The George Washington University
School of Engineering and Applied Science
Department of Computer Science

CSCI 339 Final Project
Final Report
Wireless Temperature Sensor

By Keevan Simmonds
kwsimmonds@hotmail.com
Arpil 25th, 2006

Project Abstract

This project is a wireless 1-Wire control network. It is specifically a wireless temperature sensor. Two Zilog Z8F6403 microcontrollers are used in a master/slave fashion. The master is the controller that is connected to a PC. It links to the slave via a Linx TR-916-SC-PA wireless transceiver module. The slave microcontroller is connected to a 1-Wire network that incorporates a switch and a temperature sensor. The slave will toggle the switch and/or get the current temperature on command from the master. The result of the toggle and/or the current temperature is sent back to the master via the wireless transceiver and displayed on the PC terminal.

Status

A wireless 1-Wire weather station was originally proposed. However, the scope of this project was not fully understood at the time of the proposal. In order to communicate with the weather station over the 1-Wire bus, an elaborate search algorithm must be implemented in the 1-Wire master software in order to get the wind direction. Thus, the project scope has changed to become a remote, wireless 1-Wire network controller. All 1-Wire commands except for the search algorithm were implemented and work as planned.

The wireless transceivers have been implemented in the project. However, getting them to operate as a true RS-232 wireless replacement was non-trivial. Since the transceivers are half-duplex, the data direction must be controlled by the Z8. Changing the data direction adds a lot of noise on the receiver, some of which can be interpreted as valid data. In addition, some noise is present when in normal receive operation. This was overcome by implementing a strict messaging protocol and sending each message multiple times to ensure reception.

Specification

The project uses two Zilog Z8 microcontrollers as the main processing elements in the master and slave modules. In addition each module contains a Linx 916MHz SC-PA wireless transceiver. The slave module also contains a I^2C to 1-Wire master chip and two 1-Wire slave devices. Table 1 is a listing of all hardware components in the design.

Vendor	Part Number	Quantity	Description
Zilog	Z8F6403	2	8-bit microcontroller
Linx	TR-916-SC-PA	2	Wireless transceiver
Maxim	DS2406	1	1-Wire switch
Maxim	DS1822	1	1-Wire Digital Thermometer
Maxim	DS2482	1	I^2C to 1-Wire bridge

Table 1: Hardware Listing

The master module consists of a Z8 microcontroller and a wireless transceiver. Its job is to send 1-Wire network commands to the slave module and display the results on a

terminal. In the current setup, the master will command the slave module to toggle the DS2406 switch when switch 1 is pressed on the Z8 evaluation board and will request a temperature reading from the DS1822 when switch 2 is pressed. It will report the results of each command to a terminal emulator via UART 0. The connection is at 9600 baud, 1 stop bit, no parity, and no flow control.

The master connection to the Linx transceiver is through UART 1. The baud rate is 9600 baud which is well under the 33600 baud limit of the Linx transceiver. No parity or flow control is implemented. The transceiver is connected directly to the Txd and Rxd pins on the micro-controller, bypassing the level shifters that are not needed for the implementation. The wireless transceiver is half-duplex, thus the transceiver must be configured as either a transmitter or receiver at any point in time. The transceiver is configured by enabling either the transmit enable or receive enable pin (but no both simultaneously) depending on the desired functionality. This is done by using two GPIO pins from the Z8. and connecting them to the TxEn and RxEn pins of the Linx transceiver. PE5 and PE7 were chosen for this task. PE5 is the transmit enable pin and PE7 is the receive enable pin. It is the responsibility of the Z8 software to ensure that PE5 and PE7 are not enabled at the same time.

The slave Z8 is connected to the Linx transceiver in the same fashion as the master with the exception that it is connected to UART 0 instead of UART 1. The slave Z8 is also connected to a DS2482 I^2C to 1-Wire bridge. This allows the Z8 to act as a 1-Wire master. Communication with the Z8 DS1822 is at 100 kHz or standard speed. Two slaves are present in the 1-Wire network. A DS2406 single switch is connected to a LED. Note that the data sheet refers to the DS2406 as a dual switch. However, the three pin version of the DS2406 is only a single switch. When commanded by the master, the slave Z8 commands the DS2406 to toggle the switch, turning the LED on and off. A DS1822 thermometer is connected to the 1-Wire network as well. When the temperature command is sent to the slave from the master, the slave Z8 requests a temperature update from the DS1822 and sends it to the master.

While there are two distinct software programs in each module, they share many common libraries. The software description will first describe the software in both modules at a high level, go into detail on the common modules, and then detail the master and slave specific software.

