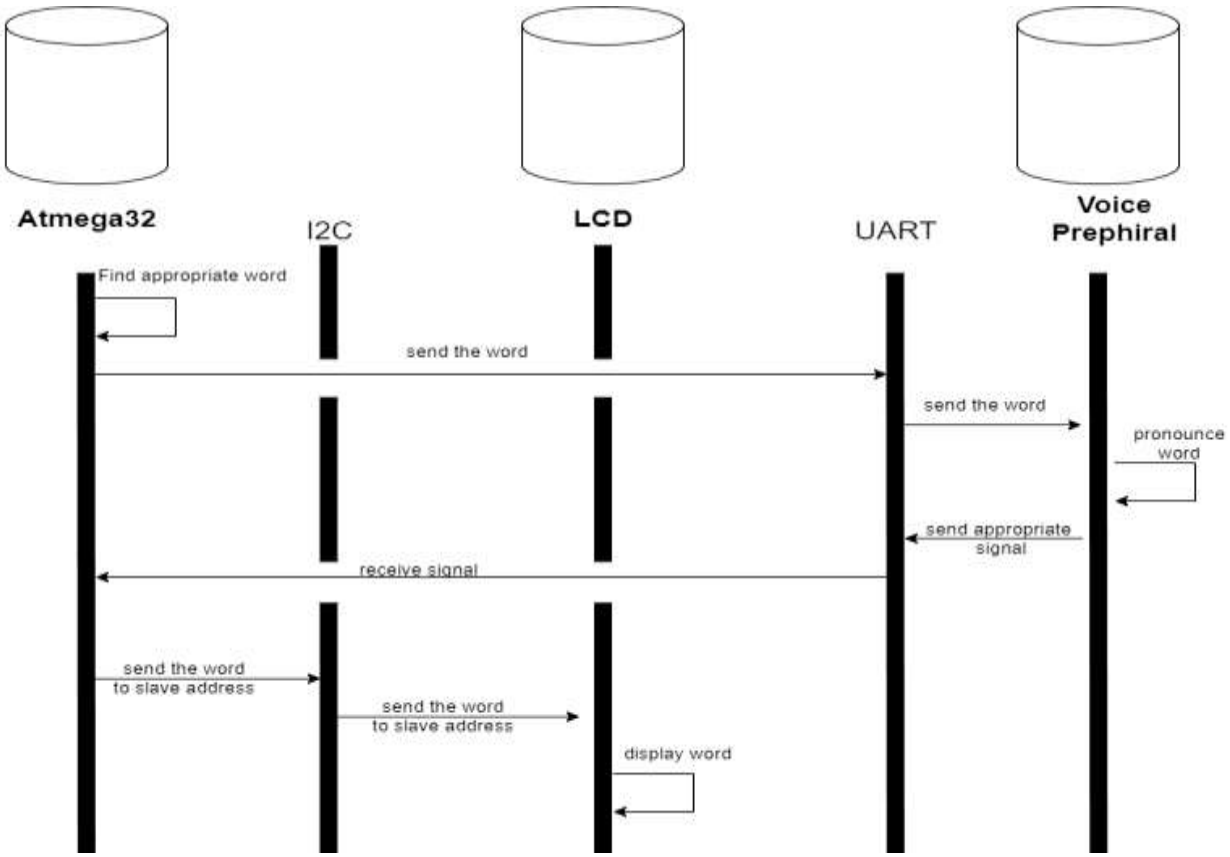
Overview

For this project, a suitable model would be AGILE model because the difficulty of determining required time for each phase and due to the lack of experience on the team , the code has to be changed a lot until it satisfies the project objectives

