California State University, Fresno

Lyles College of Engineering

Electrical and Computer Engineering Department

FINAL PROJECT REPORT

Project Title: Wireless Greenhouse Monitoring Embedded System

Course Title: ECE 178

Semester: Fall 2021

Date Submitted: December 7, 2021

Prepared By:	Sections Written:
Daniel Adam	**Daniel Adam**
Zachary Curtis	**Zachary Curtis**

INSTRUCTOR SECTION

Comments: _____

_____

_____

_____

Final Grade:

Team Member 1: _____

Team Member 2: _____

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Problem Statement

Plants are very sensitive to their environmental conditions. Different plants thrive in different climates, and for farmers or even hobbyists, having a greenhouse to grow plants can be extremely beneficial, as it can allow for the climate of the plants to be very controlled. However, there is still the issue that monitoring the greenhouse conditions can be difficult and may often require someone to enter the greenhouse, which can disturb the conditions for smaller greenhouses and become inconvenient if needed multiple times a day. With only checking the conditions every so often, there will also be no way of constantly knowing what the current conditions are. This project seeks to solve this problem by creating an embedded monitoring system that can wirelessly communicate with another embedded system.

1.2 Statement of Objectives

In this project, the main objective will be to create a system consisting of two devices that will allow the user to monitor the temperature, humidity, and pressure of the greenhouse remotely and at all times of the day. The first of these two devices will collect the data using a sensor and wirelessly transmit it to the second of these two devices, which will receive said data and then store it on an SD card as well as display the data. Controls will be available to view not only the temperature, humidity, and pressure of the most recent time where it was collected, but also at past points in time which had also been received and stored on the SD card. Both of these devices will be modeled on an FPGA with a few external components being used, and they will both be completely interrupt-driven.

2. THEORETICAL BACKGROUND

The transceiver used for both devices in the system created in this project was the nRF24L01+ 2.4 GHz wireless transceiver, which is a radio frequency transceiver that is controlled through an SPI interface. The SPI interface consists of four pins: chip select (CSN), clock (SCK), master out slave in (MOSI), and master in slave out (MISO), which make up four of the six pins used to control this device (with the other two being the chip enable (CE) and interrupt request (IRQ) pins, which will both be touched on later). On the master device, in this case the FPGA, the CSN, SCK, and MOSI connections were all outputs, and the MISO connection was the only input. Conversely, for the slave device, in this case the transceiver, the CSN, SCK, and MOSI connections were all inputs, with the MISO connection being the only output. As per the specifications of the transceiver, a clock polarity of 0 and a clock phase of 0 were used. This meant that while idle, the clock was low, and that the data was interpreted on the rising edge of the clock and then changed on the falling edge. Communication between the devices was byte by byte and would last as long as the CSN input was held at 0. Depending on the first input byte given on the MOSI line, a specific instruction could be executed, such as reading a register, writing a register, reading the RX FIFO (where incoming data is temporarily stored), writing to the TX FIFO (where data to be transferred is temporarily written to), and more. With every command sent, the first byte of the data receiver would always be the contents of the status

register. Examples of timing diagrams for a read and then a write command are shown below in Figures 2.1 and 2.2 respectively, with a key to the abbreviations used in Table 2.1.

Table 2.1: Abbreviation key for timing diagrams for nRF24L01+ transceiver [1]

Abbreviation	Description
Cn	SPI command bit
Sn	STATUS register bit
Dn	Data Bit (**Note:** LSByte to MSByte, MSBit in each byte first)

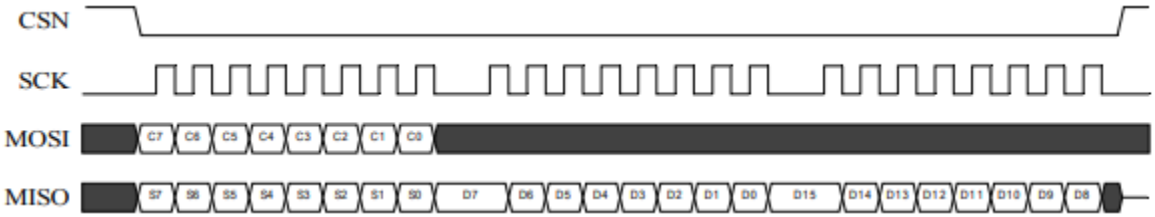

Figure 2.1: Timing diagram for SPI read command to nRF24L01+ transceiver [1]

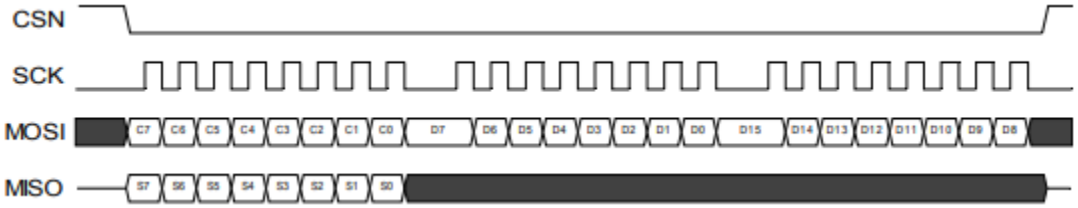

Figure 2.2: Timing diagram for SPI write command to nRF24L01+ transceiver [1]

Show below in Table 2.2 is not a complete list of the commands available through this SPI communication, but a list of the ones which proved most useful in this project. The full list is about twice as long and did not involve commands which were vital to the creation of this project.

Table 2.2: List of some of the available commands through SPI communication [1]

Command name	Command word (binary)	# Data bytes	Operation
R_REGISTER	000A AAAA	1 to 5 LSByte first	Read command and status registers. AAAAA = 5 bit Register Map Address
W_REGISTER	001A AAAA	1 to 5 LSByte first	Write command and status registers. AAAAA = 5 bit Register Map Address Executable in power down or standby modes only.
R_RX_PAYLOAD	0110 0001	1 to 32 LSByte first	Read RX-payload: 1 – 32 bytes. A read operation always starts at byte 0. Payload is deleted from FIFO after it is read. Used in RX mode.
W_TX_PAYLOAD	1010 0000	1 to 32 LSByte first	Write TX-payload: 1 – 32 bytes. A write operation always starts at byte 0 used in TX payload.
FLUSH_TX	1110 0001	0	Flush TX FIFO, used in TX mode
FLUSH_RX	1110 0010	0	Flush RX FIFO, used in RX mode Should not be executed during transmission of acknowledge, that is, acknowledge package will not be completed.

Shown below in Table 2.3 are two of the registers accessible in the nRF24L01+ transceiver. These registers would be read or written to using the SPI commands R_REGISER and W_REGISTER, and the CONFIG and STATUS registers shown below proved to be the most useful in this project. There were many other registers available as well, but the full list would span many pages, and many were left to their default value on reset.

Table 2.3: List of two of the available registers in the nRF24L01+ transceiver [1]

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
00	CONFIG				Configuration Register
	Reserved	7	0	R/W	Only '0' allowed
	MASK_RX_DR	6	0	R/W	Mask interrupt caused by RX_DR 1: Interrupt not reflected on the **IRQ** pin 0: Reflect RX_DR as active low interrupt on the **IRQ** pin
	MASK_TX_DS	5	0	R/W	Mask interrupt caused by TX_DS 1: Interrupt not reflected on the **IRQ** pin 0: Reflect TX_DS as active low interrupt on the **IRQ** pin
	MASK_MAX_RT	4	0	R/W	Mask interrupt caused by MAX_RT 1: Interrupt not reflected on the **IRQ** pin 0: Reflect MAX_RT as active low interrupt on the **IRQ** pin
	EN_CRC	3	1	R/W	Enable CRC. Forced high if one of the bits in the EN_AA is high
	CRCO	2	0	R/W	CRC encoding scheme '0' - 1 byte '1' – 2 bytes
	PWR_UP	1	0	R/W	1: POWER UP, 0:POWER DOWN
	PRIM_RX	0	0	R/W	RX/TX control 1: PRX, 0: PTX
07	STATUS				Status Register (In parallel to the SPI command word applied on the **MOSI** pin, the STATUS register is shifted serially out on the **MISO** pin)
	Reserved	7	0	R/W	Only '0' allowed
	RX_DR	6	0	R/W	Data Ready RX FIFO interrupt. Asserted when new data arrives RX FIFO[c]. Write 1 to clear bit.
	TX_DS	5	0	R/W	Data Sent TX FIFO interrupt. Asserted when packet transmitted on TX. If AUTO_ACK is activated, this bit is set high only when ACK is received. Write 1 to clear bit.
	MAX_RT	4	0	R/W	Maximum number of TX retransmits interrupt Write 1 to clear bit. If MAX_RT is asserted it must be cleared to enable further communication.
	RX_P_NO	3:1	111	R	Data pipe number for the payload available for reading from RX_FIFO 000-101: Data Pipe Number 110: Not Used 111: RX FIFO Empty
	TX_FULL	0	0	R	TX FIFO full flag. 1: TX FIFO full. 0: Available locations in TX FIFO.

Many of the commands required to utilize the nRF24L01+ transceivers could be run in any state, such as reading or writing to registers, but some of the commands, such as reading from the RX buffer and writing to the TX buffer could only be used when the device was in its correct state. A state diagram for the nRF24L01+ transceiver is shown below in Figure 2.3, where the `CE` is one of the pins on the nRF24L01+ transceiver (which was not part of the SPI pins), and the `PWR_UP` and `PRIM_RX` were both bits that could be set in the `CONFIG` register.

