I2C Laboratory

MICROCONTROLLER I²C INTERFACING

**Purpose**
- To introduce the students to the I²C protocol standard
- To develop familiarity with how to connect devices using the I²C protocol.
- To reinforce C program standards and the proper way to documentation software code.

**Learning Objectives:**
Upon successful completion of this laboratory experiment, the student will be able to:
1. Identify what a ‘Master’ controller is in I²C communication.
2. Explain what the clock is used for in I²C communication.
3. Describe where to find specification for the I²C communication protocol.
4. Describe the differences and similarities between RS232 and I²C communication protocols.
5. Calculate and set the Baud rate for the Atmel ATMEGA 128 processor.
6. Apply knowledge from reading assignment and set the clock speed of the Atmel processor.
7. Verify using a digital oscilloscope that the data and clock signals being transmitted from the master microcontroller and that the signal transmitted matches the data to be sent.
8. Verify the data rate of the I²C signal based on digital scope capture information.
9. Determine at which I²C communication rate, if any, will 30% of the microcontroller resources be used.
10. Classify the differences between hardware and software implementations of I²C communication protocol on a microcontroller, and the advantages and disadvantages of each.
11. Design a circuit to control a servo from an STK501 using a PWM signal or to communicate to a slave STK501 microprocessor via I²C.
12. Develop a program to either control a servo that is driving a sonar unit to find a bright light shining at it or to create an I²C slave out of a STK501.

**Lab Preparation**
The follow pre-lab preparation is required to be able to successfully complete this lab.
- Completion of “Introduction to the Atmel Atmega 128 controller.” (see our course website)
- Complete Reading the I²C handout.
- Read data sheet for Atmel’s Atmega 128 covering the setting of processor clock speed and calculating scaling bit for serial baud rates.
- Read the SRF08 Ultra sonic range finder Technical Specification.
- Read I²C Technical specification from Atmel’s data sheet on the ATMEGA 128.

**Components**

<table>
<thead>
<tr>
<th>Qty</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>STK500</td>
</tr>
<tr>
<td>1</td>
<td>STK501 with Atmega 128L</td>
</tr>
<tr>
<td>1</td>
<td>JTAG ICE</td>
</tr>
<tr>
<td>1</td>
<td>SRF08 sonic range finder</td>
</tr>
<tr>
<td>1</td>
<td>I²C -resistor jumper wires</td>
</tr>
<tr>
<td>8</td>
<td>Jumper wires</td>
</tr>
<tr>
<td>2</td>
<td>Serial port cable (straight through)</td>
</tr>
<tr>
<td>1</td>
<td>Servo motor with sonar adapter</td>
</tr>
<tr>
<td>1</td>
<td>Flashlight</td>
</tr>
</tbody>
</table>


Lab Overview

The I₂C lab experiment consists of a basic I₂C experiment that must be completed by everyone and two advanced experiments of which one must be completed.

These basic experiments require you to utilize the following programs on the PC; Tera Term, AVR Studio® and a C compiler to generate code for an Atmel chip.

Tera Term is a free software terminal emulator (communication program) for MS-Windows. It supports VT100 emulation, telnet connection, serial port connection, and so on.

AVR Studio 4 is the new professional Integrated Development Environment (IDE) for writing and debugging AVR applications in Windows 9x/NT/2000/XP environments. AVR Studio 4 includes an assembler and a simulator. The AVR Studio supports the following AVR development tools are also supported:

ICE50, ICE40, JTAGICE mkII, JTAGICE, ICE200, STK500/501/502 and AVRISP.

The basic experiment requires you to utilize the following hardware to complete the lab: STK500, STK501, and SRF08.

The Atmel AVR STK500 is a starter kit and development system for Atmel's AVR Flash microcontrollers. The STK500 gives designers a quick start to develop code on the AVR combined with features for using the starter kit to develop prototypes and test new designs. The STK500 interfaces with AVR Studio, Atmel's Integrated Development Environment (IDE) for code writing and debugging. The STK501 board is an expansion module designed to add 64-pin support to the Atmel STK500 Development Board. The STK500 and STK501 expansion module extends support to all current AVR devices. The STK501 includes connectors, jumpers and hardware allowing full utilization of the features on the ATmega64, ATmega128 and AT90CAN128. A Zero Insertion Force (ZIF) socket allows easy use of TQFP packages for prototyping.