The function of the master Z8 is to simply request the slave to perform an action, receive the result of the action, and display the result on a terminal. The master uses a set of generic UART drivers for both communication with the terminal and the slave. Since the Linx transceiver is essentially n wireless RS-232 replacement, the UART drivers are used to communicate with the slave. They are used by Linx drivers which handle the half-duplex feature of the Linx transceiver. The master also uses a timer to poll the each switch, perform debouncing, and perform the desired action when a switch is pressed. The timer also updates the display. A display library is also used which will up the LED display. Since the display is not used in the project, all this does I turn all of the LEDs

off. The master also uses a .h file that declares a set of structures that is to be used as the message protocol between the master and the slave.

The slave module uses the UART and Linx916 drivers to communicate with the master. It also uses the led drivers to clear the display. The slave module also has drivers to interface with the DS2482 via the I^2C port. In addition, DS2406 and DS1822 specific drivers are used in the slave module.

The UART drivers are implemented in uart.c and uart.h. They provide full transmit and receive capabilities for UARTs 0 and 1. There are putch functions to write a single character to the UART buffer and a print function to write a string of characters to the UART buffer. Reception of characters is accomplished by enabling the receive interrupt and placing the received character in a 64 byte circular buffer. The buffer can be read by calling getch or getbuf. The getch function retrieves the next character in the buffer and the getbuf function copies the entire buffer into a user buffer. In order to use the functions mentioned, the user must initialize the uart drivers by calling init_uart for the desired UART.

The Linx916 drivers are found in Linx916.c and Linx916.h. These drivers handle the fact that the Linx transceivers are not full duplex as a wired UART is. Initialize the drivers by calling linx916_init with the desired UART (0 or 1) as the argument. This will initialize either the UART 0 or 1 drivers. All function calls after init will be to the desired UART. The transceiver must be set either to transmit or receive mode. This is accomplished by calling linx916_enableTX or linx916_enableRX. These functions ensure that TxEn or RxEn are enabled mutually exclusively. In addition, the UART interrupts are disabled when in transmit mode. There is also functionality to call getch, getbuf, putch, and print by calling the linx916 equivalent. Note that it is up to the user to ensure that the transceiver is in the proper mode before calling a get or put function. There is also a function that will generate a 10ms delay. This is because after enabling a specific mode (Tx or Rx), there is a settle time of around 10ms. Thus the function delay10ms will set a 10ms one shot timer that sets a global variable. The function waits for the variable to be set and then returns.

The led drivers are in LED_Characters.c/.h and ledlib.c/.h. These libraries have quite a bit of functionality developed for other applications. In this application, they are only used to initialize the drivers by calling init_led_gpio and to turn off all of the LEDs by calling turn_off_led.

The file MasterSlaveStructs.h contains a set of structures that define the message protocol between the master and the slave. Each message starts with a header which contains the message type, message size, and number of times the message has been sent. The type is used to identify the message. The size tells the receiver how may bytes to expect and the number of times is used since the wireless link can be unreliable and thus a message may need to be send multiple times. If the receiver receives the first transmission and the subsequent transmission, the receiver can tell that the subsequent transmission is a copy because it has a greater transmission number. The valid message types are:

Message Type	Sent By	Description
DATA_REQ_MSG	Master	Request temperature from slave
TEMP_MSG	Slave	Current temperature from DS1822
ARM_MSG	Master	Toggles (dis/arms) DS2406
ARM_ACK_MSG	Slave	Acknowledges successful DS2406 toggle

Table 2: Master/Slave Message Set

Messages are sent and received by calling the functions getNextMsg and sendMsg in the file linx916_msgHandler.h/.c. The getNextMsg function continually reads the receiver uart buffer by calling linx916_getch. It places the last character at the end of a buffer. Since the Linx transceivers are very noisy and many characters in the UART buffer are due to this noise, the function must always check to ensure the character is valid. This is accomplished by shifting the buffer every time a character is received. The character in position 0 is shifted out of the buffer; the character in position 1 is shifted to position 0 so on and so forth. The new buffer is typecast as a header structure. The header is checked to see if it is a valid header, by calling the function check header. This function checks to see if the message type is valid and if so if the message size matches the message type. If so, getNextMsg will read the rest of the message and return. If not, it will shift the buffer and try again on the next message reception. The sendMsg function sends the message three times to ensure reception. The number three was determined through experimentation. This number gives a high probability that the receiver will receive the message.

The slave module incorporates drivers for the I^2C port. The drivers for the DS2482 use these drivers to communicate with the DS2482 via the I^2C port. The I^2C drivers are implemented provide facilities to initialize the I^2C hardware, send and receive a byte, send start and stop bits, wait for transmit data register empty and receive data register full, wait for an n/ack, and explicitly send a nack to a slave. The drivers interface directly with the I^2C control, status, and data registers. While the drivers themselves are fairly straightforward, how to use them is a bit non-standard. This is due to some anomalies in the Z8 implementation. For example, Figure 1 shows the I^2C specification to write a byte(s) to a slave. The sequence begins when the master issues a start (S). It then issues the 7-bit slave address and a one bit write bit 0. The master should receive and acknowledge from the slave for the address and every data byte thereafter. When the master is finished, it sends a stop bit (P) to terminate the sequence.