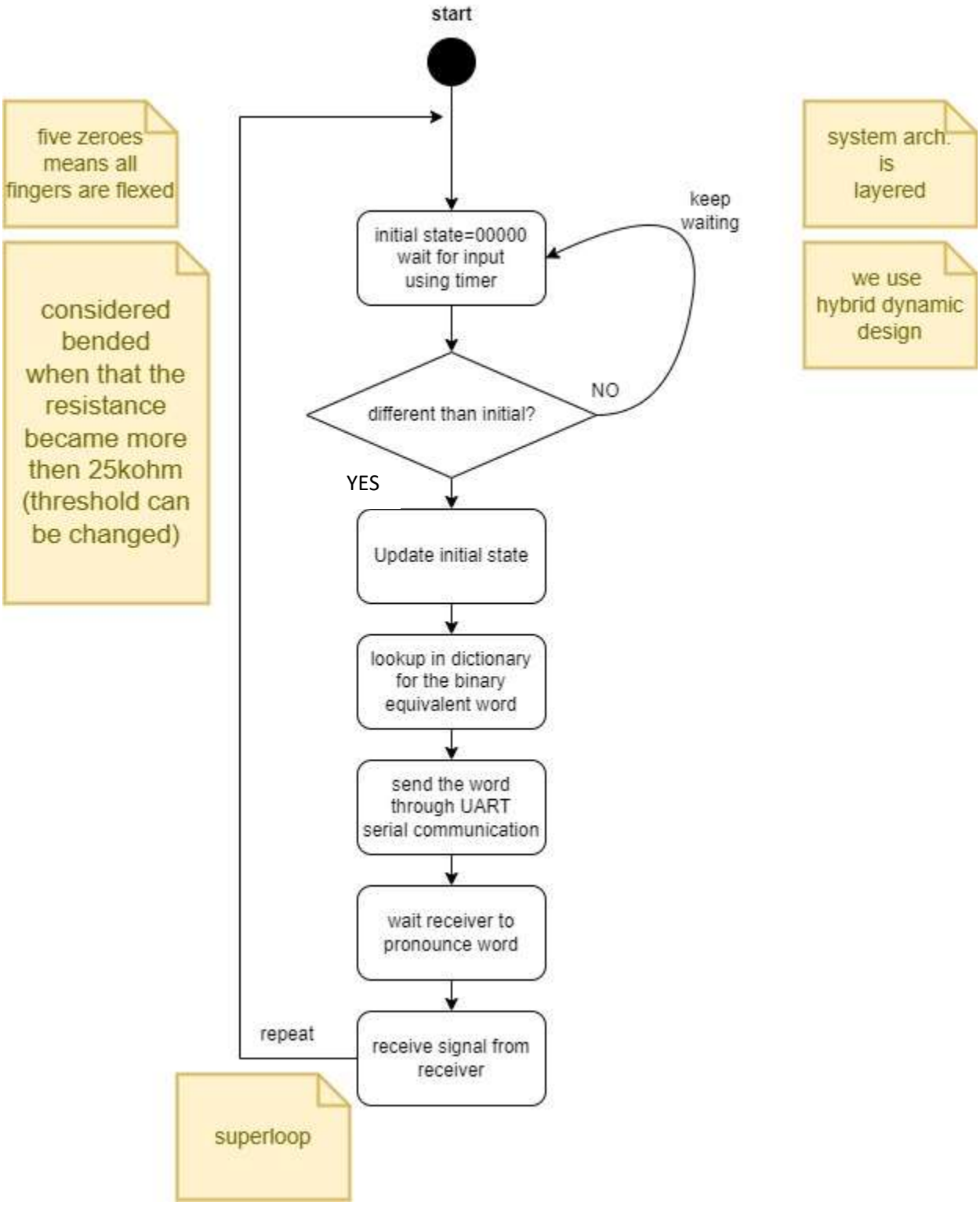
Gantt chart



Sequence diagram:



Activity diagram:



LIB LAYER

BIT_MATH.H

```
#ifndef _BIT_MATH_H
#define _BIT_MATH_H

#define SET_BIT(Reg, bitnum)       (Reg |=1<<bitnum)
#define CLR_BIT(Reg, bitnum)       (Reg &=~(1<<bitnum))
#define GET_BIT(Reg, bitnum)       ((Reg>>bitnum)& 0x01)
#define TOGGLE_BIT(Reg, bitnum)    (Reg ^=(1<<bitnum))

#define SET_BYTE(Reg,value)              ( Reg = value )

#define Conc_help(b7,b6,b5,b4,b3,b2,b1,b0) 0b##b7##b6##b5##b4##b3##b2##b1##b0
#define CONC_BIT(b7,b6,b5,b4,b3,b2,b1,b0) Conc_help(b7,b6,b5,b4,b3,b2,b1,b0)
#define BIT_IS_CLEAR(REG,BIT) ( !(REG & (1<<BIT)) )


#endif
```

TEXT_TRANSLATOR.H

```
#ifndef _TEXT_TRANSLATOR_H_
#define _TEXT_TRANSLATOR_H_
#include "STD_TYPES.h"

struct DictionaryEntry {
    u8 fingerCombination;
    u8* word;
};

u8* lookupWord(u8 fingerState, struct DictionaryEntry* dictionary, u8
dictionarySize);

u8* lookupWord(u8 fingerState, struct DictionaryEntry* dictionary, u8
dictionarySize) {

    for (int i = 0; i < dictionarySize; ++i) {

        if (dictionary[i].fingerCombination == fingerState) {
            return dictionary[i].word;
        }
    }
    return "Unknown"; // Return a default word for unknown combinations
}

#endif
```

LIB LAYER

STD_TYPES.H

```
#ifndef _STD_TYPES_H
#define _STD_TYPES_H

typedef unsigned char      u8;
typedef unsigned short int u16;
typedef unsigned long  int u32;

typedef signed char        s8;
typedef signed short int   s16;
typedef signed long  int   s32;

typedef float              f32;
typedef double             f64;

typedef void(*pf)(void);
#define PIN0_ID                0
#define PIN1_ID                1
#define PIN2_ID                2
#define PIN3_ID                3
#define PIN4_ID                4
#define PIN5_ID                5
#define PIN6_ID                6
#define PIN7_ID                7
#endif
```

LIB LAYER ENDS

HAL LAYER

FLEX_SENSOR

FLEX_SENSOR_CFG.H

File is empty

FLEX_SENSOR_PRIVATE.H

File is empty

FLEX_SENSOR_INTERFACE.H

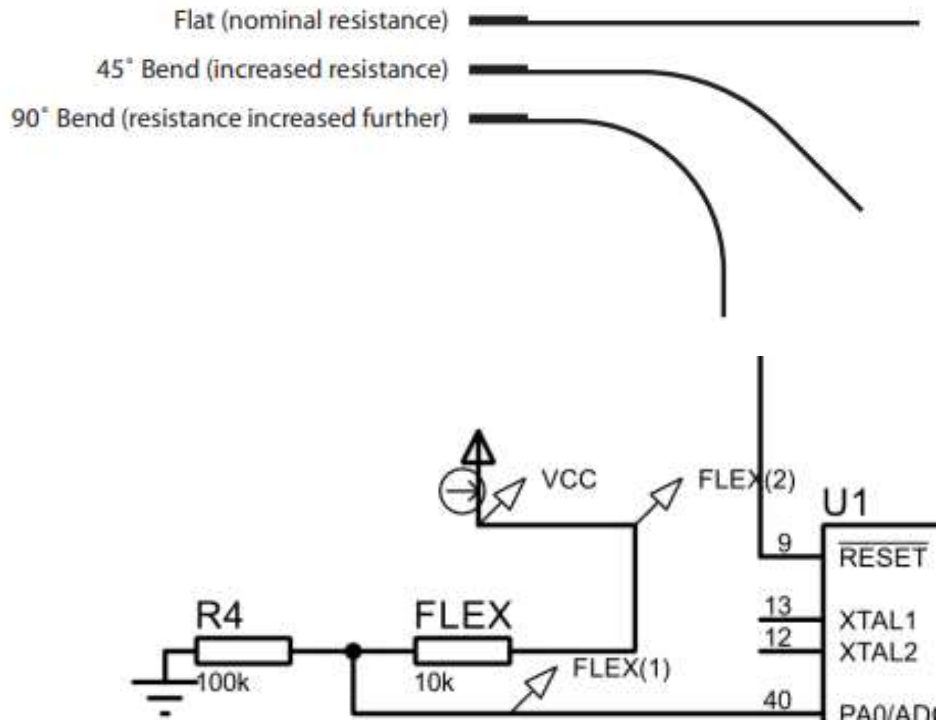
```
#ifndef FLEX_SENSOR_INTERFACE_H_
#define FLEX_SENSOR_INTERFACE_H_

u16 HFLEX_SENSOR_CALCULATE_RES(u16 Voutput,u16 Vinput,u32
fixed_resistance);

#endif
```