The SRF08 is a smart sonar sensor that transmits an ultrasonic "ping" when instructed by your program and returns a signal when it receives an echo back. It provides a sensing range of the 1” to 18’ plus an on-board microcontroller which handles the critical timing functions and distance calculations. The SRF08 can report up to 17 echoes from each ping, allowing for its use in identifying the shape of a target or identify an open doorway. Echoes can be reported as inches, centimeters or time of reflection.

For the lab experiment you will have to set the clock and baud rate of the ATMEGA 128 and the corresponding baud rate of the Tera term terminal window. You will also have to setup the I₂C baud rate, if needed and all associated I₂C registers in the microcontroller. Then connect the SRF08 up to the ATMEGA 128. You will then have to ping the SRF08 and have the sonar readings sent to Tera term terminal window on the PC. Once this is completed successfully you can proceed to the advanced lab experiments. (see Figure 1)

---

**Figure 1** Laboratory setup for I₂C lab
This figure depicts the configuration of the serial cable between the computer running Teraterm and the STK 500/501 mounted with an ATMEGA 128l processor and the STK 500/501 the SRF04 and the SRF08 connection cable running I²C.

Introduction

In this lab you will investigate how to program microcontrollers to use their I²C interface and connect two microcontrollers together using an I²C bus.

The Inter-Integrated Circuit bus (I²C) is a patented interface developed by Philips Semiconductors. (In order for an IC manufacturer to implement the I²C bus in hardware, they must obtain licensing from Philips.) The I²C bus is a half-duplex, synchronous, multi-master bus requiring only two signal wires: data (SDA) and clock (SCL). These lines are pulled high via pull-up resistors and controlled by the hardware via open-drain drivers, giving a wired-AND interface. I²C uses an addressable communications protocol that allows the master to communicate with individual slaves using a 7-bit or 10-bit address. Each device has an address that is assigned by Philips to the manufacturer of the device. In addition, several special addresses exist, including a "general call" address (which addresses every device on the bus) and a high-speed initiation address. During communication with slave devices, the master generates all clock signals for both communications to and from the slave. Each communication begins with the master generating a start condition, an 8-bit data word, an acknowledge bit, followed by a stop condition or a repeated start (see Figure 2). Each data bit transition takes place while SCL is low, except for the start and stop conditions. The start condition is a high-to-low transition of the SDA line while the SCL line is high. A stop condition is a low-to-high transition of the SDA line while the SCL line is high. The acknowledge bit is generated by the receiver of the message by pulling the SDA line low while the master releases the line and allows the line to float high. If the master reads the acknowledgement bit as high, it should consider the last communication word not received and take appropriate action, including possibly resending the data.

![Figure 2  I²CData Transmission](image)

This figure shows a simple communication exchange using a start bit to initiate communications, the 8 bit message, the back and then the stop bit, if additional data was to be sent a restart bit could be sent instead of the stop.

I²C has a rather interesting feature called clock stretching, which is done when the slave device is unable to process the bit and wishes for more time. When this happens, the slave pulls the SCL line low. Since the signal behaves as a wired-AND, when the master releases the SCL line while the slave is "stretching" the clock, the master should notice that the line stays low. The I²C bus has three speeds: slow (less than 100Kbps), fast (400Kbps), and high-speed (3.4Mbps), each downward compatible. Philips has specified a recommended wiring arrangement should the signals need to leave the circuit board.

The I²C bus distances are often limited to on-board communications, although I have heard of developers using I²C successfully over distances of 50 feet! The true limit to I²C distances is the bit-rate and capacitance of the bus. As such, for off-board communications, I²C is practically limited to less than 10 feet for moderate speeds. (Patrick)

Summary of basic I²C specifications

- Maximum number of devices - 7
- Maximum Speed Greater than 3,400 (KBPS)
Maximum Distance 15 feet
- At Maximum Distance, Maximum speed is Less then 10 (KBPS)
- Maximum number of Pins 2

Lab Report Questions

While you do this lab, please consider the following questions that you will be required to answer in your lab report. Think about these questions as you perform each lab, noting there is no one correct answer for these questions so provide your reasoning and justify your answer for each.