Figure 1: I^2C Write Sequence Specification

It is logical to assume that the Z8 interface would work similarly to this. There are facilities to send start and stop, transmit and receive data registers, and n/ack status in the status register. However, the Z8 does not return the ack status until after stop has been sent. Thus to write a byte, the following sequence must be used:

1. Send start
2. Send Slave address and write bit
3. Wait for transmit data register to empty
4. Write data byte
5. Wait for transmit data register to empty
6. Send Stop
7. Wait for ack
8. If no ack, go to 1.

The transmit data register logs whether or not the data register is full (i.e. byte has not yet been sent). If another byte is written to the transmit data register while it is full, it will not be sent. A read from the slave is not intuitive either. Figure 6 (reference 4), shows the sequence for a read from the slave. It begins like write with a start bit and a slave address. The last bit of the address field is 1 to command the slave to send data to the master. The master receives this and sends an ack plus any data. The master terminates with a nack and stop to tell the slave to stop transmitting.

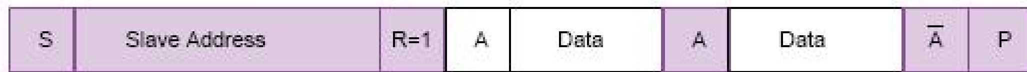

Figure 2: I²C Read Specification

Once again, the master should not look for an ack from the slave until after it has sent stop. In addition, the read byte routine sends nack immediately after the slave address has been sent to command the slave to send one byte only. The sequence is:

1. Send slave address and read bit
2. Send nack
3. Send start
4. Wait for receive data register to fill
5. read byte
6. Send stop

The Z8 interfaces with the DS2482 via the I^2C port using the drivers specified above. In addition, drivers have been written for the DS2406 and DS1822 that use the DS2482 drivers. The DS2482 drivers contain functions to reset the DS2482, read status, write configuration, reset the 1-wire bus, write a byte to the 1-wire bus, read a byte from the 1-wire bus, and send a match ROM command.function.

The DS2406 is a switch and EPROM. The EPROM functionality has not been utilized on this project. Thus, the drivers for this device only handle toggling of the switch and reading the device status. The switch is opened or closed by writing to the only byte of SRAM on the device. The SRAM is mapped in EPROM as address 0x0007 and is the last byte in the DS2406 status memory map. The memory map is shown in Figure 3. If a 1 is written to the PIO-A flip-flop, the PIO-A pin will float. If a zero is written to is, PIO-A will be connected to ground.

DS2406 STATUS MEMORY MAP

ADDRESS	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0 (EPROM)	BM3	BM2	BM1	BM0	WP3	WP2	WP1	WP0
1 (EPROM)	1	1	1	1	1	1	Redir. 0	Redir. 0
2 (EPROM)	1	1	1	1	1	1	Redir. 1	Redir. 1
3 (EPROM)	1	1	1	1	1	1	Redir 2	Redir 2
4 (EPROM)	1	1	1	1	1	1	Redir 3	Redir 3
5 (EPROM)	EPROM Factory Test byte							
6 (EPROM)	Don't care, always reads 00							
7 (SRAM)	Supply Indication (read only)	PIO-B Channel Flip-flop	PIO-A Channel Flip-flop	CSS4 Channel Select	CSS3 Channel Select	CSS2 Source Select	CSS1 Source Select	CSS0 Polarity

Figure 3: DS2406 Status Memory

The procedure for setting/clearing PIO-A is:

1. Write skipROM/match ROM (follwed by serial number) to the 1-wire bus
2. Write the DS2406 Write Status (0x55) to the DS2406.
3. Write the lsb of the status register (0x07) followed by the msb (0x00)
4. Write the desired value of the status register with PIO-A set or cleared depending on the desired state.
5. Write 0xFF to the DS2406. This is a dummy byte that transfers the status from the scratchpad to memory and implements the desired action.