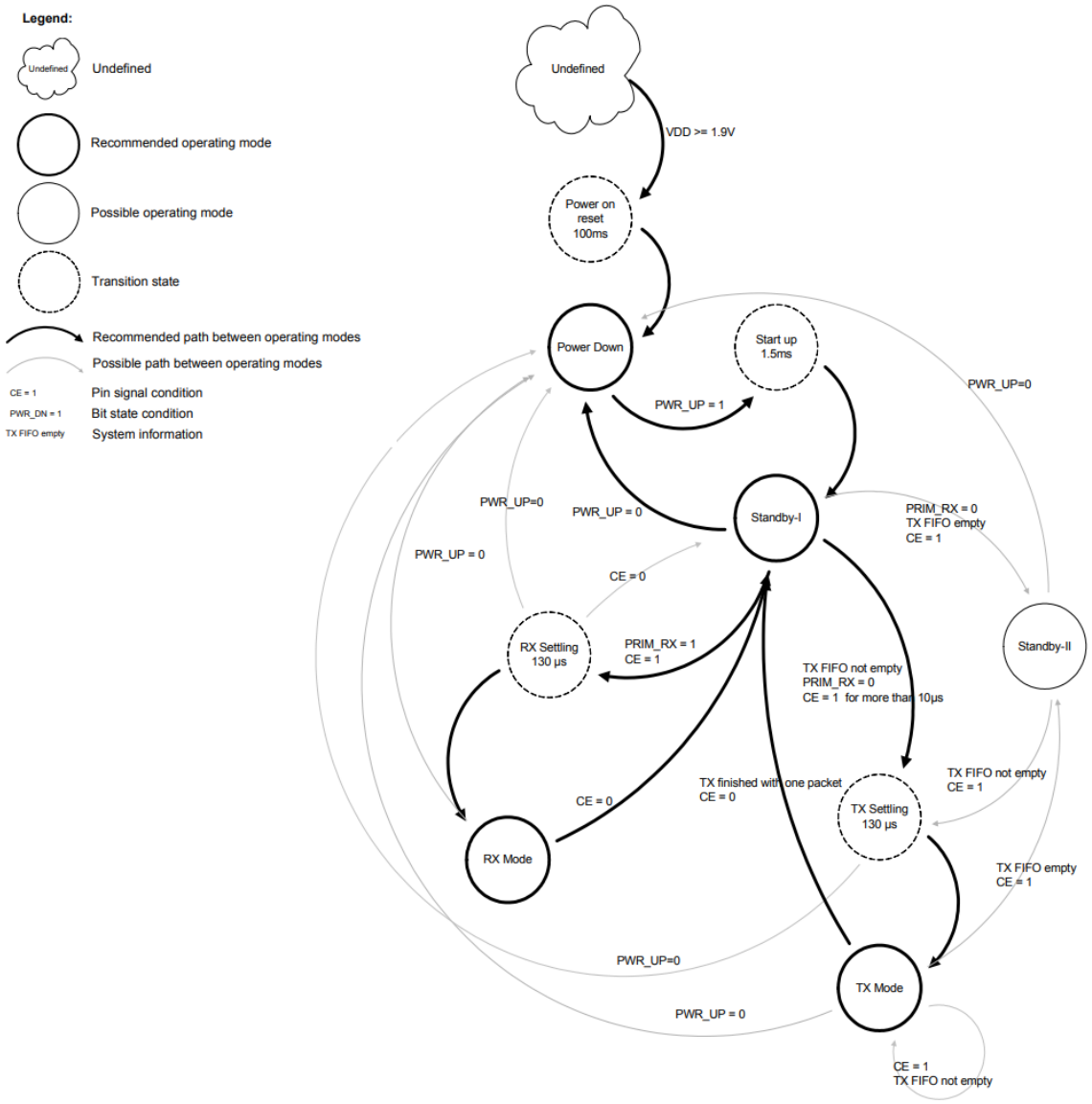

Figure 2.3: State diagram of the nRF24L01+ transceiver [1]

To successfully utilize the nRF24L01+ transceiver, this state diagram had to be followed to put the device in either RX mode or TX mode, depending on whether the device was meant to receive or transmit data. One last thing to note is that in RX mode, once the data was received by

the device, the IRQ pin would transition from high to low, and this is what allows for the reception of data to be interrupt driven.
To be able to read temperature, pressure, and humidity, a BME280 Environmental sensor was used. This sensor supports both SPI and I2C communication protocols; however, this project only used SPI communication. The sensor has 6 pins: CSB (chip select), SCK (clock), MOSI/SDI(master out slave in), MISO/SDO(master in slave out), VCC, and GND. CSN, SCK, and MOSI are output pins, while MISO is an input pin. To be able to communicate with the sensor, the protocol in Figures 2.4-6 must be followed. To perform a read operation, CSB needs to be set to 0, and the RW bit must be set to one. Additionally, the address of the register must be provided. In Table 2.4, the memory mapping for the sensor can be seen. Note that the MSB of the address is ignored due to the RW control bit. In the read operation, the address will automatically increment and send data. For example, in Figure 2.5 it shows a read control byte being sent for register address 0xF6. The sensor will send the data at 0xF6, and then will send 0xF7 right after it, and increment until the last register is reached or the communication is stopped. To perform a write operation, the RW bit in the control byte must be sent to zero, and the register address must be sent. After the control byte is sent, the data byte must be sent after it. Note that the register address does not automatically increment like in the read operation. In the register memory map (Table 2.4) for the sensor, it should be noted that each register is a byte long.

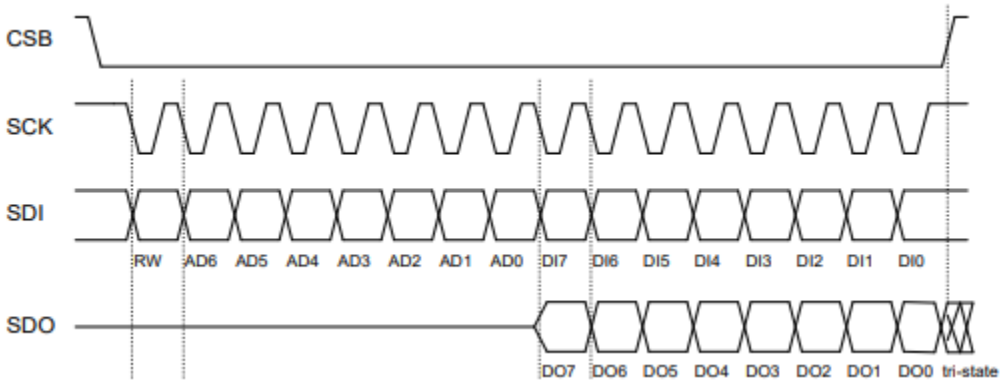

Figure 2.4: SPI Communication Waveform for BME280 [2]

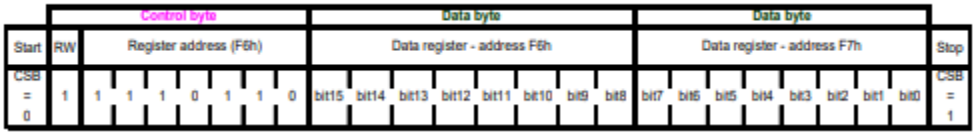

Figure 2.5: SPI Multi-Byte Read [2]

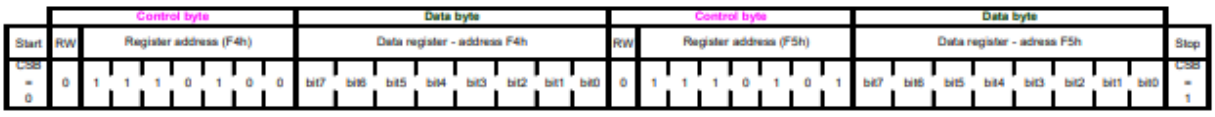

Figure 2.6: SPI Multi-Byte Write [2]

Table 2.4: BME280 Registers [2]

Register Name	Address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state
hum_lsb	0xFE	hum_lsb<7:0>								0x00
hum_msb	0xFD	hum_msb<7:0>								0x80
temp_xlsb	0xFC	temp_xlsb<7:4>				0	0	0	0	0x00
temp_lsb	0xFB	temp_lsb<7:0>								0x00
temp_msb	0xFA	temp_msb<7:0>								0x80
press_xlsb	0xF9	press_xlsb<7:4>				0	0	0	0	0x00
press_lsb	0xF8	press_lsb<7:0>								0x00
press_msb	0xF7	press_msb<7:0>								0x80
config	0xF5	t_sb[2:0]			filter[2:0]				spi3w_en[0]	0x00
ctrl_meas	0xF4	osrs_t[2:0]			osrs_p[2:0]			mode[1:0]		0x00
status	0xF3					measuring[0]			im_update[0]	0x00
ctrl_hum	0xF2						osrs_h[2:0]			0x00
calib26..calib41	0xE1…0xF0	calibration data								individual
reset	0xE0	reset[7:0]								0x00
id	0xD0	chip_id[7:0]								0x60
calib00..calib25	0x88…0xA1	calibration data								individual

Registers:	Reserved registers	Calibration data	Control registers	Data registers	Status registers	Chip ID	Reset
Type:	do not change	read only	read / write	read only	read only	read only	write only