1) What effect, if any, would a change in the pull up resistors value for the SDA and SCL have on the I²C bus?
2) Explain how you would integrate a memory chip with I²C input with one of two microcontrollers and describe how you would code and complete the circuit using your chosen components.
3) Should there be current (e.g., maximum amperage) specifications associated with these protocols?
4) Identify as many things about the circuitry that will limit the maximum rate at which data can be reliably transmitted, and explain why. (e.g., what effect might the length of the wires have? What effect might capacitance between data transmission and ground have?)
5) Is it possible that the power supply could have an effect, and if so, what?
6) Explain why you may use I²C, a serial protocol over SPI, another serial protocol.
7) What are the strong points of I²C and RS232 protocols compared with each other?

Procedure

1. Connect 12 VDC Power to the STK500. Verify 12 volts before plugging power into the STK500
2. Connect a serial cable from the communication port 1 of the computer to the RS232 CTRL port on the STK500 main board. This will be used to down load and debug code for this experiment.
3. Connect a second serial cable from the computers communication port 2 to the spare RS232 port on the STK500 main board.
4. Place jumper wires from port PE pins 0 & 1 to RX & TX on the STK500. Verify that the wiring is correct. It should be port PE pin 0 to Rx and port PE pin 1 to TX on the main board.
5. If instructed by TA you may need to compile all three programs used in this lab. If so you will need to open the folders and file specified by the TA. Compile the code using ICCAVR, as the code has not been check for compiling using GCC. The TA will provide compiler configuration setting.
6. If no code compiling is need the TA will provide location to folders where the *,HEX file can be found.
7. Open a HyperTerminal or Teraterm session. Configure it to Communication on port 2 setting set to baud rate to 9600, 8, none, 1, none.
8. Now disconnect the HyperTerminal session by clicking on the phone with the receiver off the hook, do not close the window.
9. Open AVR studio 4. Make sure the power is ON on the STK500 board.
10. Click the small icon in the AVR studio window that looks like a small IC with the letters AVR on it. A new window should pop up, if not see the TA.
11. Verify that the new window has the title “STK500 with STK501 top module.” If not, see TA.
12. Click on Program TAB and set device to ATmega 128. This is a pull down device menu. So find the ATmega128 in this list. The program mode is ISP. Verify that both are correct.
13. Click the board tab and verify the voltage on the board by clicking on read voltage button. Did you get the voltage reading back? If not, see TA.
14. Now click on the Read button under Oscillator and ISP clock. The two numbers should be STK500 = 3.686 Mhz and ISP Freq. = 460.8 Khz. If not, set these two numbers using the pull down menu and then press the write button. Then press read again to verify that the write took place and that the numbers are correct. If any problems with this, see TA.
15. Click on the advance tab and press read button. You should get a number like Ox?? Ox?? Ox?? Where the ?? are numerical values between 0 and 9. If you don’t see this, contact the TA.
16. Below the Ox?? Ox?? Ox?? There will be a message, verify that is states that this number matches the device selected. If it states that the device does not match return to step 12 and start over. If you are still having problems see the TA.
17. Click on the Lock bit tab and check that it is set correctly. The following bits should be checked; Mode 1…, Application Protection Mode 1 and Boot Loader Bit. If any are not checked, ask the TA for help. IF THESE ARE SET WRONG THE CHIP CAN’T BE USED AGAIN, LAB OVER.
18. Click on the fuse bit tab and verify the following items are checked; JTAG enabled, boot flash section size = 4096…, Brown out detection level at VCC = 2.7 B…, and Int RC Osc 4Mhz: start-up:6ck+64 ms… If these are not configured this way, see TA.
19. Press program location button, “…”, and locate the first program to down load. This should be the serial port 0 hex file. If you compiled this code from source look for the file compile name you used.
20. Press the program button and verify that the program loaded successfully. Do this by watching the execution display window at the bottom of the program window. You should only see OK! If there are any failures in this window, see the TA for help.
21. Congratulations, the device is now programmed.
22. Turn the STK500 off.
23. Bring the HyperTerminal window to the front of the monitor and press on the phone that has its receive down.
24. Look at the bottom of the terminal window and you should see the word connecter and a timer. If not, see the TA.
25. Turn the power ON and a message should appear in the HyperTerminal window.
26. If no message, see the TA. If you got message of congratulations, part one is done.
27. You have completed part one and are ready to move on.
28. Click the small icon in the AVR studio window that looks like a small IC with the letters AVR on it. You have done this in a previous step.
29. Verify that the new window has the title “STK500 with STK501 top module.”
30. Click on Program TAB and next to the device window that will be displaying ATmega 128 there is a Erase Device button. Click this button to erase the last program loaded into the ATmega128.
31. Turn the power off on the STK500
32. Take the Sonar adapter board and connect a 10-pin ribbon cable between it and the PORT D header on the STK500 board. The red marked wire should be on the end of the header with pins 0 and 1 on the STK 500 and on the end away from the RED and BLACK wires on the sonar board.
33. Turn the STK500 board on, this will power up the sonar board. You should see the red led light on the Sonar board and the red light on the STK500 both lit. If not, recheck connections and if the stk500 is not lit the power supply and it’s connection. If it’s still not on, contact TA.
34. The LEDs on both systems are on and the system is ready to load a new program.
35. Click the small icon in the AVR studio window that looks like a small IC with the letters AVR on it. A new window should pop up. If not, see the TA.
36. Verify that the new window has the title “STK500 with STK501 top module.” If not, see TA.
37. Download program “23C16” to the ATMEGA128, follow the directions above and verify all settings are correct before programming the ATMEGA128. **NOTE**, if you compile this code from source look for the file compile name you used.