FLEX_SENSOR_PROGRAM.C

```
#include "../../LIB/STD_TYPES.h"


u16 HFLEX_SENSOR_CALCULATE_RES(u16 Voutput,u16 Vinput,u32
fixed_resistance){
	//voltage divider equationg is
	//Vout=Vin*(R2/(R1+R2) where R1 is the flex
	//hence
	//flex resestince = (fixed resistance * (Vin - Vout)) / Vout
	u16 flex=(fixed_resistance*(Vinput-Voutput))/Voutput;
	return flex;
}
```

Flex sensor is changing resistance that changes

Based on the bending state



To determine the values of bending state it's important to use voltage divider circuit

In order to make Vout distinguishable and then we calculate the resistance of flex sensor using

Following equation:

$$\underline{Vout}=\underline{Vin}*(R2/(R1+R2)$$

Where

R1 represents (flex sensor)

R2 represents Fixed resistance (100kohm in this case)

Vin represents input voltage (5v in this case)

Vout represents output voltage (voltage read by microcontroller)

After calculating the resistance value of FLEX SENSOR

We can then use it in a simple function with customized threshold

For example if **threshold** was **25**kohm

And the function returned resistance value of flex sensor = **26**kohm

It will be considered bended (1)

Moreover if the function returned resistance value of flex sensor = **24**kohm

It will be considered flexed (0)

With this simple function while using 5 flex sensors we can distinguish

2^5 different moves for the fingers

However due to the accurate determination of the flex sensor resistance value

It can be later used to distinguish more than binary states (flexed,bended) (0,1)

But due to complexity and time we had it's decided to be binary instead

FLEX SENSOR ENDS

HAL LAYER

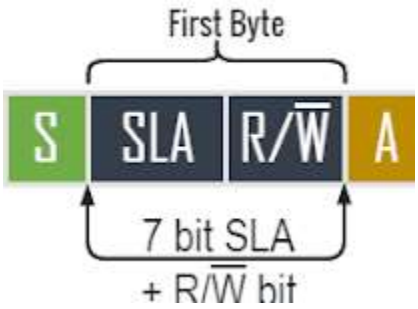
LCD_I2C

LCDI2C_CFG.H

```
#ifndef _LCD_CFG_

#define LCD_RS_PIN_ID            0
#define LCD_RW_PIN_ID            1
#define LCD_E_PIN_ID             2
//LCD COMMANDS:
#define LCD_GO_TO_HOME           0x02
#define LCD_CLEAR_COMMAND        0x01
#define LCD_BACKLIGHT            0x08
#define LCD_CGRAM_START          0x40
#define LCD_SET_CURSOR_LOCATION  0x80

#define slave_address_fixed_bits 0x27 << 1

#endif
```

File contains definitions of commands
And the address of the slave

Address is shifted by 1 to leave R/$\overline{W}$ bit



LCDI2C_PRIVATE.H

File is empty

LCDI2C_INTERFACE.H

```
#ifndef LCD_H_
#define LCD_H_

#include "../../LIB/STD_TYPES.h"



#define ERROR       0
#define SUCCESS     1



//read an write for i2c
#define WRITE       0
#define READ        1

static u8 PCF8574_write(u8 data);
void LCD_init(void);
void LCD_sendWithHighEn();
void LCD_sendWithLowEn();
void LCD_sendCommand(u8 command);
void LCD_displayCharacter(u8 data);
void LCD_displayString(const u8 *Str);
void LCD_clearScreen(void);
void LCD_Heart(void);

#endif /* LCD_H_ */
```

STD_TYPES.h is included to use customized data types (u8 for example)

Error,success,read,write are Defined to make the code more readable

LCDI2C_PROGRAM.C

I2c protocol will be explained later when we come to it in the code

```
#include <util/delay.h>
#include "../../LIB/BIT_MATH.h"
#include "../../MCAL/I2C/TWI_interface.h"
#include "LCDI2C_Interface.h"
#include "LDCI2C_CFG.h"



u8 global_LCD=0;//performs like a buffer

static u8 PCF8574_write(u8 data)
{
	_delay_ms(1);
	data|=LCD_BACKLIGHT
	TWI_start();
	if (TWI_getStatus() != TWI_START)
		return ERROR;

	TWI_writeByte((slave_address_fixed_bits) | WRITE);
	if (TWI_getStatus() != TWI_MT_SLA_W_ACK)
		return ERROR;

	TWI_writeByte(data);
	if (TWI_getStatus() != TWI_MT_DATA_ACK)
			return ERROR;

	TWI_stop();

	return SUCCESS;
}

void LCD_init(void)
{
	_delay_ms(30);
	TWI_init();

	LCD_sendCommand(LCD_GO_TO_HOME);
	LCD_sendCommand(LCD_CLEAR_COMMAND);
	LCD_HEART();

}
```

global_LCD variable as a buffer to save the data it will be overwritten during the program run many times

PCF8574 is i/o expander that is connected the LCD Bytes will be sent to it using i2c protocol

LCD_BACKLIGHT is sent with data However it doesn't affect the data Is it performs on pin that's not Connected to the LCD

in init function it delay(30)milli second at first then it sends command to LCD to put the cursor on the first position and then it sends command to clear the screen as resetting the ATMEGA32 doesn't reset the LCD.