To display the data on the LCD display, the Altera 16x2 character LCD interface can be used during the Qsys hardware design process. The module itself hardly has any sort of configuration, with only the option for what type of cursor to implement (option between normal, blinking, both, or none). With this module implemented, the easiest way to control the LCD display now is to use the included HAL library, which can be used after including the file `altera_up_avalon_character_lcd.h`. Using this library in the design process is far easier than controlling the module manually by reading and writing to its registers, as the display can be easily initialized and written to with just a few commands. Shown below in Figure 2.7 is an example snippet of code used to display the text “Welcome to the DE2 board” to the LCD display.

```
#include "altera_up_avalon_character_lcd.h"

int main(void)
{
	alt_up_character_lcd_dev * char_lcd_dev;

	// open the Character LCD port
	char_lcd_dev = alt_up_character_lcd_open_dev ("/dev/Char_LCD_16x2");
	if ( char_lcd_dev == NULL)
		alt_printf ("Error: could not open character LCD device\n");
	else
		alt_printf ("Opened character LCD device\n");

	/* Initialize the character display */
	alt_up_character_lcd_init (char_lcd_dev);

	/* Write "Welcome to" in the first row */
	alt_up_character_lcd_string(char_lcd_dev, "Welcome to");

	/* Write "the DE2 board" in the second row */
	char second_row[] = "the DE2 board\0";
	alt_up_character_lcd_set_cursor_pos(char_lcd_dev, 0, 1);
	alt_up_character_lcd_string(char_lcd_dev, second_row);
}
```

Figure 2.7: Example code for displaying text to the character LCD display [3]

In this example code, the LCD device is opened using the alt_up_character_lcd_open_dev command, which as a parameter takes in the name specified for the device in the Qsys design process. After successfully opening, the display is initialized using the alt_up_character_lcd_init command, making the device ready to display to. Once initialized, the alt_up_character_lcd_string command can be used to write a string to the LCD display, and the alt_up_character_lcd_set_cursor_pos can be used to change the cursor position, in this case moving it to the next line. With just these few commands, a message can be displayed on the LCD display, so long as the message consists of at most 16 characters per line and consists of ASCII characters.

To use the SD card for this system, the Altera SD card interface can be instantiated during the Qsys design process. Similar to the LCD device, using this interface also allows for the use of the corresponding SD card HAL library. This allows users to not only access the data in the SD card, but to access it through the FAT 16 file system, making the SD card format compatible with modern computers. By using files in the design process to store data, the data files can easily be transferred to a personal computer as well and can either be interpreted directly or an interpreter could be written for them. Shown below in Table 2.5 is a list of not all the HAL subroutines, but some of the most useful and frequently used in this project.

Table 2.5: List of some of the available HAL subroutines available for the SD card interface [4]

Prototype:	alt_up_sd_card_dev* alt_up_sd_card_open_dev(const char *name)
Inputs:	const char *name - the instance name of SD Card IP Core in the Qsys system
Outputs:	alt_up_sd_card_dev* - a pointer to a structure holding a *base* field. The *base* field holds the address of the SD Card IP Core in the Qsys system.
Description:	Initializes the SD Card IP Core HAL device driver. Returns NULL, when the specified device name does not exist in the system.

Prototype:	short int alt_up_sd_card_fopen(char *name, bool create)
Inputs:	char *name - name of the file to open, relative to root directory bool create - set to **true** if the file should be created if it is not found
Outputs:	short int - A handle to the opened file. A negative result indicates an error as follows: -1 means that the file could not be opened, and -2 means the file is already open.
Description:	Opens a file for use in your application.

Prototype:	short int alt_up_sd_card_read(short int file_handle)
Inputs:	short int file_handle - handle to a file as returned by the alt_up_sd_card_fopen
Outputs:	short int - a byte of data (0-255) when successful, or a negative value when failure occurs. -1 indicated the End-of-File, and -2 indicates an inability to read from the SD card.
Description:	Reads a byte of data from a given file at current position in the file. The position in the file is incremented when data is read.

Prototype:	bool alt_up_sd_card_write(short int file_handle, unsigned char byte_of_data)
Inputs:	short int file_handle - handle to a file as returned by the alt_up_sd_card_fopen unsigned char byte_of_data - a byte of data to be written
Outputs:	bool - **true** when successful and **false** when unsuccessful
Description:	Writes a byte of data at the current position in the file. The position in the file is incremented when data is written.

Prototype:	bool alt_up_sd_card_fclose(short int file_handle)
Inputs:	short int file_handle - handle to a file as returned by the alt_up_sd_card_fopen
Outputs:	bool - **true** when successful and **false** when unsuccessful
Description:	Closes a previously opened file and invalidates the given file_handle.

Prototype:	bool alt_up_sd_card_is_Present(void)
Inputs:	None
Outputs:	bool - **true** when an SD card is present in the SD card socket, and **false** otherwise
Description:	Checks if an SD card is present in the SD card socket

Using the commands in this list, a program could easily be implemented to initialize the device, open the specified file, read from or write to the file byte-wise, and then close the file when completed. The alt_up_sd_card_is_Present subroutine could also be used to force the program to wait for an SD card to be inserted before continuing with the program.

3. EXPERIMENTAL PROCEDURE

3.1 Hardware and Software Used

Hardware:

- Two DE2-115
    - A-B USB cable
    - 12V Power cable
- Two PCs
- Waveshare BME280 Environmental Sensor
- Two nRF24L01+ 2.4 GHz Wireless Transceivers
- Wires for GPIO connection

Software:

- Quartus Prime Lite v16.1
    - Qsys
- Nios II Embedded Development - Software Build Tools For Eclipse

3.2 Procedure for Collector Hardware

The first task that must be completed is to set up the hardware for both the collector and display systems (two different systems were made). This was done using Quartus Prime Lite and its Qsys function. The base of both systems were already set up through the previous assignments in this course.

Normally, to utilize the SPI communication, an SPI core provided by the Quartus software would be used. However, there were difficulties communicating with the sensor and transceivers using these default cores. Because of this, a technique called “bit banging” was used to control the communication through software, rather than relying on the SPI core in Qsys. To achieve this, on the sensor system, PIOs of 1 bit length were added for each data pin on the sensor. The CSN, SCK, and MOSI pins were set to be output PIOs, while the MISO pin was set to be an input PIO. A similar process was followed for the nRF24L01+ transceiver, where a total of 6 PIOs were added of one bit length. The CSN, SCK, MOSI, and CE were set to be outputs, while the IRQ and MISO pins were set to be inputs.

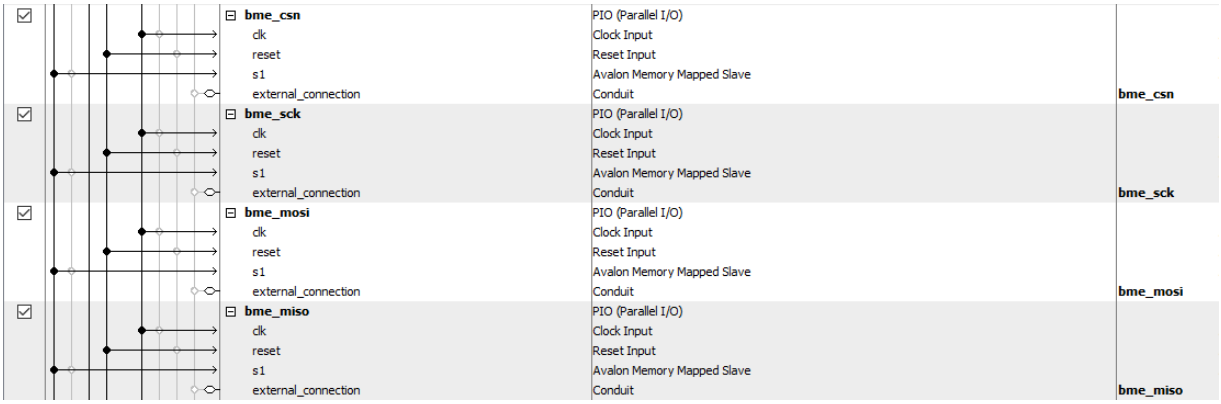

Figure 3.1: BME280 Sensor PIOs in Qsys

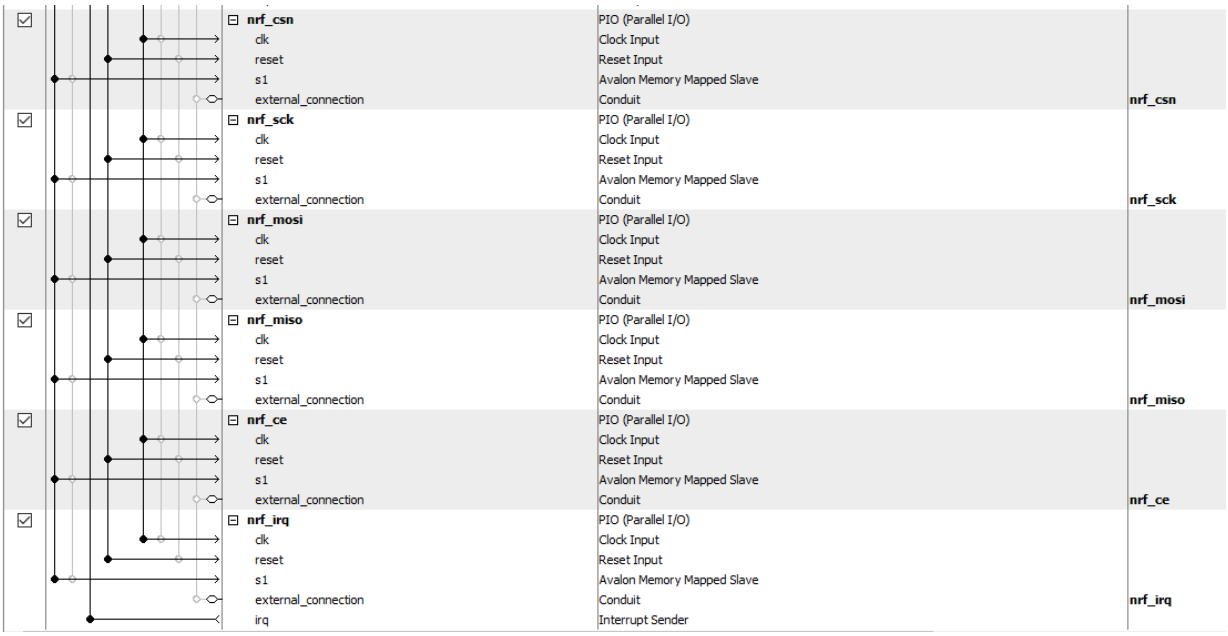

Figure 3.2: nRF24L01+ Transceiver PIOs in Qsys

Once these were added, the verilog files were generated for the system. After this, the top level Verilog module was made by referencing the Verilog file that was generated during Qsys. Next, the pins must be assigned by using the Pin Planner on Qsys. The new PIOs that were added were assigned to the GPIO pins on the right side of the DE2-115, which can be seen in Figure 3.3. The BME pins were assigned to the GPIO pins on the upper part of the GPIO pins (see Figure 3.4). The VCC and GND pin assignment is not shown in the Pin Planner, but they were plugged into the pins directly under AD15 and AE15. The nRF24L01+ were assigned to the lower part of the GPIO pins (see Figure 3.6) and the VCC and GND pins were plugged into the pins above AG22 and AE24. Once the pin assignment was completed, the project was compiled and the hardware setup for the data collector system is now complete.