38. Press STK500 reset button. After the board resets, a message should appear in the HyperTerminal window. If no message, recheck all settings and try again. If it still does not work, see the TA.

39. If you receive the message, then the I2C communication of the ATmega 128 is working.

40. You received the message, congratulations. You are done with part two, go on to part three.

41. Verify the LED on the sonar board is still on. If not, check power and connections.

42. Click the small icon in the AVR studio window that looks like a small IC with the letters AVR on it. A new window should pop up. If not, see the TA.

43. Verify that the new window has the title “STK500 with STK501 top module.” If not, see TA.

44. Download program “SRF08” to the ATMEGA128. Follow the directions above and verify all settings are correct before programming the ATMEGA128. **NOTE**, if you compile this code from source look for the file compile name you used.

45. Press STK500 reset button. After the board resets, a message should appear in the HyperTerminal window. If no message, recheck all settings and try again. If it still does not work, see the TA.

46. The message gives you the range the sonar is finding - first in inches, then in centimeters and then the gain value. The last information is the photo resistor value.

47. Put your hand over the sonar unit and move it away and towards the unit. The range values should change. If not, see TA.

48. Now cover and uncover the photo resistor and note the change in values.

**Appendix** (will be provided in the lab): **I²C EEPROM test program**

```c
/*
$Header$
*
/****************************************************************************/
/** @file main.c
* @author G. Hayward & L. Wayne Johnson
* @brief Test program to validate the I2C library code. Talks to an
*       Atmel AT24C16. The C16 version of the chip has no address line
*       so there can only be one on the bus at a time.
* Copyright (c) 2005, G. Hayward & L. Wayne Johnson. All rights reserved. */
/*****************************************************************************/
#endif

#include "i2c_lib.h"
#include "iom128v.h"
#include "stdio.h"

/** MOVE to a serial library.*/

/****************************************************************************
/** BAUD RATE NUMBERS FOR UBRR0   (I am not sure but I think that
/** the ICC library supports the
table diretll!)
/*****************************************************************************/
#define b9600 (26)   // 3.69MHz clock
#define b19200 (11)```

```
```c
#define  b38400 (5)
#define  b57600 (3)

//****************** Init USART Function************************************
void init_USART(unsigned int baud)
{
    UCSR0B = 0x00; //disable while setting baud rate
    UCSR0A = 0x00;

    //UCSR0C = 0x86;
    UCSR0C = 0x06;
    UBRR0L = (unsigned char)baud; //set baud rate lo
    UBRR0H = (unsigned char)(baud >> 8); //set baud rate hi
    UCSR0B = 0x98;
}

/**************************************************************************
* I2C / AT24Cxx test ******************************************************
**************************************************************************
*/

* The AT24Cxx has several addressing options
* depending on the size of the device. The 4 most
* significant bits of the address are always 1010
* (0xA Hex.) The next 3 significant bits are
* configurable as outlined below. The list significant
* bit is a read/write flag.
*
*  * 2K devices have 3 address pins A0, A1, A2
*  * that can be strapped to change the 3 least
*  * significant bits of the device address.
*  *
*  * 4K devices have 2 address pins available for
*  * strapping -- A1 and A2.
*  *
*  * 8K devices have 1 address pin available for
*  * strapping -- A2.
*  *
*  * 16K devices have NO address pins available for
*  * strapping. This limits the bus to a single device
*  * at one time.
*  *
* We are using the AT24C16 so there is only 1 possible device address./*

#define MEM_I2C_ADDR    (0xA0)

void main(void)
{
    unsigned char   MemVal;

    /** Initialize the baud rate so that we know the
    * printf baud rate.*/
    init_USART(b9600);

    printf("Welcome to the SRF 08 Test!\n");

    i2c_init();
    /** See if we can read a byte from the current address
    * counter on the memory. Normally a write is required to
    * set the address counter on the chip but for now we don't
    * care which address we read is.*/
```
I2C_START_RX(MEM_I2C_ADDR);
MemVal = i2c_receive(I2C_QUIT);
i2c_stop();