Moreover lcd_heart(); is function that saves heart shape on CGRAM location 0

LCDI2C_PROGRAM.C continued

```
void LCD_sendWithHighEn(){
SET_BIT(global_LCD,LCD_E_PIN_ID);
PCF8574_write(global_LCD);
_delay_ms(1);

}
void LCD_sendWithLowEn(){
CLR_BIT(global_LCD,LCD_E_PIN_ID);
PCF8574_write(global_LCD);
_delay_ms(1);
}
void LCD_sendCommand(u8 command)
{
//COMMAND MODE
_delay_ms(1);
CLR_BIT(global_LCD,LCD_RS_PIN_ID);
CLR_BIT(global_LCD,LCD_RW_PIN_ID);
LCD_sendWithHighEn();
//high 4 bits
global_LCD = (global_LCD & 0x0F) | (command & 0xF0);
LCD_sendWithHighEn();
LCD_sendWithLowEn();
//low 4 bits
global_LCD =(global_LCD&0x0F)|((command & 0x0F)<<4);
LCD_sendWithHighEn();
LCD_sendWithLowEn();
}
```

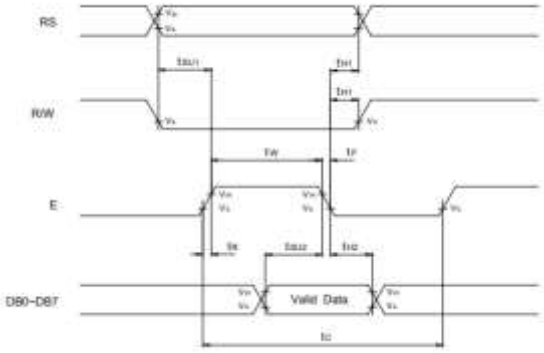
To send command on LCD

RS,ES PINS value must be 0

Then E pin must be enabled

Then the command is sent

And then the E pin is disabled



Its repeated as in 4-bits mode

We have to send first high bits

Then we send last low 4 bits

LCDI2C_PROGRAM.C continued

```
void LCD_displayCharacter(u8 data)
{//SAME as command but we set RS_PIN ->1
	_delay_ms(1);
	//DATA MODE
	SET_BIT(global_LCD,LCD_RS_PIN_ID);
	CLR_BIT(global_LCD,LCD_RW_PIN_ID);
	//high 4 bits
	global_LCD = (global_LCD & 0x0F)|(data&0xF0);
	LCD_sendWithHighEn();
	LCD_sendWithLowEn();
	//LOW 4 bits
	global_LCD =(global_LCD&0x0F)|((data&0x0F)<< 4);
	LCD_sendWithHighEn();
	LCD_sendWithLowEn();
}


void LCD_displayString(const u8 *Str)
{
	u8 i = 0;
	while(Str[i] != '\0')
	{
		LCD_displayCharacter(Str[i]);
		i++;
	}

}

void LCD_clearScreen(void)
{
	LCD_sendCommand(LCD_CLEAR_COMMAND);
}
void LCD_HEART(void) {
    // Custom character data for heart shape
    const u8 heartCharMap[8] = {
        0b00000,
        0b01010,
        0b11111,
        0b11111,
        0b01110,
        0b00100,
        0b00000,
        0b00000
    };

    LCD_sendCommand(LCD_CGRAM_START);

    // Load the custom character data into CGRAM
    for (u8 i = 0; i < 8; ++i) {
        LCD_displayCharacter(heartCharMap[i]);
    }
}
```

LCD display character works the same
As command yet it enables RS pin

LCD display string sends all characters in
String to LCD display character function

LCD clear screen sends command to clear
The lcd screen

Lcd heart make heart shaped character
And save it on cgram location 0

HAL LAYER ENDS

MCAL LAYER

ADC (ANALOG DIGITAL CONVERTOR)

```
#define ADMUX  (*(volatile u8 *)0x27)
#define ADCSRA (*(volatile u8 *)0x26)
#define ADCH   (*(volatile u8 *)0x25)
#define ADCL   (*(volatile u8 *)0x24)

#define ADCLH  (*(volatile u16 *)0x24)

#define ADC_CHANNEL_MASK 0b11100000
```

These lines define macros for accessing specific memory addresses, likely representing the ADC-related registers in the microcontroller.

ADMUX, ADCSRA, ADCH, and ADCL are defined as pointers to volatile unsigned 8-bit integers (u8). They likely correspond to the ADC Multiplexer, ADC Control and Status Register A, and the high and low bytes of the ADC result registers.

(volatile u8 *): This part of the expression is a type cast. It tells the compiler how to interpret the data at the memory address. In this case, it is cast to a volatile unsigned 8-bit variable (u8 means an unsigned 8-bit integer)

So, altogether, (*(volatile u8 *)(hex_address)) means "interpret the data at memory address (hex_address) as a volatile unsigned 8-bit integer and retrieve its value."

```
void MADC_voidInit (void)
{
	SET_BIT(ADMUX,6);
	CLR_BIT(ADMUX,7);
	CLR_BIT(ADMUX,5);
	CLR_BIT(ADCSRA,5);
	CLR_BIT(ADCSRA,0);
	SET_BIT(ADCSRA,1);
	SET_BIT(ADCSRA,2);
	SET_BIT(ADCSRA,7);

}
```

This function initializes the ADC module with specific configurations.