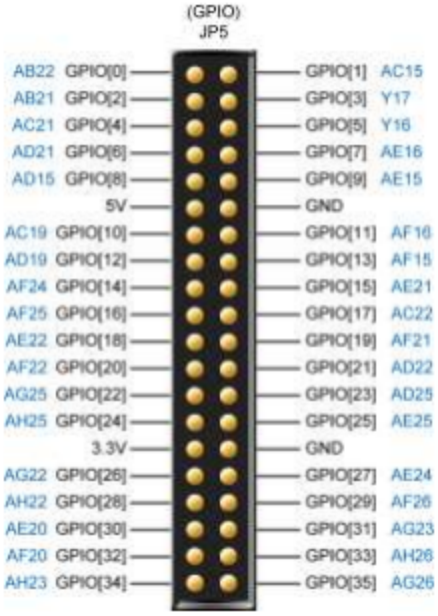

Figure 3.3: GPIO Pins on the DE2-115 [5]

out BME_CSN	Output	PIN_AD21
in BME_MISO	Input	PIN_AE15
out BME_MOSI	Output	PIN_AE16
out BME_SCK	Output	PIN_AD15

Figure 3.4: Pin Assignments for BME280

out NRF_CE	Output	PIN_AE25
out NRF_CSN	Output	PIN_AH25
in NRF_IRQ	Input	PIN_AF22
in NRF_MISO	Input	PIN_AD22
out NRF_MOSI	Output	PIN_AG25
out NRF_SCK	Output	PIN_AD25

Figure 3.5: Pin Assignments for nRF24L01+

3.3 Procedure for Collector Software

The next step for the collector system was the software set up. In the software, before the main data collection loop can start, the PIOs must be set up. This involves setting up the timer IRQ, the period, and the START, CONT, and ITO bits. Additionally, the nRF24L01+ must be configured by setting the SCK to 0 and the CSN to 1. Next, the auto ACK must be set by sending two bytes: 0x3D and 0x6 to the transceiver. After this, the TX FIFO must be flushed by sending

0xE1 to the transceiver. Finally, the status register must be set by sending 0x27 and 0x1E. These bytes were sent by using a spi_comm_nrf function - details of this function will be provided in the analysis section. Once the initial setup for the transceiver is done, the environmental sensor needs to be set up as well. This is first done by setting the SCK to 0 and CSN 1 to one (similar to the transceiver). Next, the oversampling settings need to be set for the temperature, pressure, and humidity on the sensor by writing to the crtl_meas register and crtl_hum registers. Note that the oversampling for the crtl_hum register can only be set after an effective write operation to the crtl_meas register. Note that the data is sent and read from the sensor using a spi_comm function (similar to spi_comm_nrf), and details of this function will be discussed in the analysis. After this, the calibration data needs to be read for the temperature, pressure, and humidity. This calibration data is found in registers 0x88 to 0xA1 and 0xE1 to 0xF0. Details on why this calibration is needed will be discussed in the analysis section. Once this is done, the main state of the program can be entered. Every 5 seconds, the system will repeat the following process. It will set the transceiver to TX mode by sending 0x20 and 0x0A to the transceiver. After a short delay, the system will read the sensor data byte by sending 0xF7 to the sensor. As discussed in the theoretical background, the sensor will send the register data back, increment to the next register, and then send the register data. This process will repeat for 8 registers (3 for temperature, 3 for pressure, and 2 for humidity). The sensor data will then be passed through their corresponding compensation functions and assigned to the global result values. The compensation functions will be discussed in the analysis. Next, the CE for the transceiver will be set to 1, and the program will delay for at least 130 µs. After this delay, the result data will be separated into bytes and sent to the transceiver to prepare for transmission. After a 1 ms delay, the CE for the transceiver will be set to 0, which tells the transmitter to start sending the data in the TX FIFO. Once the data has been sent, the same process of reading and transmitting the data will repeat until the system shuts down.

3.4 Procedure for Display Hardware

To create the hardware for the data display system, the same base system which had been used in previous assignments was once again used, including the NIOS II core, timer, SDRAM, and all the existing PIOs. In this device, the transceiver hardware was also added, once again following the Qsys implementation shown in Figure 3.2 as well as the pin assignments shown in Figure 3.5. Because this system was intended to receive the data and needed to perform the corresponding interrupt-driven subroutine for the data reception process, the PIO for the transceiver's IRQ was configured to assert an interrupt on the falling edge of the signal, as the signal made a transition from high to low once data had been received. The configuration of this IRQ signal in the PIO configuration can be seen below in Figure 3.6.

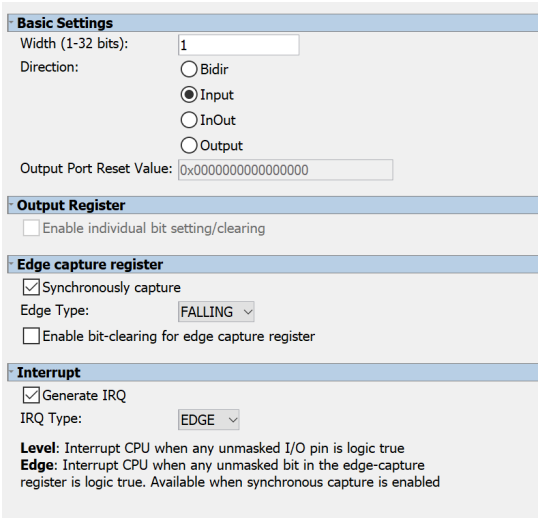

Figure 3.6: Configuration of IRQ PIO for the transceiver in the display device

Also added to this system were the 16x2 character LCD controller and SD card controller in order to display and store the data respectively. For the LCD controller, the only configuration to be made was to turn the cursor off, since it was not needed to represent the data on the screen. For the SD card controller, no configuration was available or necessary in order to successfully access the SD card data. The instantiation of both of these modules in Qsys can be seen below in Figure 3.7 for the LCD controller and Figure 3.8 for the SD card controller.

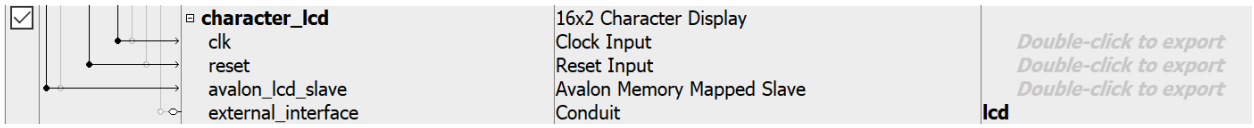

Figure 3.7: Qsys implementation of character LCD controller

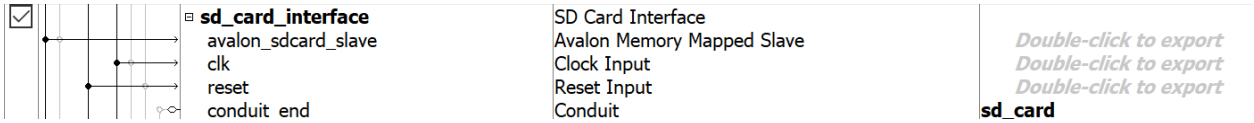

Figure 3.8: Qsys implementation of SD card interface controller

Once these had been implemented in Qsys, the top verilog module had to be updated to reflect the newly added I/O used to drive the newly added devices. This included I/O for the transceiver, the LCD display, and the SD card. The full top verilog module including these new I/O can be found at the end of this document in Appendix B. After running a validation check to see that the top module had been properly configured, the pins for the new hardware had to be assigned. As previously mentioned, the pin assignment for the transceiver follows the same pin assignment used in the collector device, with the GPIO pins shown in Figure 3.3 and the assignments made in the pin planner shown in Figure 3.5. To assign the pins for the LCD, the pin assignments found in the DE2-115 user manual were followed, which are shown below in Table 3.1. Shown

in Figure 3.9 are the assignments which ended up being made in the pin planner. Similarly, the DE2-115 user manual was also followed for the pin assignments for the SD card interface, with the arrangement from the DE2-115 user manual shown in Table 3.2 and the actual pin assignments made in the pin planner shown in Figure 3.10.