The DS1822 is a digital thermometer that will output its temperature onto the 1-Wire bus. The drivers for this device will start a temperature conversion, wait for a temperature conversion to complete, and read the result last temperature conversion. The start conversion utility commands the DS1822 to start a temperature reading. This is done by simply issuing a skip/match ROM command followed by the start conversion command (0x44). The default resolution is 12-bits (the maximum). This is not changed in this application so all readings are 12-bits. As mentioned in the hardware section, while the DS1822 is taking a reading and transferring it into memory (can take up to 750ms for a 12-bit conversion). While the conversion is in progress, the DS1822 will pull the 1-Wire bus down. The wait for temperature conversion routine simply reads the status of the bus and returns when the bus is high. Once the conversion is complete, the master can read the result of the last conversion from DS1822 memory. This is done by:

1. Send a match/skip ROM command to DS1822
2. Send a read scratchpad command (0xbe)
3. Read two bytes, the lower 8-bits and upper 4-bits of the temperature respectively
4. Reset the 1-wire bus
5. Combine the two temperature bytes into a 12-bit word.
6. Multiply the result by 125/2000 to get the temperature in Celsius.
7. Perform a conversion into Fahrenheit, if desired: $T_f = (9/5) * T_C + 32$.

The master application software is in main.c. It initializes uart 0, timer 2, and the Linx transceiver drivers on UART 1. The Linx transceiver is defaulted to receive mode. The application is an infinite while loop that is interrupted every 3ms by timer 2 isr. This isr

is the switch monitor function. If the monitor determines that switch 1 or 2 has been pressed, it sets global variable armNetwork or getTemp respectively. If armNetwork is set, the main function sends a ARM_MSG message to the slave to command it to toggle the DS2406. It then waits for an ARM_ACK message from the slave and prints the result to the terminal. If getTemp is set, the main function sends a DATA_REQ_MSG to the slave requesting a temperature update from the DS1822. It then waits for a TEMP_MSG from the slave and prints the updated temperature to the terminal.

The slave application is also in main.c. The application starts by initializing the I^2C drivers, the linx916 drivers on UART0, resetting the DS2482 and the 1-Wire bus, and ensuring the DS2406 is switched off. The application then loops infinitely calling getNextMsg. If a message is received, the application will act depending on the message received. If the message is an ARM_MSG, the slave will enable the switch if it is disabled or disable the switch if it is enabled. It then sends back an ARM_ACK message to the master. If the message is a DATA_REQ_MSG, the slave will start a DS1822 temperature conversion, wait for the conversion to complete, and read the temperature from the DS1822. It then takes the 12-bit temperature and converts it to degrees Fahrenheit by the equation:

Temp C = temp12bit * 125.0/2000.0
Temp F = (9/5) * TempC + 32.

Note that the floating point library must be enabled in the project settings to perform this conversion. The slave now rounds the temperature in Fahrenheit to the nearest integer and sends it to the master in a TEMP_MSG.

Implementation and Consturction

A hardware block diagram is shown in Figure 4. The master is connected to a terminal and to the slave via the Linx SC-PA transceiver. The slave is connected to the master via a Linx transceiver and to the 1-Wire devices (DS2406 and DS1822) via the DS2482 I^2C to 1-Wire bridge.

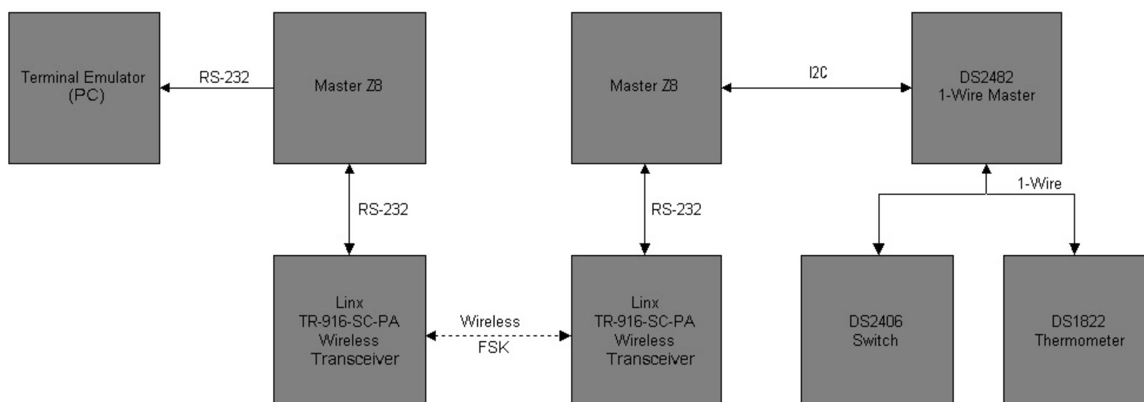

Figure 4: Hardware BD

A schematic of the master is shown in Figure 5. This shows that the terminal is connected to the console connector of the Z8 master. The Linx916 wireless transceiver is connected directly to Tx1 and Rx1, bypassing the level shifters. PE7 and PE5 are connected to TxEn and RxEn respectively.