It sets up the voltage reference to AVCC with an external capacitor at the AREF pin, chooses right adjustment for the result, disables auto-triggering, selects a clock division factor of 64, and finally enables the ADC.

```
u16 MADC_u16GetDigitalvalue (ADC_CHANNELS A_AdcChannel)
{
	u16 local_u16DigitalValue=0;
	if (A_AdcChannel <32)
	{
		ADMUX &= ADC_CHANNEL_MASK;
		ADMUX |= A_AdcChannel;
		SET_BIT(ADCSRA,6);
		while (GET_BIT(ADCSRA,4) == 0);
		SET_BIT(ADCSRA,4);
		local_u16DigitalValue = ADCLH;
	}
	return local_u16DigitalValue;
}
```

This function reads the digital value from a specified ADC channel.

It checks if the given ADC channel is within a valid range (less than 32-number of ADC channels in atmega32),

clears the lower 5 bits of ADMUX ,this step ensures that the previous ADC channel selection bits are set to 0 before updating them

then sets the selected ADC channel, starts the conversion, waits for the ADC completion using polling, clears the ADC interrupt flag(the microcontroller can be configured to generate an interrupt when the conversion is complete.), and then reads the 16-bit result from ADCLH. The result is then returned.

ADC ENDS

DIO (Digital Input/Output)

```
#define PORTA_REG   (*((volatile u8 *)0x3B))
#define DDRA_REG    (*((volatile u8 *)0x3A))


#define PORTB_REG   (*((volatile u8 *)0x38))
#define DDRB_REG    (*((volatile u8 *)0x37))


#define PORTC_REG   (*((volatile u8 *)0x35))
#define DDRC_REG    (*((volatile u8 *)0x34))

#define PORTD_REG   (*((volatile u8 *)0x32))
#define DDRD_REG    (*((volatile u8 *)0x31))

void MDIO_voidInit()
{

      DDRA_REG=CONC_BIT(PA7_INITIAL_DIRECTION,PA6_INITIAL_DIRECTION,PA5_INITIAL_DIRECTIO
N,PA4_INITIAL_DIRECTION

,PA3_INITIAL_DIRECTION,PA2_INITIAL_DIRECTION,PA1_INITIAL_DIRECTION,PA0_INITIAL_DIRECTION)
;

..etc

      PORTA_REG=CONC_BIT(PA7_INITIAL_VALUE,PA6_INITIAL_VALUE,PA5_INITIAL_VALUE,PA4_INITI
AL_VALUE

,PA3_INITIAL_VALUE,PA2_INITIAL_VALUE,PA1_INITIAL_VALUE,PA0_INITIAL_VALUE);

..etc
```

Only one function that initialize the direction and initial values of pins of microcontroller base on configuration file no need for more functions under the scope of this project

DIO ENDS

GI (General Interrupt)

```
#define SREG (*(volatile u8 *)0x5F)

#define GLOBAL_INTERRUPT_ENABLE_BIT  7
void MGI_voidEnable (void)
{
	SET_BIT(SREG,GLOBAL_INTERRUPT_ENABLE_BIT);
}
void MGI_voidDisable (void)
{
	CLR_BIT(SREG,GLOBAL_INTERRUPT_ENABLE_BIT);
}
```

General Interrupt must be enabled to perform any interrupt on the system. In this project interrupt is needed (TIMERS INTERRUPT)

GI ENDS

TWI (i2c protocol)

```
void TWI_init(void)
{
    /* Bit Rate: 400.000 kbps using (pre-scaler=1 -->TWPS=00) and F_CPU=8Mhz
*/
    TWBR = 0x02;
       TWSR = 0x00;

    /* Two Wire Bus address my address if any master device want to call me:
     * bits(7:1) 0x1 (used in case this MC is a slave device)
     * bit 0: General Call Recognition: Off
     */
    TWAR = 0b00000010; // my address = 0x01 :)

    TWCR = (1<<TWCR_TWEN); /* enable TWI */
}
```

Configures the bit rate to achieve a specific communication speed (400.000 kbps in this case) with a pre-scaler of 1.

`TWSR = 0x00;`
To indicate that prescaler=1

Sets the Two-Wire Bus address. In this example, it's configured as 0x02.

Enables the TWI (module).

```
void TWI_start(void)
{
    /*
        * Clear the TWINT flag before sending the start bit TWINT=1 (not cleared
automatically)
        * send the start bit by TWSTA=1
        * Enable TWI Module TWEN=1
        */
    TWCR = (1 << TWCR_TWINT) | (1 << TWCR_TWSTA) | (1 << TWCR_TWEN);

    /* Wait for TWINT flag set in TWCR Register (start bit is send
successfully) */
    while(BIT_IS_CLEAR(TWCR,TWCR_TWINT));
}
```

Clears the TWINT flag, sets the TWSTA (Start) bit, and enables the TWI module, initiating a start condition.

Waits until the TWINT flag is set, indicating that the start condition has been sent successfully.

```
void TWI_stop(void)
{
    /*
        * Clear the TWINT flag before sending the stop bit TWINT=1
        * send the stop bit by TWSTO=1
        * Enable TWI Module TWEN=1
        */
    TWCR = (1 << TWCR_TWINT) | (1 << TWCR_TWSTO) | (1 << TWCR_TWEN);
}
```

Clears the TWINT flag, sets the TWSTO (Stop) bit, and enables the TWI module, initiating a stop condition

```c
void TWI_writeByte(u8 data)
{
    /* Put data On TWI data Register */
    TWDR = data;
    /*
        * Clear the TWINT flag before sending the data TWINT=1
        * Enable TWI Module TWEN=1
        */
    TWCR = (1 << TWCR_TWINT) | (1 << TWCR_TWEN);
    /* Wait for TWINT flag set in TWCR Register(data is send successfully) */
    while(BIT_IS_CLEAR(TWCR,TWCR_TWINT));
}
```

Places the data to be transmitted into the TWI Data Register.