Table 3.1: Pins for LCD display module [5]

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
LCD_DATA[7]	PIN_M5	LCD Data[7]	3.3V
LCD_DATA[6]	PIN_M3	LCD Data[6]	3.3V
LCD_DATA[5]	PIN_K2	LCD Data[5]	3.3V
LCD_DATA[4]	PIN_K1	LCD Data[4]	3.3V
LCD_DATA[3]	PIN_K7	LCD Data[3]	3.3V
LCD_DATA[2]	PIN_L2	LCD Data[2]	3.3V
LCD_DATA[1]	PIN_L1	LCD Data[1]	3.3V
LCD_DATA[0]	PIN_L3	LCD Data[0]	3.3V
LCD_EN	PIN_L4	LCD Enable	3.3V
LCD_RW	PIN_M1	LCD Read/Write Select, 0 = Write, 1 = Read	3.3V
LCD_RS	PIN_M2	LCD Command/Data Select, 0 = Command, 1 = Data	3.3V
LCD_ON	PIN_L5	LCD Power ON/OFF	3.3V
LCD_BLON	PIN_L6	LCD Back Light ON/OFF	3.3V

Node Name	Direction	Location
LCD_DATA[7]	Bidir	PIN_M5
LCD_DATA[6]	Bidir	PIN_M3
LCD_DATA[5]	Bidir	PIN_K2
LCD_DATA[4]	Bidir	PIN_K1
LCD_DATA[3]	Bidir	PIN_K7
LCD_DATA[2]	Bidir	PIN_L2
LCD_DATA[1]	Bidir	PIN_L1
LCD_DATA[0]	Bidir	PIN_L3
LCD_EN	Output	PIN_L4
LCD_ON	Output	PIN_L5
LCD_RS	Output	PIN_M2
LCD_RW	Output	PIN_M1

Figure 3.9: Pin assignments for LCD display module made in pin planner

Table 3.2: Pins for SD card interface module [5]

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
SD_CLK	PIN_AE13	SD Clock	3.3V
SD_CMD	PIN_AD14	SD Command Line	3.3V
SD_DAT[0]	PIN_AE14	SD Data[0]	3.3V
SD_DAT[1]	PIN_AF13	SD Data[1]	3.3V
SD_DAT[2]	PIN_AB14	SD Data[2]	3.3V
SD_DAT[3]	PIN_AC14	SD Data[3]	3.3V
SD_WP_N	PIN_AF14	SD Write Protect	3.3V

Node Name	Direction	Location
SD_CLK	Output	PIN_AE13
SD_CMD	Bidir	PIN_AD14
SD_DAT	Bidir	PIN_AE14
SD_DAT3	Bidir	PIN_AC14

Figure 3.10: Pin assignments for SD card interface module made in pin planner

3.5 Procedure for Display Software

For the final part of this procedure, the display software had to be created. The entirety of this software was interrupt driven, and so the only parts included in the main function are all for initialization purposes. First, the timer was configured, since it was needed in order to wait the proper amount of time for the startup of the transceiver. After this, the transceiver was initialized by first initializing its IRQ, followed by a 150 ms wait time for the transceiver to turn on. After this, the rest of the IRQ initialization was called (primarily just the IRQ for the push buttons), followed by further initialization to put the transceiver into RX mode to receive data, then the initialization of the LCD display, and finally the initialization of the SD card interface. The card further initialized and put into RX mode by first setting the SCK to 0 and CSN to 1, leaving the PIO interface at an idle state. After this, auto-acknowledgement was enabled, followed by resetting the status register, setting the data pipe byte count to 12 (as each package to be received would be 12 bytes long), then flushing the RX FIFO. Finally, the transceiver was put in RX mode by setting the `PWR_UP` bit to 1, then after a 2 ms wait, enabling the `CE` input as well as the `PRIM_RX` bit in order to transition the transceiver into RX mode. The LCD and SD card interfaces were initialized by opening both devices using the provided HAL subroutines, with the SD card involving a bit of extra logic to ensure that the SD card is inserted correctly before continuing.

With all of the initialization ready, it was up to the interrupt handlers for the transceiver and push buttons to continue the functionality of this software, as from here on out the program was

entirely event driven. Upon new data being received by the transceiver, the interrupt handler was called, and the RX buffer was read using an SPI command sent to the transceiver. After piecing together the individual bytes back into their corresponding float values, these received values were then written to the `data.txt` file in the SD card. If the most recent data was currently displayed on the LCD, the LCD would be automatically updated to show this newly received data. If a push button interrupt was triggered, the interrupt handler would first evaluate which button had been pressed, and then took the corresponding action. `KEY1` was intended to switch between temperature, humidity, and pressure, which it cycled through by updating the `display` variable which determined which metric the display would update to after retrieving the corresponding data from the SD card. `KEY2` and `KEY3` were used to search forwards and backwards throughout time respectively, and after the corresponding data had been retrieved from the SD card, it would be displayed to the LCD screen as well. Upon any button press, the SD card was accessed by iterating a certain amount of times to find the data at the proper point in time (indicated by the `index` variable), and then the data retrieved would be stored into the temperature, humidity, and pressure variables, which would then be the values which the display would update to.

4. ANALYSIS

4.1 Source Code and Results

```
void spi_comm (uint8_t input[], uint8_t output[], int in_length, int out_length)
{
    for(int i = 0; i < out_length; i++) //Initializing output array to all zeros
    {
        output[i] = 0;
    }

    IOWR(BME_CSN_BASE, 0, 0);   //CSN to 0 to begin comm
    for(int i = 0; i < ((in_length > out_length)?in_length:out_length); i++)    //Loop once for each byte of communication
    {
        for(int j = 7; j >= 0; j--) //Loop 8 times for each byte
        {
            IOWR(BME_SCK_BASE, 0, 0);   //Clock falling edge
            if(i < in_length)   //Change input byte (if still inputting)
                IOWR(BME_MOSI_BASE, 0, (input[i] & (1 << j))?1:0);
            if(i < out_length)  //Change output byte (if still outputting)
                output[i] = output[i] | (IORD(BME_MISO_BASE, 0) << j);
            IOWR(BME_SCK_BASE, 0, 1);   //Clock rising edge (when input is read by transceiver
        }
    }
    IOWR(BME_SCK_BASE, 0, 0);   //Reset clock back to 0
    IOWR(BME_CSN_BASE, 0, 1);   //Reset CSN back to 1, ending communication
}
```

Figure 4.1: SPI Communication Function

```
// Compensate ADC temperature input function (from data sheet)
long signed int BME280_compensate_T_int32(long signed int adc_T)
{
    long signed int var1, var2, T;
    var1 = ((((adc_T>>3) - ((long signed int)dig_T1<<1))) * ((long signed int)dig_T2)) >> 11;
    var2 = (((((adc_T>>4) - ((long signed int)dig_T1)) * ((adc_T>>4) - ((long signed int)dig_T1))) >> 12) * ((long signed int)dig_T3)) >> 14;
    t_fine = var1 + var2;
    T = (t_fine * 5 + 128) >> 8;
    return T;
}
```

Figure 4.2: One of the Compensation Functions for Sensor Data [2]

```c
// Print sensor data and compensate the data
void print_array_v (uint8_t a[], int size)
{
    if(size > 1)
    {
        printf("Received Sensor Data: ");
        for(int i = 0; i < size - 1; i++)
        {
            alt_printf("%x, ", (a[i] & 0xFF));
        }
    }
    alt_printf("%x\n", (a[size - 1] & 0xFF));

    // temperature
    long signed int t = 0;
    t = (a[4] << 12) + (a[5] << 4) + a[6];
    t = t & 0x000FFFFF;
    t_result = BME280_compensate_T_int32(t);
    printf("compensated T: %g\n", t_result * 0.01);

    // pressure
    long signed int p = 0;
    p = (a[1] << 12) + (a[2] << 4) + a[3];
    p = p & 0x000FFFFF;
    p_result = BME280_compensate_P_int64(p);
    p_result_float = ((float)p_result/256);
    printf("compensated P: %g\n", p_result_float);

    // humidity
    long signed int h = 0;
    h = (a[7] << 8) + a[8];
    h = h & 0xFFFF;
    h_result = bme280_compensate_H_int32(h);
    h_result_float = ((float)h_result / 1024);
    printf("compensated H: %g\n", h_result_float);
    return;
}
```

Figure 4.3: Function to Print and Compensate Data

```c
else if (SETUP == 3) {
  // get Sensor data
  IOWR(TIMER_0_BASE, 1, 8);
  test_input[0] = 0xF7;
  spi_comm(test_input, test_output, 1, 9);
  print_array_v(test_output, 9);
```

Figure 4.4: Reading BME Registers

```
test_input[0] = 0b10100000;
union float_by_bit p;
p.f = p_result_float;
union float_by_bit h;
h.f = h_result_float;
// Moving result data into bytes
test_input[1] = (t_result >> 24) & 0xFF;
test_input[2] = (t_result >> 16) & 0xFF;
test_input[3] = (t_result >> 8) & 0xFF;
test_input[4] = (t_result >> 0) & 0xFF;
test_input[5] = (p.i >> 24) & 0xFF;
test_input[6] = (p.i >> 16) & 0xFF;
test_input[7] = (p.i >> 8) & 0xFF;
test_input[8] = (p.i >> 0) & 0xFF;
test_input[9] = (h.i >> 24) & 0xFF;
test_input[10] = (h.i >> 16) & 0xFF;
test_input[11] = (h.i >> 8) & 0xFF;
test_input[12] = (h.i >> 0) & 0xFF;
printf("Transmitted data: ");
for (int i = 0; i < 13; i++) {
    printf("%x ", test_input[i]);
}
printf("\n");
spi_comm_nrf(test_input, test_output, 13, 0);
```