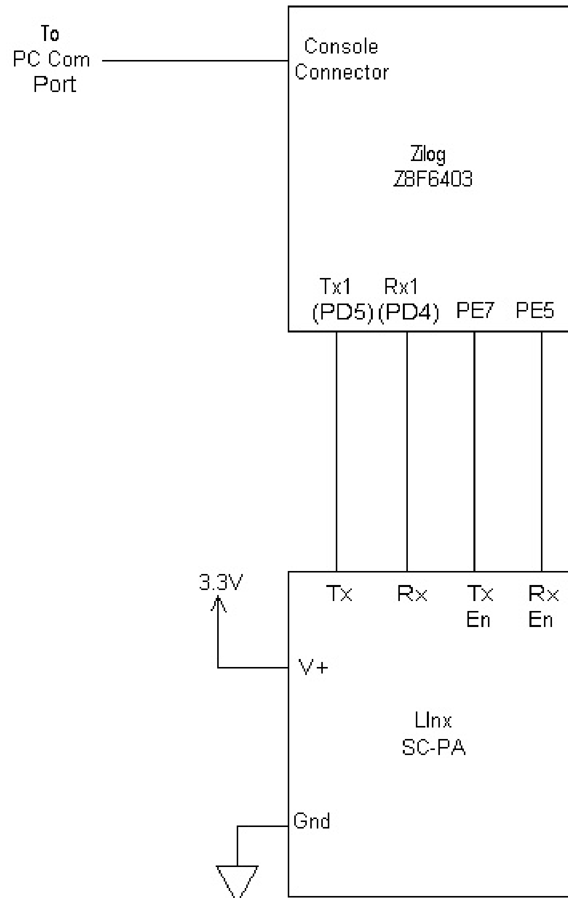

Figure 5: Master Schematic

A schematic of the slave module is shown in Figure 6. The Linx916 transceiver is connected to the Z8 via UART0. These are direct connections to the microprocessor pins, bypassing the level shifters. The Z8 is connected to the DS2482 via the I^2C pins SCL and SDA. The DS2482 connects to the 1-Wire network via the IO pin. This is terminated by a 100 ohm resistor to prevent bus signal reflections and pulled up to 5V by a 4.4k resistor. The PCLTZ pin in not connected since a strong pull-up is not required in this 1-Wire network. The 1-Wire data line is connected to the dat pins on the DS2406 and DS1822. The PIOA pin on the DS2406 is the switch pin. When enabled, the pin is connected to ground. When disabled, it floats. An LED is connected between 5V and PIOA. When the switch is enabled, the LED will light up. The DS1822 is powered from Vcc, not parasitically. This is because if it were powered parasitically, a strong pull-up would be required when the DS1822 was performing a temperature conversion. In addition, the DS1822 could not pull the bus low while converting temperature since the master would be pulling the data line up strongly.