Clears the TWINT flag, enables the TWI module, and initiates the data transmission.

Waits until the TWINT flag is set, indicating that the data has been sent successfully.

```c
u8 TWI_readByteWithACK(void)
{
      /*
        * Clear the TWINT flag before reading the data TWINT=1
        * Enable sending ACK after reading or receiving data TWEA=1
        * Enable TWI Module TWEN=1
        */
    TWCR = (1 << TWCR_TWINT) | (1 << TWCR_TWEN) | (1 << TWCR_TWEA);
    /* Wait for TWINT flag set in TWCR Register (data received successfully) */
    while(BIT_IS_CLEAR(TWCR,TWCR_TWINT));
    /* Read Data */
    return TWDR;
}
```

Clears the TWINT flag, enables the TWI module, and sends an acknowledge bit after reading the data

Waits until the TWINT flag is set, indicating that the data has been received successfully.

Returns the received data.

```c
u8 TWI_readByteWithNACK(void)
{
       /*
        * Clear the TWINT flag before reading the data TWINT=1
        * Enable TWI Module TWEN=1
        */
    TWCR = (1 << TWCR_TWINT) | (1 << TWCR_TWEN);
    /* Wait for TWINT flag set in TWCR Register (data received successfully) */
    while(BIT_IS_CLEAR(TWCR,TWCR_TWINT));
    /* Read Data */
    return TWDR;
}
```

Same as previous yet the function doesn't include enabling ACK

```c
u8 TWI_getStatus(void)
{
    u8 status;
    /* masking to eliminate first 3 bits and get the last 5 bits (status bits) */
    status = TWSR & 0xF8;
    return status;
}
```

This function retrieves the status of the TWI communication by reading the TWSR register and masking out the irrelevant bits.

Reads the TWSR register and masks out the first three bits, leaving only the last five bits representing the status bits.

Returns the status of the TWI communication.

TWI(I2C) ENDS

TIMERS

```
void (*TIMER0_CTC_CALLBACK)(void) = NULL;
void MTIMER0_voidInit (void)
{
	// CTC Mode
	SET_BIT(TCCR0,3);
	CLR_BIT(TCCR0,6);

	// Enable CTC Interrupt
	SET_BIT(TIMSK,1);
	CLR_BIT(TIMSK,0);

	// Set OCR0 Value
	OCR0 = OCR0_VALUE;

	// Start Timer by setting its clock
	TCCR0 &= 0b11001000;
	TCCR0 |= (TIMER0_CLK | (CTC_OC0_MODE <<4));

}
void MTIMER0_voidSetCTCCallback (void (*A_PtrToFunc)(void))
{
	if (A_PtrToFunc != NULL)
	{
		TIMER0_CTC_CALLBACK = A_PtrToFunc;
	}
}

void __vector_10(void) __attribute__((signal));
void __vector_10(void)
{
	if (TIMER0_CTC_CALLBACK != NULL)
	{
		TIMER0_CTC_CALLBACK();
	}
}
```

Configures TIMER0 for CTC mode by setting the WGM01 bit and clearing the WGM00 bit in the TCCR0 register.

Enables the CTC interrupt by setting the OCIE0 bit (bit 1) in the TIMSK register. It also clears the TOIE0 bit (bit 0) to disable the overflow interrupt.

Sets the Compare Match value by assigning the value of OCR0_VALUE to the OCR0 register.

Configures the clock source for TIMER0 by clearing the lower three bits of TCCR0 and then setting them based on TIMER0_CLK. It also sets the CTC mode action on compare match based on CTC_OC0_MODE.

```
void __vector_10(void) __attribute__((signal))
```

This is the interrupt service routine (ISR) for the CTC interrupt.

```
void MTIMER0_voidSetCTCCallback(void (*A_PtrToFunc)(void))
```

This function sets the callback function to be executed when the CTC interrupt occurs.

Checks if the provided function pointer is not NULL and assigns it to TIMER0_CTC_CALLBACK. This allows external code to register a callback function to be executed on CTC interrupt.

```
__attribute__((signal))
```

is an attribute applied to the function, indicating that it is an interrupt service routine. Which ensures that the compiler understands the function's intended role as an interrupt service routine.

TIMERS ENDS

UART

```
void MUART_voidInit(void)
{
	u16 local_u16BaudRate = BAUD_RATE_EQUATION;

	// Set Baudrate
	UBRRL = (u8) local_u16BaudRate;
	UBRRH = (u8) (local_u16BaudRate >> 8);

	// Character Size 8-bit
	CLR_BIT(UCSRB,2);

	// Character Size 8-bit
	// Mode Asynchronous operation
	// Parity Disabled
	// 1 Stop Bit
	UCSRC = CONC_BIT(1,0,0,0,0,1,1,0);

	// Enable Receiver
	// Enable Transmitter
	SET_BIT(UCSRB,3);
	SET_BIT(UCSRB,4);

}
```

This function initializes the UART module.

Configures the Baud Rate by setting the low and high bytes of the UBRR register.

Sets the character size to 8 bits and configures the mode for asynchronous operation, with no parity and one stop bit.