Figure 4.5: Preparing Compensated Environmental Data to be Transmitted

```
int display = 0; // 0 = temp, 1 = humidity, 2 = pressure
int index = 0;
int max_index = 0;
float temperature, humidity, pressure;
```

Figure 4.6: Global variables used in display device

```
void sd_write_data()
{
    union float_by_bit n;
    max_index++;
    short int handle = alt_up_sd_card_fopen(FILE_NAME, false);
    alt_up_sd_card_write(handle, max_index);
    for(int i = 0; i < max_index * 12; i++) {alt_up_sd_card_read(handle);}
    n.f = temperature;
    for(int i = 3; i >= 0; i--) {alt_up_sd_card_write(handle, (n.i >> (i*8)));}
    n.f = humidity;
    for(int i = 3; i >= 0; i--) {alt_up_sd_card_write(handle, (n.i >> (i*8)));}
    n.f = pressure;
    for(int i = 3; i >= 0; i--) {alt_up_sd_card_write(handle, (n.i >> (i*8)));}
    alt_up_sd_card_fclose(handle);

    if(index == max_index - 2)
    {
        index++;
        update_lcd();
    }

    if(max_index == 1)
        update_lcd();
}
```

Figure 4.7: Function to write new data to the SD card

```
void sd_read_data()
{
    union float_by_bit n;
    short int handle = alt_up_sd_card_fopen(FILE_NAME, false);
    alt_up_sd_card_read(handle);
    for(int i = 0; i < index * 12 + 12; i++) {alt_up_sd_card_read(handle);}
    n.i = 0;
    for(int i = 3; i >= 0; i--) {n.i = n.i | (alt_up_sd_card_read(handle) << (i*8));}
    temperature = n.f;
    n.i = 0;
    for(int i = 3; i >= 0; i--) {n.i = n.i | (alt_up_sd_card_read(handle) << (i*8));}
    humidity = n.f;
    n.i = 0;
    for(int i = 3; i >= 0; i--) {n.i = n.i | (alt_up_sd_card_read(handle) << (i*8));}
    pressure = n.f;
    alt_up_sd_card_fclose(handle);
}
```

Figure 4.8: Function to retrieve data from the SD card

```
void update_lcd()
{
    if(max_index == 0)
    {
        lcd_print("No data", "available yet.");
        return;
    }
    char *temp0 = (char*)malloc(16 * sizeof(char));
    char *temp1 = (char*)malloc(16 * sizeof(char));
    switch(display) {
        case 0:
            sprintf(temp0, "Temperature at");
            sprintf(temp1, "t=%d: %g C", index, temperature);
            lcd_print(temp0, temp1);
            break;
        case 1:
            sprintf(temp0, "Humidity at");
            sprintf(temp1, "t=%d: %g%%", index, humidity);
            lcd_print(temp0, temp1);
            break;
        case 2:
            sprintf(temp0, "Pressure at");
            sprintf(temp1, "t=%d: %g Pa", index, pressure);
            lcd_print(temp0, temp1);
            break;
    }
}
```

Figure 4.9: Function to update the lcd given the current index/time

```
void key_int_handler()
{
    int state = IORD(KEYS_BASE, 0);
    if(!(state & 2))
    {
        //alt_printf("KEY1\n");
        char *temp = (char*)malloc(16 * sizeof(char));
        if(display == 2)
            display = 0;
        else
            display++;
        sd_read_data();
        update_lcd();
    }
    else if(!(state & 4))
    {
        //alt_printf("KEY2\n");
        if(use_data_prompt)
        {
            no_response = 0;
            response = 0;
            return;
        }
        if(index < max_index - 1)
        {
            index++;
            sd_read_data();
            update_lcd();
        }
    }
    else if(!(state & 8))
    {
        //alt_printf("KEY3\n");
        if(use_data_prompt)
        {
            no_response = 0;
            response = 1;
            return;
        }
        if(index > 0)
        {
            index--;
            sd_read_data();
            update_lcd();
        }
    }
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE + 3, 0x0);
}
```

Figure 4.10: Push button interrupt handler to display the correct data

```
void nrf_int_handler()
{
    union float_by_bit n;

    //Reading RX payload
    nrf_input[0] = 0b01100001;
    spi_comm(nrf_input, nrf_output, 1, 13);
    //print_array(nrf_output, 13);

    unsigned int temp = 0;
    for(int i = 0; i < 4; i++)
    {
        temp = temp | (nrf_output[i + 1] << (3-i)*8);
    }
    temperature = temp * 0.01;
    //alt_printf("Temperature: %x\n", temp);
    temp = 0;
    for(int i = 0; i < 4; i++)
    {
        temp = temp | (nrf_output[i + 5] << (3-i)*8);
    }
    n.i = temp;
    pressure = n.f;
    //alt_printf("Pressure: %x\n", temp);
    temp = 0;
    for(int i = 0; i < 4; i++)
    {
        temp = temp | (nrf_output[i + 9] << (3-i)*8);
    }
    n.i = temp;
    humidity = n.f;
    //alt_printf("Humidity: %x\n", temp);

    //Resetting status register
    nrf_input[0] = 0x27;
    nrf_input[1] = 0x4E;
    spi_comm(nrf_input, nrf_output, 2, 1);

    printf("%g, %g, %g\n", temperature, pressure, humidity);

   sd_write_data();

   IOWR_ALTERA_AVALON_PIO_EDGE_CAP(NRF_IRQ_BASE + 3, 0x0);
}
```

Figure 4.11: Transceiver interrupt handler to store new data

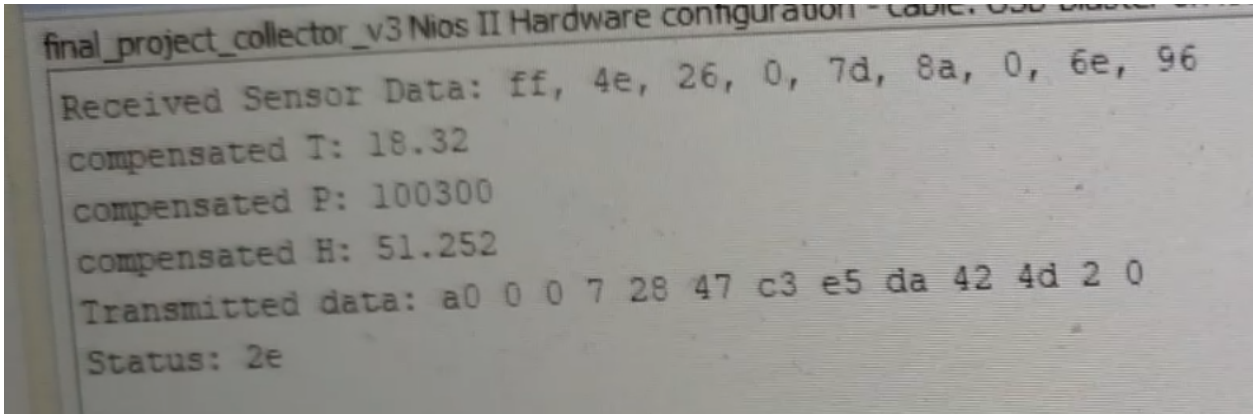

Figure 4.12: t = 0 Data Collector Console Output

```
Received Sensor Data: ff, 4d, c8, 0, 7f, 4e, 0, 6e, a9
compensated T: 21.92
compensated P: 101830
compensated H: 51.2588
Transmitted data: a0 0 0 8 90 47 c6 e2 ce 42 4d 9 0
Status: 2e
```

Figure 4.13: t = 1 Data Collector Console Output

```
Received Sensor Data: ff, 4e, 15, 0, 80, 59, 0, 74, f0
compensated T: 23.27
compensated P: 101151
compensated H: 60.3711
Transmitted data: a0 0 0 9 17 47 c5 8f 62 42 71 7c 0
Status: 2e
```