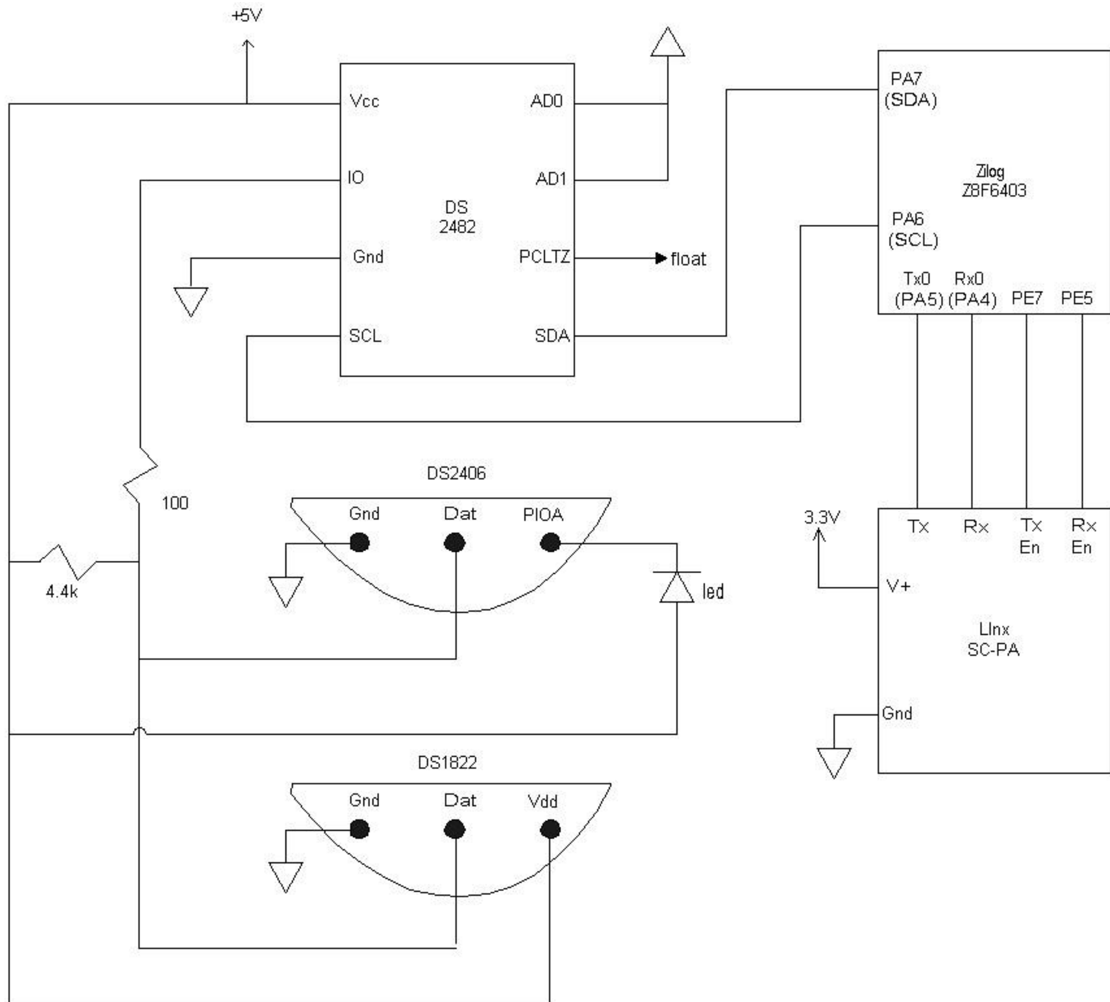

Figure 6: Slave Schematic

A block diagram of the master Z8 software is shown in Figure 7. The master continually polls to see if Sw1 or Sw1 has been pressed. If Sw1 is pressed, it sends an ARM_MSG to the slave which requests the slave to toggle the switch. It then waits for a ARM_ACK from the slave and reports the result on the terminal. If Sw2 is pressed, it sends a DATA_REQ_MSG to the slave which requests the current temperature. It waits for the temperature response from the slave and sends it to the terminal.