printf("We tried to read from memory : 0x%02X\n", MemVal);

while (1) {
    
}

i2c_lib.h program

/************************************************************************
Title : C include file for the I2C FUNCTIONS library (i2c.h)
Author: Chris efstathiou hendrix@otenet.gr
Date: 13/Jul/2002
Software: AVR-GCC with AVR-AS
Target: any AVR device
Comments: This software is FREE.
************************************************************************/
#endif I2C_H
#define I2C_H 1

/************************************************************************
** START OF CONFIGURATION BLOCK
************************************************************************/

#if define F_CPU
#define F_CPU 3686400L /* The cpu clock frequency in Hertz (used to calculate time)*/
#endif

#define I2C_SDA_PORT D /* The SDA port. Use capital letter (A,B,C,D... etc.) */
#define SDA_PIN 4 /* The SDA port pin */
#define I2C_SCL_PORT D /* The SCL port. Use capital letter (A,B,C,D... etc.) */
#define SCL_PIN 5 /* The SCL port pin */

/* The I2C_DELAY_TIME normally is 5 microseconds for a 100 Khz I2C bus ( (1/100000)/2 ).
   but due to bus capacitance and device response time in my application i use 50 microseconds.
   The I2C_TIMEOUT_TIME is set to whatever is the maximum time you think your device will take
   to perform the requested task. After that time the i2c_transmit function will return
   a "I2C_ERRORDEVICE_NOT_RESPONDING" or "I2C_ERRORDEVICE_BUSY" error code.
*/
#define I2C_DELAY_TIME 50 /* in microseconds (max over 1 second) */
#define I2C_TIMEOUT_TIME 1000 /* in microseconds (max over 1 second) */

/************************************************************************
** END OF CONFIGURATION BLOCK
************************************************************************/

/** Keyword definitions */

#define I2C_READ 1
#define I2C_WRITE 0
#define I2C_QUIT 0
#define I2C_CONTINUE 1
#define I2C_NO_ERROR 0
/* Function Declaration */

extern void i2c_init(void);
extern void i2c_start(void);
extern void i2c_stop(void);
extern unsigned char i2c_transmit(unsigned char data);
extern unsigned char i2c_receive(unsigned char ack);

/* Macro definitions */

#define I2C_ERRORDEVICE_BUSY 1
#define I2C_ERRORDEVICE_NOTRESPONDING 2

#define I2C_START(ADDRESS)     { i2c_start(); i2c_transmit(ADDRESS); }
#define I2C_START_TX(ADDRESS)  I2C_START(ADDRESS)
#define I2C_START_RX(ADDRESS)  I2C_START(ADDRESS | I2C_READ)

I2C main sonar program;

Main
#include "srf08.h"
#include <iom128v.h>
#include <stdio.h>

/* BAUD RATE NUMBERS FOR UBRR. These numbers were calculated using
 * table 74 in the ATmega 128 spec.
 */

typedef enum
{
  b9600  = 26,
}
b19200 = 11,
b38400 = 5,
b57600 = 3
} ATMEGA_128_BAUD_RATES;