Figure 4.14: t = 18 Data Collector Console Output

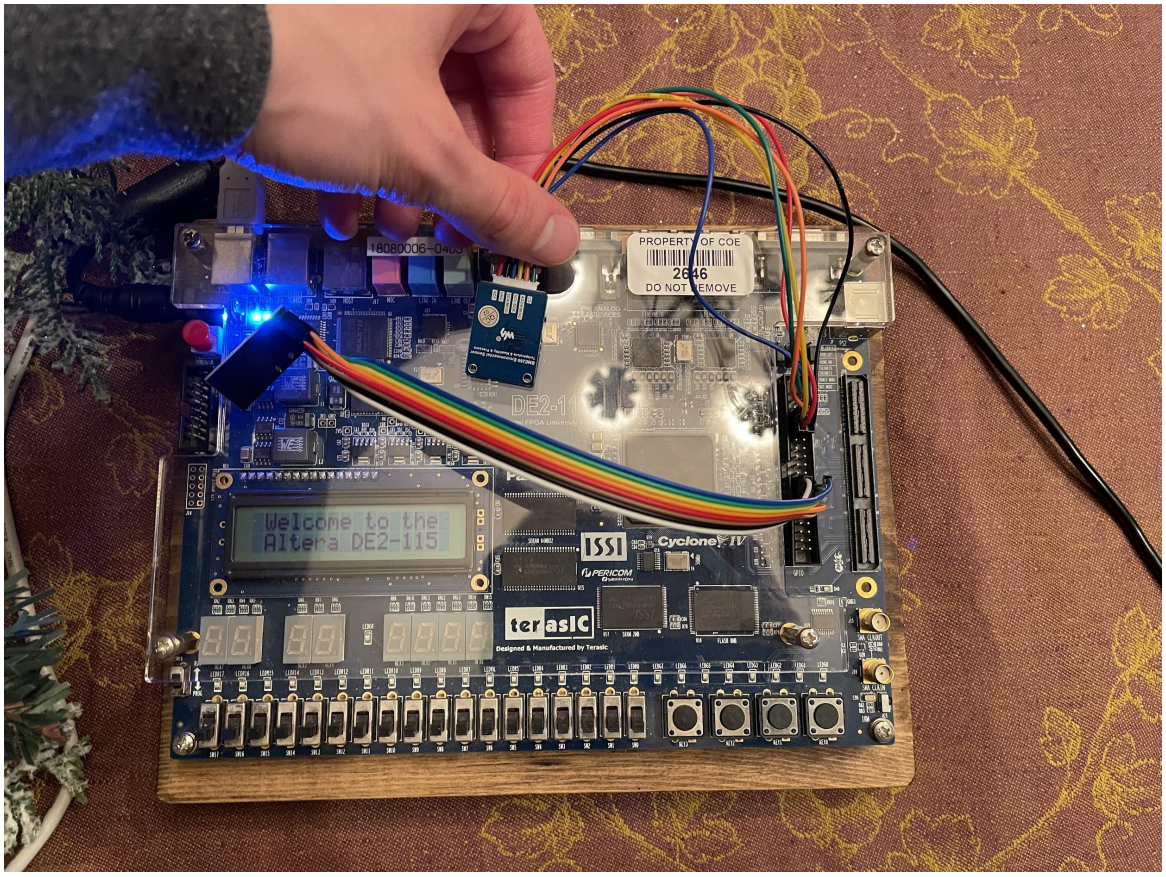

Figure 4.15: Data Collector Board

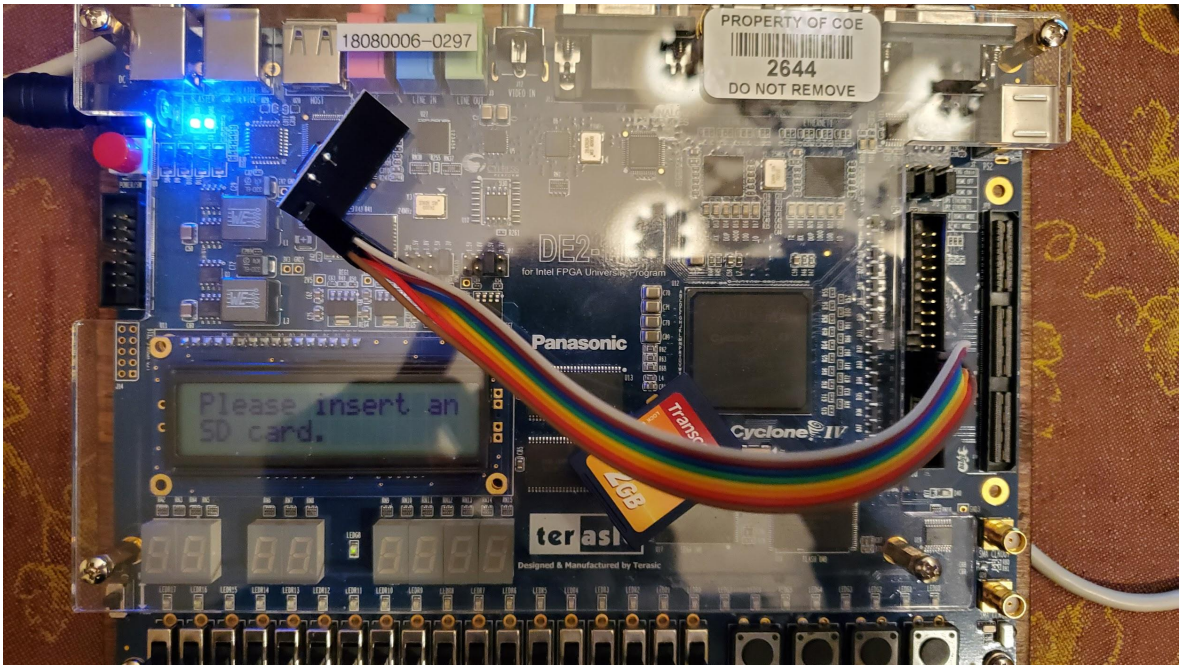

Figure 4.16: Display board on startup without SD card inserted

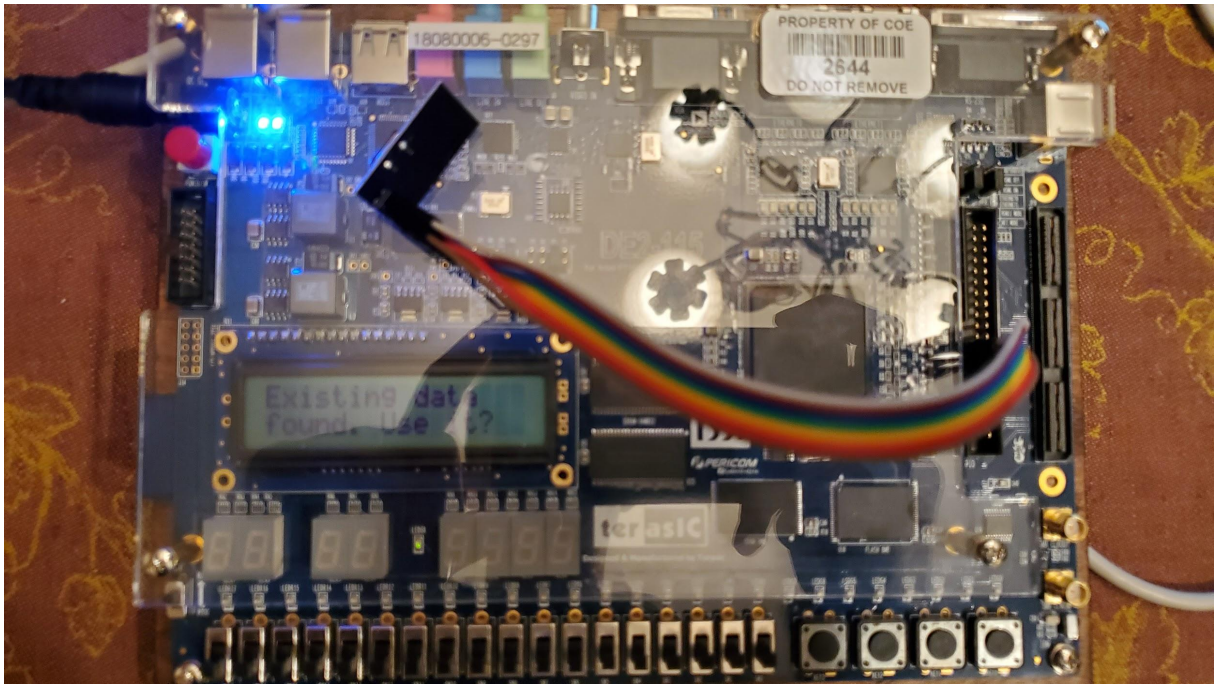

Figure 4.17: Display board with SD card inserted and option to use existing data

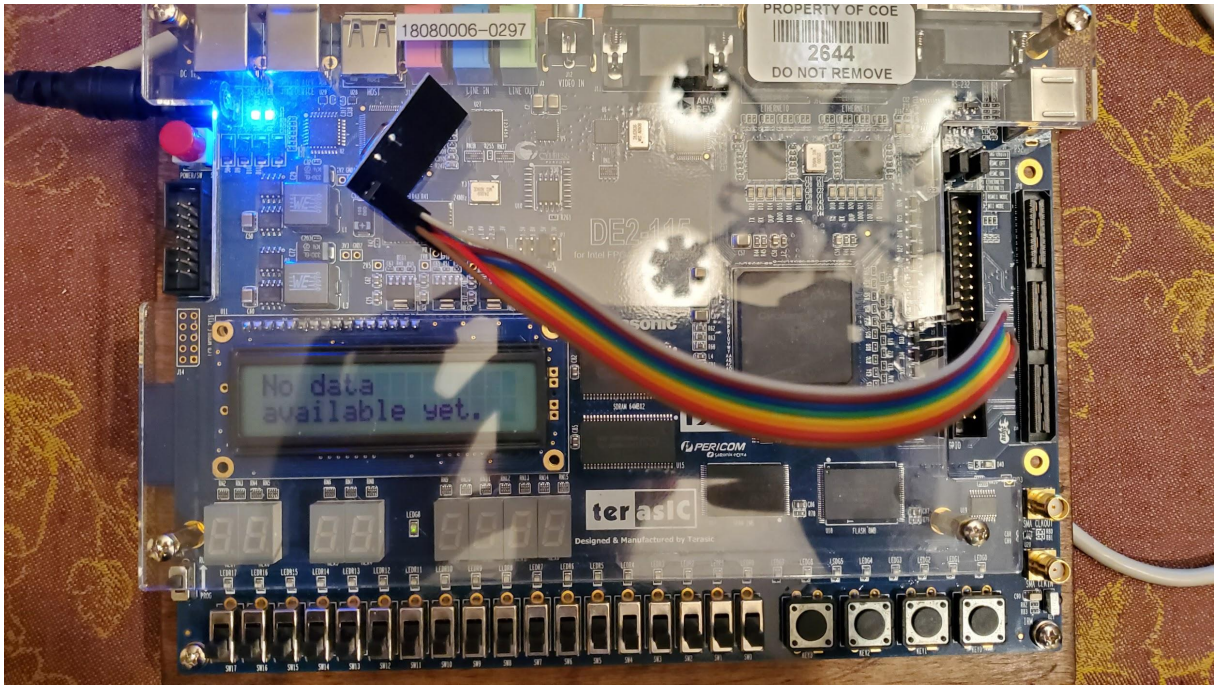