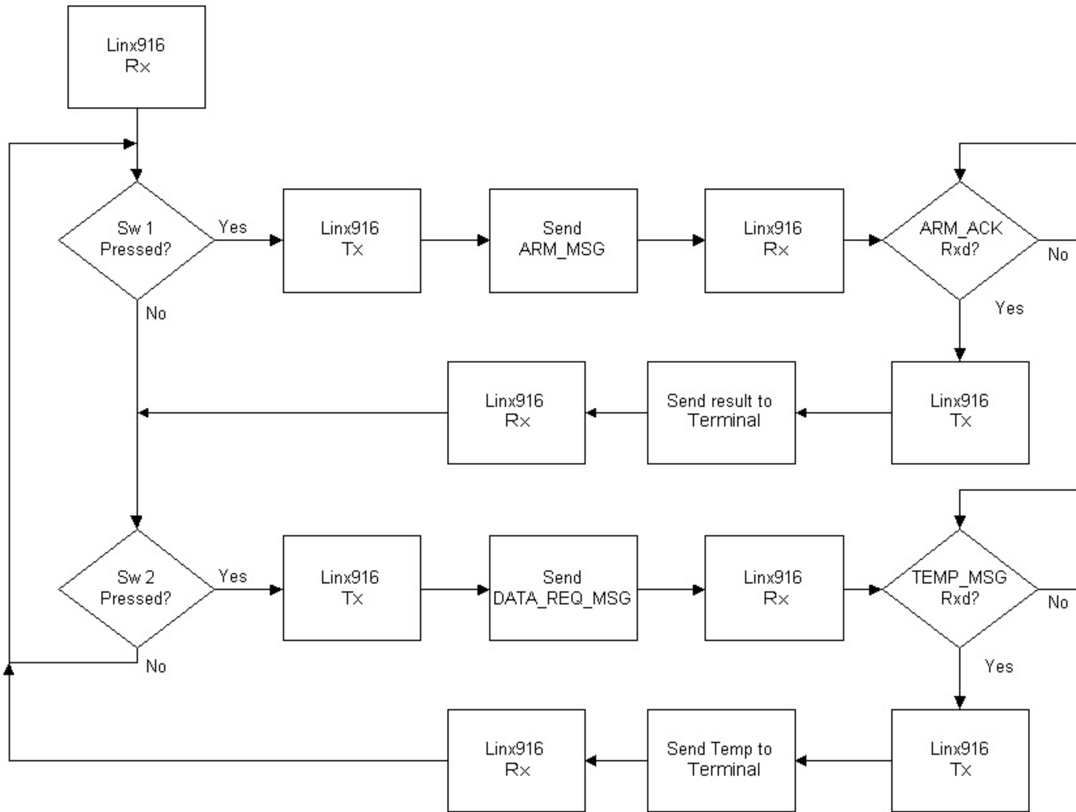

Figure 7: Master Software Block Diagram

Figure 8 is a block diagram of the slave Z8 software. The slave waits for a message from the master. If an ARM_MSG is received, it either enables or disables the DS2406 and sends and ARM_ACK back to the master. If a DATA_REQ_MSG is received, the slave requests a DS1822 temperature conversion, waits for the conversion to complete, requests the temperature from the DS1822, converts the temperature to Fahrenheit, and sends a TEMP_MSG to the master which contains the temperature.

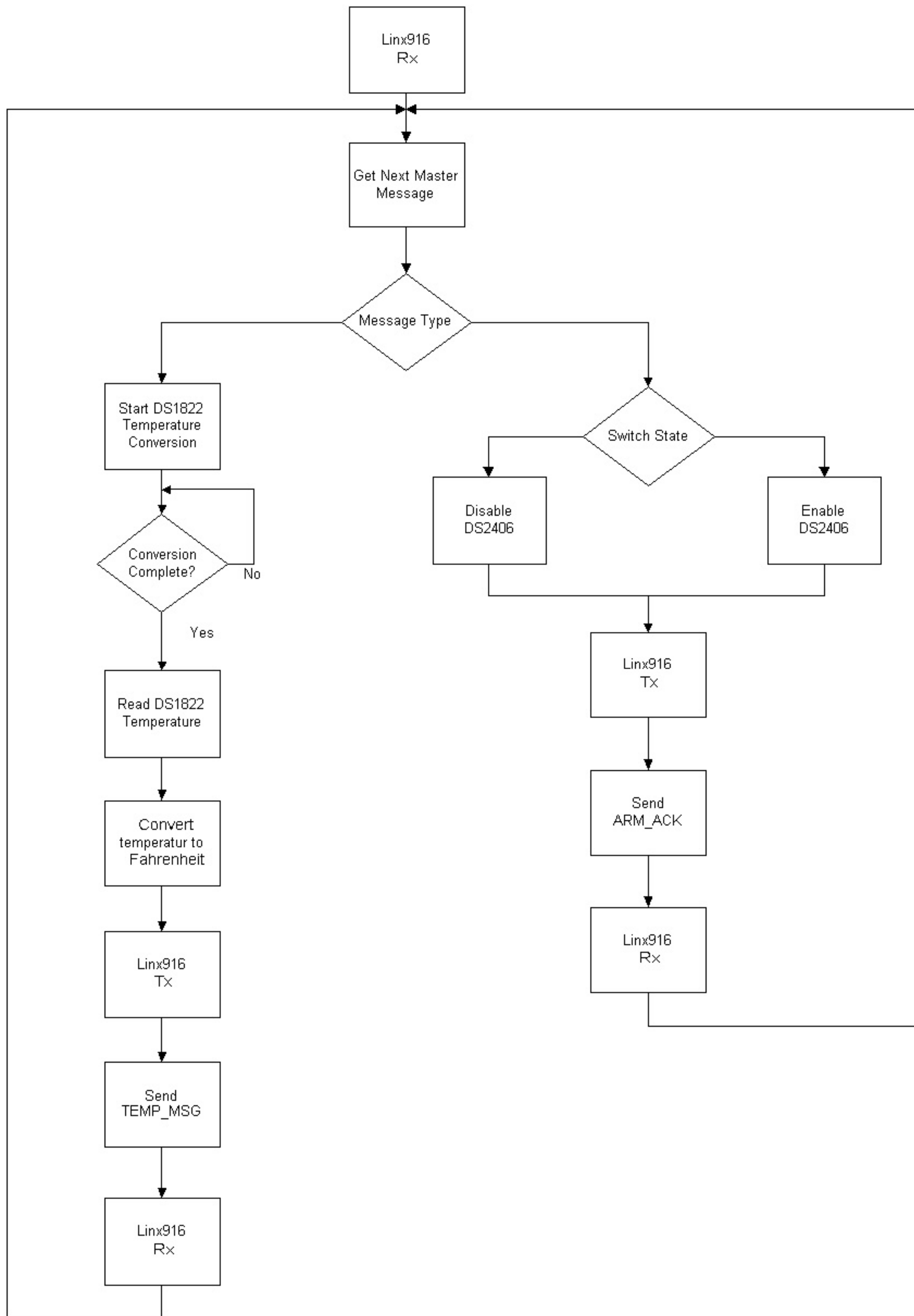

Figure 8: Slave Software Block Diagram

Figure 9 is a picture of the slave module. This picture was taken for the presentation. Unfortunately, I turned most of the hardware for this project to the professor before I wrote the paper and did not get a chance to take a picture of the entire project.