/******************************************************************************
* Function     init_USART ******************************************************/
/******************************************************************************
* @brief       Intializes UART 0 using the specified baud rate.
* @param       baud    The baud rate to use when intializing UART 0 for
*                     stdin and stdout.
* @return      NONE
******************************************************************************/
void init_USART(unsigned int baud)
{
    UCSR0B = 0x00; //disable while setting baud rate
    UCSR0A = 0x00;
    //UCSR0C = 0x86;
    UCSR0C = 0x06;
    UBRR0L = (unsigned char)baud; //set baud rate lo
    UBRR0H = (unsigned char)(baud >> 8); //set baud rate hi
    UCSR0B = 0x98;
}
/* These variables are used in the delay loop. They
* are declared volatile so that the access to them
* are not optimized out by the compiler. This makes
* the delay more consistent across different compiler
* optimization settings.
***/
volatile long DelayLoopVar; /**< Counter of the delay loop. volatile so the
    /* compiler won't optimize it out.*/
volatile long A1; /**< Dummy variable used in the delay loop.*/
volatile long A2; /**< Dummy variable used in the delay loop.*/

/******************************************************************************
* Function     Delay                                                        */
/******************************************************************************
* @brief       This function delays for a specific amount of time. The the
*              exact delay has not be characterized and, for the current
*              use, does not need to be a specific length.
* @return      NONE
******************************************************************************/
void Delay(void)
{
    for (DelayLoopVar = 0; DelayLoopVar < 10000; DelayLoopVar ++)
    {
        A1 = A1 * A2;
    }
}
/** The number of Sonar samples to average together when
* calculation the distance.*/
#define AVG_FLT_SAMPLES (3)

/******************************************************************************
* Function     main *************************************************************/
/******************************************************************************
* @brief       The main entry point for our test application. This function
*              never returns.
*/
void main(void)
{
    unsigned int Range = 0;
    unsigned char Counter1;
    unsigned char Gain;

    init_USART(9600);
    /* This enables DOS EOL's, the default is use use \n only.
    * DOS EOL are \n, Unix \n, and MAC \r ...
    */
    {
        extern int _textmode; // this is defined in the library
        _textmode = 1;
    }
    printf("Welcome to the SRF 08 Test!\n");

    srf08_select_unit(SRF08_UNIT_0);

    printf("Initializing the SRF 08\n");
    srf08_init();
    /* Only the selected SRF08 unit will be initialized! */
    Gain=SRF08_MAX_GAIN;
    srf08_set_gain(Gain);
    srf08_set_range(SRF08_MAX_RANGE); /* Set range to 11008 mm */
    while (1)
    {
        /* AVERAGING FILTER */
        for (Counter1=0, Range=0; Counter1<AVG_FLT_SAMPLES; Counter1++)
        {
            Range+=srf08_ping(SRF08_CENTIMETERS);
        }
        Range /= AVG_FLT_SAMPLES;

        /* AUTOMATIC GAIN CONTROL */
        if (srf08_read_register(SRF08_ECHO_5)!=0)
        {
            if (Gain>=5)
            {
                srf08_set_gain(Gain-=5);
            }
            else
            {
                srf08_set_gain(Gain=0);
            }
        }
        else if (srf08_read_register(SRF08_ECHO_2)<=0 && Gain!=31)
        {
            srf08_set_gain(++Gain);
        }
        printf("RANGE (E1)  = %u\n", Range);
        printf("RANGE (E2)  = %u\n", srf08_read_register(SRF08_ECHO_2));
        printf("RANGE (E5)  = %u\n", srf08_read_register(SRF08_ECHO_5));
        printf("LIGHT sensor= %u\n", srf08_read_register(SRF08_LIGHT));
        Delay();
    }
    return;
}
Sonar I²C communication routines program:
/*
$Header$
*/
/****************************************************************************/
/** @file   srf08.c
*  @author G. Hayward & L. Wayne Johnson
*  @brief  SRF08 function library. This code is based on code written by
*          Chris efstathiou hendrix@otenet.gr, 13/Jul/2002. Most of the
*          modifications involved getting it to build with the ICC compiler.
* Copyright (c) 2005. All rights reserved. */
/*****************************************************************************/
#if 0
/*
$Log$
*/
#endif
#include "i2c.h"
#include "srf08.h"
/********************/
/* Global Variables */
/********************/
/** The I2C address to use when addressing the SRF 08.*/
static unsigned char address=SRF08_UNIT_0;