Enables the UART receiver and transmitter.

```
void MUART_voidSendByteSyncBlocking (u8 A_u8DataByte)
{
	while(GET_BIT(UCSRA,5)==0);//0=not empty
	UDR = A_u8DataByte;
	while(GET_BIT(UCSRA,5)==0);
	SET_BIT(UCSRA,6);

}
```

This function sends a byte synchronously and blocks until the operation is complete.

Waits until the UART Data Register is empty, indicating that it is ready to accept new data.

Sets the data byte into the UART Data Register to start transmission.

Waits until the UART Data Register is empty again, indicating that the transmission is complete.

Clears the transmit interrupt flag.

```
void MUART_voidSendStringSyncBlocking (u8 *A_pu8String)
{
	while (*A_pu8String > 0)
	{
		MUART_voidSendByteSyncBlocking(*A_pu8String++);
	}
}
```

Send string using **`MUART_voidSendByteSyncBlocking`** function

```
u8 MUART_u8ReadByteSyncBlocking (void)
{
	// Wait for Rx Flag
	while (GET_BIT(UCSRA,7)==0);//0=not finished
	return UDR;
}
```

This function reads a byte synchronously and blocks until the operation is complete

Waits until the Receive Complete flag is set, indicating that a byte has been received then it returns the data

```
u8* MUART_u8ReadStringSyncBlocking(void) {
    u8* LocalStringRead = (u8 *)malloc(sizeof(u8));
    u8 LocalStringLastByte = MUART_u8ReadByteSyncBlocking();
    u8 LEN = 0;

    while (LocalStringLastByte != '\n' && LocalStringLastByte !=NULL ) {
        LocalStringRead = (u8 *)realloc(LocalStringRead, (LEN + 1) * sizeof(u8));
        LocalStringRead[LEN] = LocalStringLastByte;
        LEN++;
        LocalStringLastByte = MUART_u8ReadByteSyncBlocking();
    }

    LocalStringRead = (u8 *)realloc(LocalStringRead, (LEN + 1) * sizeof(u8));
    if (LocalStringRead == NULL) {
        free(LocalStringRead);
        return NULL;
    }

    LocalStringRead[LEN] = '\0';
    return LocalStringRead;
    /* don't forget to free the variable that holds this function result */
}
```

This function reads a string synchronously and blocks until the operation is complete.

Allocates memory to store the received string.

Reads characters until a newline character or NULL is encountered, dynamically reallocating memory as needed then it adds one more character space of memory to the allocated memory to add (‘\0’)

Terminates the string and returns the dynamically allocated memory (which is the received string).

MCAL ENDS

APPLICATION LAYER

MAIN.C

```
struct DictionaryEntry
dictionary[] = {
    {0b00000, "PROJECT"},
    {0b00001, "banana"},
    {0b00010, "orange"},
    {0b00011, "grape"},
    {0b00100, "kiwi"},
    {0b00101, "pear"},
    {0b00110, "melon"},
    {0b00111, "cherry"},
    {0b01000, "You"},
    {0b01001, "lime"},
    {0b01010, "blueberry"},
    {0b01011, "strawberry"},
    {0b01100, "raspberry"},
    {0b01101, "blackberry"},
    {0b01110, "pineapple"},
    {0b01111, "mango"},
    {0b10000, "watermelon"},
    {0b10001, "peach"},
    {0b10010, "plum"},
    {0b10011, "apricot"},
    {0b10100, "nectarine"},
    {0b10101, "fig"},
    {0b10110, "pomegranate"},
    {0b10111, "coconut"},
    {0b11000, "kiwi"},
    {0b11001, "grapefruit"},
    {0b11010, "papaya"},
    {0b11011, "avocado"},
    {0b11100, "cantaloupe"},
    {0b11101, "honeydew"},
    {0b11110, "dragonfruit"},
    {0b11111, "passionfruit"},
};
```

DICTIONARY OF 32 DIFFERENT WORDS

```
#define ISBEND(value, threshold) ((value) > (threshold) ? 1 : 0)
#define ISEQUAL(value, value2) ((value) == (value2) ? 1 : 0)
#define TO_BINARY(b1, b2, b3, b4, b5) \
    (((b1) << 4) | ((b2) << 3) | ((b3) << 2) | ((b4) << 1) | (b5))

u8 GET_STATS(){
      // Read in mv
      for (u8 i=0;i<5;i++){
            local_u16DigitalValue = MADC_u16GetDigitalvalue(i);
            local_u16AnalogValue  = (local_u16DigitalValue * 5000UL) / 1024;

      hand_analog_reads[i]=HFLEX_SENSOR_CALCULATE_RES(local_u16AnalogValue,Vin,f
ixed_resistor)/1000;

      }
      stats = TO_BINARY(
          ISBEND(hand_analog_reads[0], 25),
          ISBEND(hand_analog_reads[1], 25),
          ISBEND(hand_analog_reads[2], 25),
          ISBEND(hand_analog_reads[3], 25),
          ISBEND(hand_analog_reads[4], 25)
            );
      return stats;
}
```

This function determines the five hands signals

Digital*5000/1024 =

digital *5*1000/1024

digital=digital value

*5=Vin

*1000=to read in millivolt

/1024= maximum possible digital value 2^{10}

The reads are saved in array

Then these reads are converted to binary with threshold=25 (kohm)

Where if any read is larger it'll be considered bended (1 in binary)

```
void Timer0_CTC() {
    static u16 counter = 0;
    counter++;
    u8 *text;
    u8 *receiver;

    if (counter == 30000) {//10000=2 seconds
        counter = 0;
        stats = GET_STATS();

        switch (ISEQUAL(stats, pre_stats)) {
            case 1:
                // Do nothing as no changes occurred
                break;
            case 0:
                text = lookupWord(stats, dictionary, 32);
                LCD_clearScreen();
                MUART_voidSendStringSyncBlocking(text);
                receiver = MUART_u8ReadStringSyncBlocking();

                if (receiver) {
                            LCD_displayString(receiver);

                }
                else {
                    LCD_displayString("Error");
                }

                free(receiver);
                pre_stats = stats;
                break;
        }
    }
}
```

Called by interrupt every 200ticks (prescaler is 8 with 8MHZ clock hence tick=1microsecond) to increment counter.