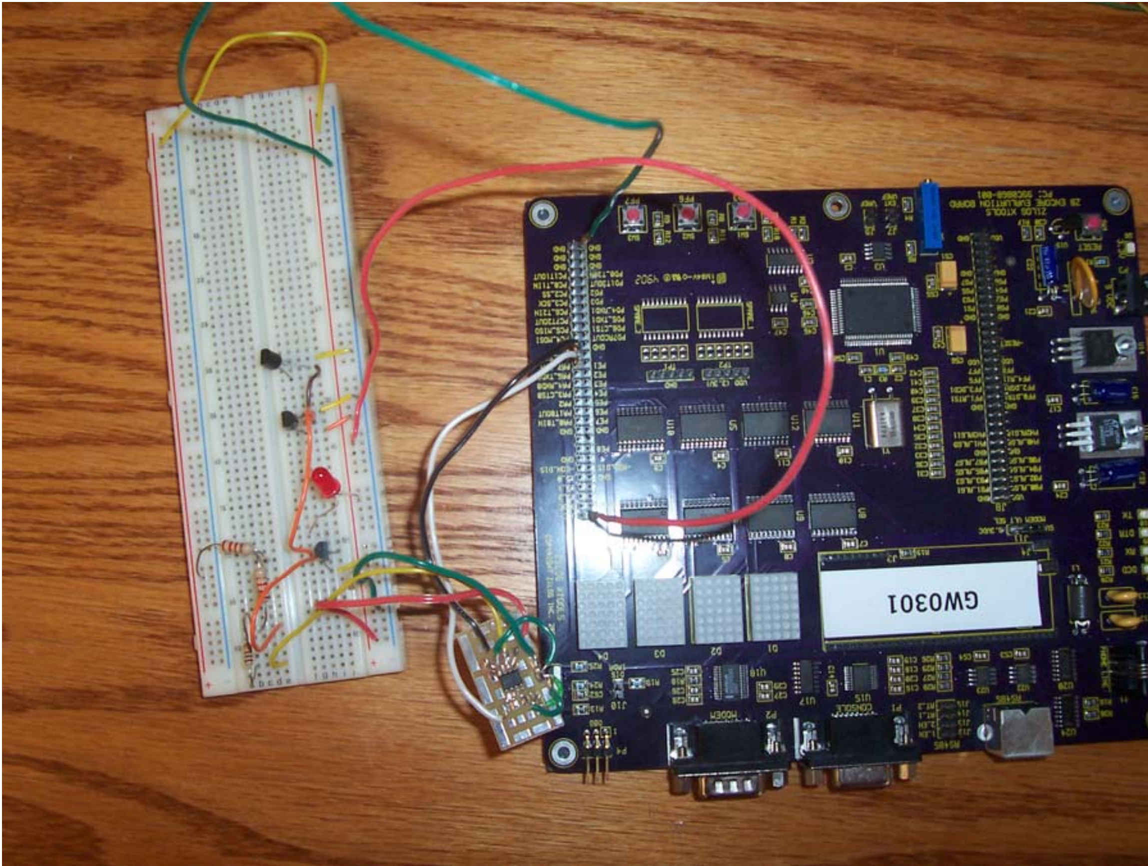

Figure 9: Slave Module

Retrospective

The most important design decision was whether or not to implement the actual weather station into the project or not. This was part of the original proposal. However, for personal reasons, I had to present my project a week earlier than the rest of the class. Most of the slave module, except for the wireless transceiver part was already implemented as part of the presentation. I really only had time to implement either the weather station, which requires an elaborate 1-Wire search ROM algorithm or the wireless transceiver. I felt that implementing the wireless feature was a better option since it makes the project more functional.

I learned quite a bit from this project. I had never used I^2C or 1-Wire busses before I did this project. I now have a very good understanding of both of these protocols. Implementing the wireless transceiver was also a very good learning experience. I was not prepared for the noise I would see on the wireless receiver. This may have been averted by actually using a circuit board with a ground plane layer on it instead of just soldering the transceiver to a PC board. I thought about using aluminum foil on the bottom of the PC board, but I felt that I may just as easily short something out doing that.

Attachments

The source code for both the master and slave modules are provided in the same folder as this report. They are in the folder named source. There are three folders, one for the master application, on for the slave application and one for files shared by both in the Common Files folder. Datasheets for the DS2482, DS2406, DS1822, and the Linx SC-PA are provided in Adobe Acrobat format. The hardware block diagram, master and slave schematics, and the master and slave software block diagrams are provided in Visio and JPEG format. The project summary is also included in Word format.