/********************/
/* Function     srf08_init */
/********************/
/**
* @brief       Initializes the SRF 08
*
* @return      NONE
*/
 /******************************************************************************/
void srf08_init(void)
{
  unsigned int status=0;
  i2c_init();
  I2C_START_TX(address);
  i2c_transmit(0);
  i2c_transmit(0x51);
  do
  {
    i2c_start();
    status=i2c_transmit(address);
    i2c_stop();
  }while (status != I2C_NO_ERROR);
  return;
}

/********************/
/* Function     srf08_set_gain */
/********************/
/**
* @brief       Sets the Gain on the SRF 08. TODO: What gain?
*
* @param       gain    The new value to use when setting the gain.
*
* @return      NONE
*/
 /******************************************************************************/
void srf08_set_gain(unsigned char gain)
{
  if (gain>31)
{  
gain=31;
}
I2C_START_TX(address);
i2c_transmit(1);
i2c_transmit(gain);
i2c_stop();

return;
}
/************************** Function     srf08_set_range ******/
/**
* @brief       TODO: What does set range mean?
* @param       millimeters
* @return      NONE
******************************************************************************/
void srf08_set_range(unsigned int millimeters)
{
  millimeters= (millimeters/43);
  if (millimeters > 0xff)
    
  millimeters=0xff;
  
I2C_START_TX(address);
i2c_transmit(2);
i2c_transmit(millimeters);
i2c_stop();

return;
}
/************************** Function     srf08_ping ************************/
/**
* @brief       Tells the sonar to send out a ping. The ping is used to
*              calculate the distance to the nearest object. The distance
*              is returned in the specified metric unit.
* @param       metric_unit
* @return      The distance to the nearest object calculated by the sonar
*              module.
******************************************************************************/
unsigned int srf08_ping(unsigned char metric_unit)
{
  union i2c_union
  
  unsigned int rx_word;
  unsigned char rx_byte[2];
  
i2c;

I2C_START_TX(address);
i2c_transmit(0);
i2c_transmit(metric_unit);

do
{
  i2c_start();
  i2c.rx_byte[0]=i2c_transmit(address);
i2c_stop();
}
while (i2c.rx_byte[0] != I2C_NO_ERROR);
I2C_START_TX(address);
i2c_transmit(SRF08_ECHO_1);
I2C_START_RX(address);
i2c.rx_byte[1]=i2c_receive(I2C_CONTINUE); /* get high byte msb first */
i2c.rx_byte[0]=i2c_receive(I2C.Quit);    /* get low byte msb first */
i2c.stop();
return(i2c.rx_word);
}

/******************************************************************************/
/**
* @brief       Reads an SRF 08 register.
* @param       srf08_register  The number of the register to read.
* @return      The value read from the specified register is returned.       */
/******************************************************************************/
unsigned int srf08_read_register(unsigned char srf08_register)
{
union i2c_union
{
    unsigned int  rx_word;
    unsigned char rx_byte[2];
} i2c;
I2C_START_TX(address);
i2c_transmit(srf08_register);
I2C_START_RX(address);
/* get high byte msb first */
if (srf08_register>=2)
{
    i2c.rx_byte[1]=i2c_receive(I2C_CONTINUE);
}
/* get low byte msb first */
i2c.rx_byte[0]=i2c_receive(I2C.Quit);
i2c_stop();
return(i2c.rx_word);
}

/******************************************************************************/
/**
* @brief       Changes the I2C address of the SRF 08. I2C bus supports
*              multiple devices with different I2C addresses. This command
*              changes the I2C address for the SRF 08 thus allowing multiple
*              SRF 08 on the same I2C bus.
* @param       new_i2c_address     The New address for the SRF 08
* @return      NONE                                                           */
/******************************************************************************/
void srf08_change_i2c_address(unsigned char new_i2c_address)
{
    /* Start the I2C address changing procedure */
    I2C_START_TX(address);
i2c_transmit(SRF08_COMMAND);
i2c_transmit(0XA0);
i2c_stop();

    I2C_START_TX(address);
i2c_transmit(SRF08_COMMAND);
i2c_transmit(0XAA);
i2c_stop();
    I2C_START_TX(address);
void srf08_select_unit(unsigned char srf08_address) {
    /* New address validity check */
    if ( (srf08_address<0xE0 || srf08_address>0XFE) && srf08_address != 0 ) {
        return;
    }
    if (srf08_address%2) {
        return;
    }
    /* Make the new i2c address the active one. */
    address=srf08_address;
    return;
}

References