If counter reached 5000 (1 second in this case) it determines if the hand states changed, moreover if the state changed it will communicate through UART protocol with another peripheral that supposed to convert the meant word to voice and play it.

The UART functions are synchrounous blocking so it will keep waiting until the peripheral transmit data to avoid delays (new word displayed before the previous word is pronounced)

If the stats read equals the previous stats this function will do nothing until the counter is called again.

briefly the function does nothing if the hand signals didn't change to avoid displaying and pronouncing the same word consecutively

```
int main (void)
{
	MDIO_voidInit();
	MADC_voidInit();
	MGI_voidEnable();
	LCD_init();
	LCD_displayString("I ");
	LCD_displayCharacter(0);
	LCD_displayString(" IMT");
	_delay_ms(10000);
	MTIMER0_voidSetCTCCallback(Timer0_CTC);
	MTIMER0_voidInit();
	MUART_voidInit();
	while(1)
	{

	}
}
```

The main function initialize all layers and display a welcome message for 10 seconds then goes for superloop that will be interrupted frequantly by the timer

UART Receiver peripheral program

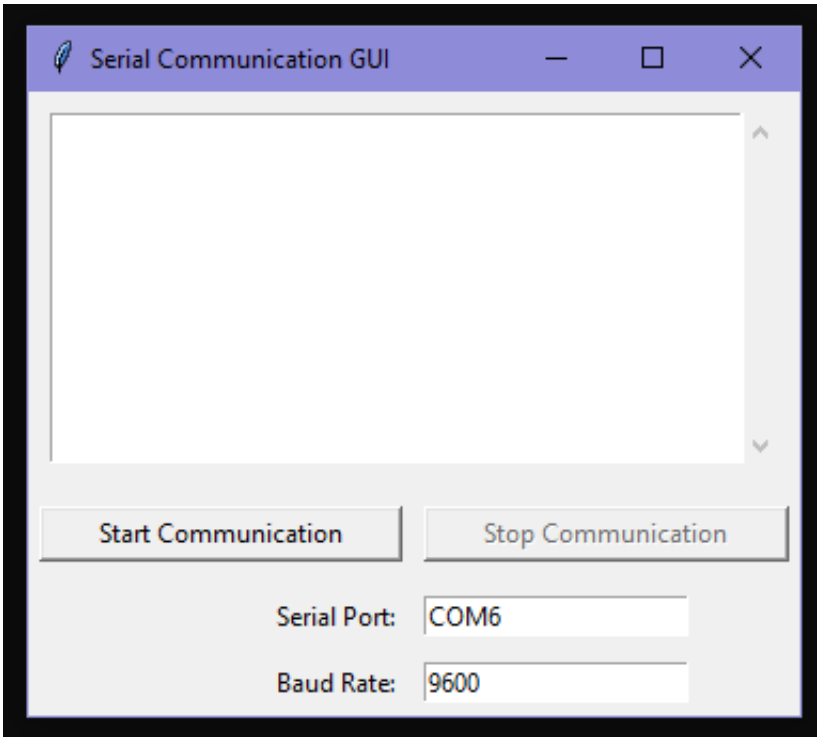
```
import serial
import time
import pyttsx3
def send_and_read(port, baudrate):
    myserial = serial.Serial(port=port, baudrate=baudrate)

    if myserial.is_open:
        # Initial exchange to synchronize communication
        myserial.write("start_com\n");
        time.sleep(1)  # Give the AVR some time to process

        while True:
            size = myserial.inWaiting()
            if size:
                data = myserial.read(size)
                text_to_speak = str(data, 'utf-8')
                print(text_to_speak)
                txt2speech(text_to_speak)
                text_to_speak=text_to_speak+'\n'
                # Send a response back to the
microcontroller
                myserial.write(text_to_speak.encode())

            time.sleep(1)
def txt2speech(text):
    engine = pyttsx3.init()
    engine.say(text)
    engine.runAndWait()

send_and_read('COM6', 9600)
```

This python code opens

Desired port and starts communication

The code works but it was enhanced to be easier to use

By making simple GUI using tkinter library



Test Scenarios

Test Case 1: Initialization

- **Description:** Ensure that the initialization of digital I/O, ADC, global interrupts, and LCD is successful.
- **Steps:**
    1. Run the program.
    2. Observe if there are any initialization errors or crashes.
    3. Confirm that the LCD displays the expected initial message.

Test Case 2: Sensor Readings

- **Description:** Verify that the program correctly reads analog sensor values.
- **Steps:**
    1. Provide known analog values for each sensor.
    2. Run the program.
    3. Check if the **GET_STATS** function returns the expected binary representation based on provided analog values.

Test Case 3: Timer Interrupt

- **Description:** Confirm that the timer interrupt (**Timer0_CTC**) works as expected.
- **Steps:**
    1. Set a breakpoint in the **Timer0_CTC** function.
    2. Run the program.
    3. Check if the timer interrupt is triggered after the specified interval.
    4. Observe if the LCD updates correctly based on sensor changes.

Test Case 4: UART Communication

- **Description:** Ensure that UART communication functions as intended.
- **Steps:**
    1. Provide a known set of sensor statistics.
    2. Run the program.
    3. Check if the program sends the correct data over UART.
    4. Simulate receiving data over UART and observe if the LCD displays the received data.

PROJECT ENDS