Figure 4.18: Display board with existing data unused and no new data yet

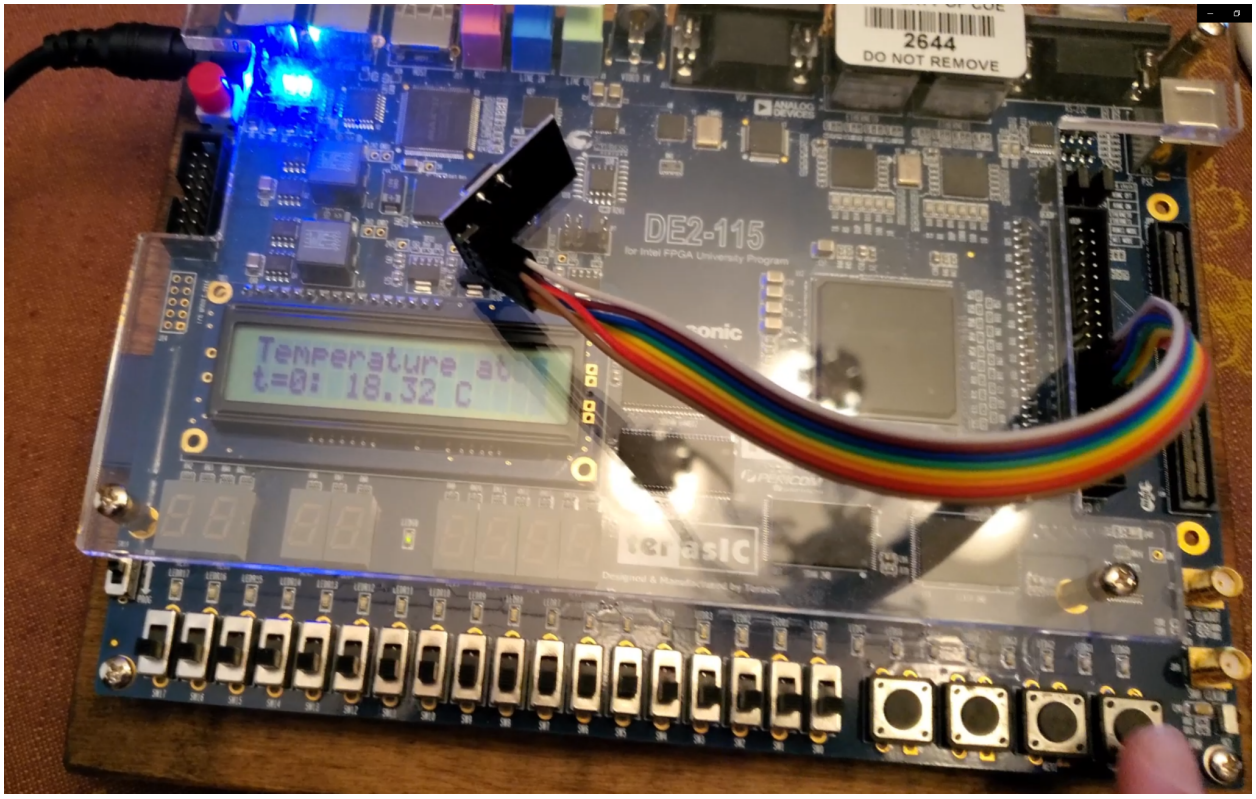

Figure 4.19: Display board showing first collected temperature at t = 0

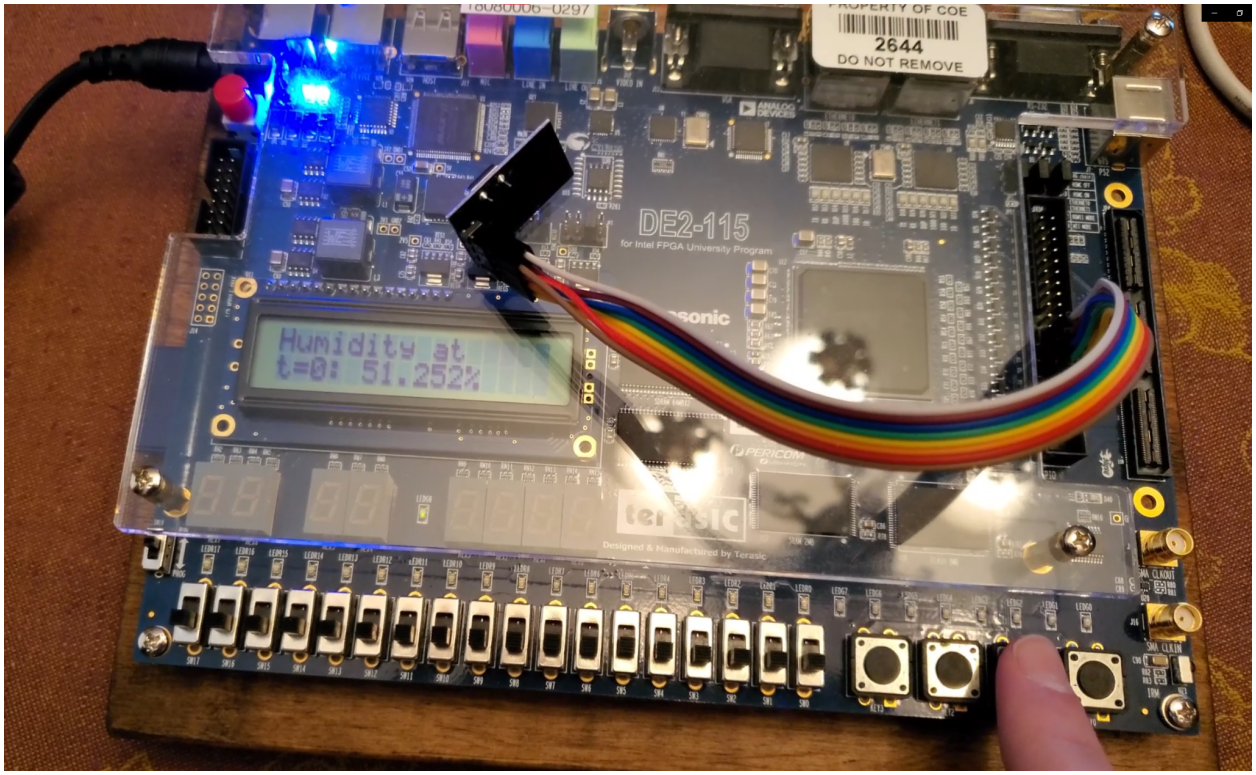

Figure 4.20: Display board showing first collected humidity at t = 0

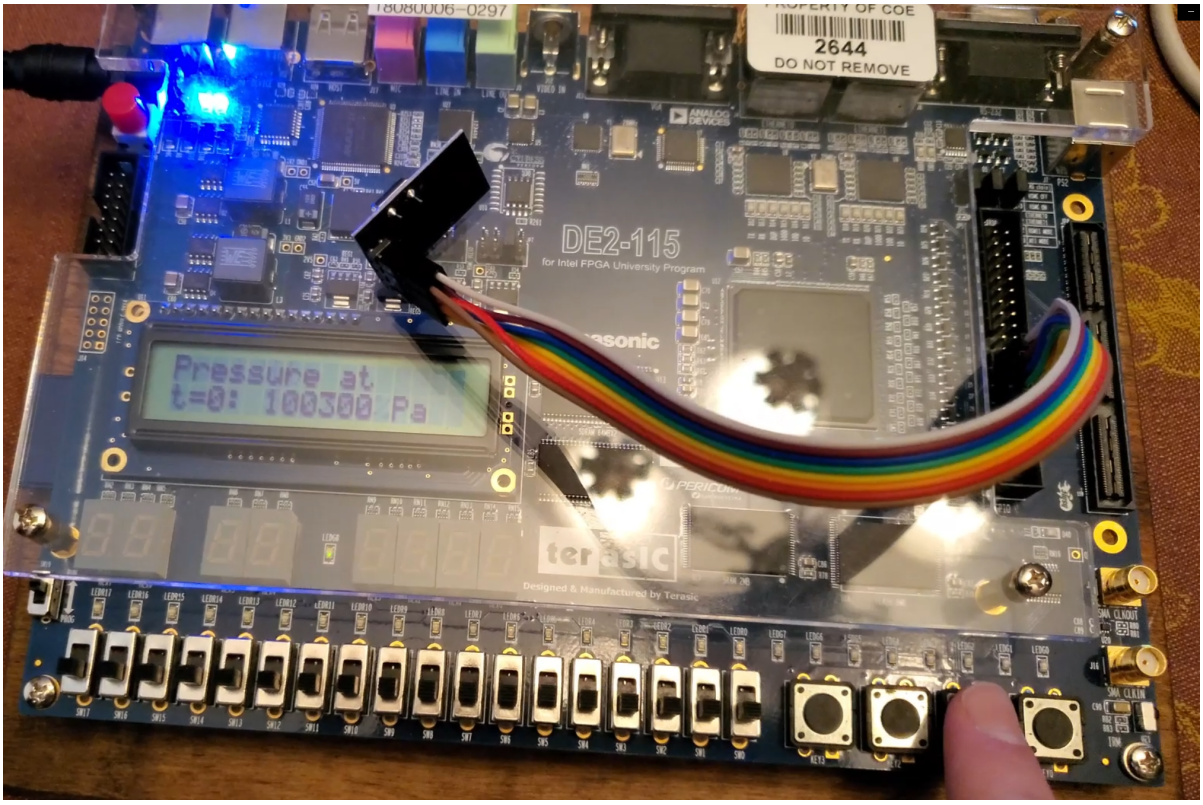

Figure 4.21: Display board showing first collected pressure at t = 0

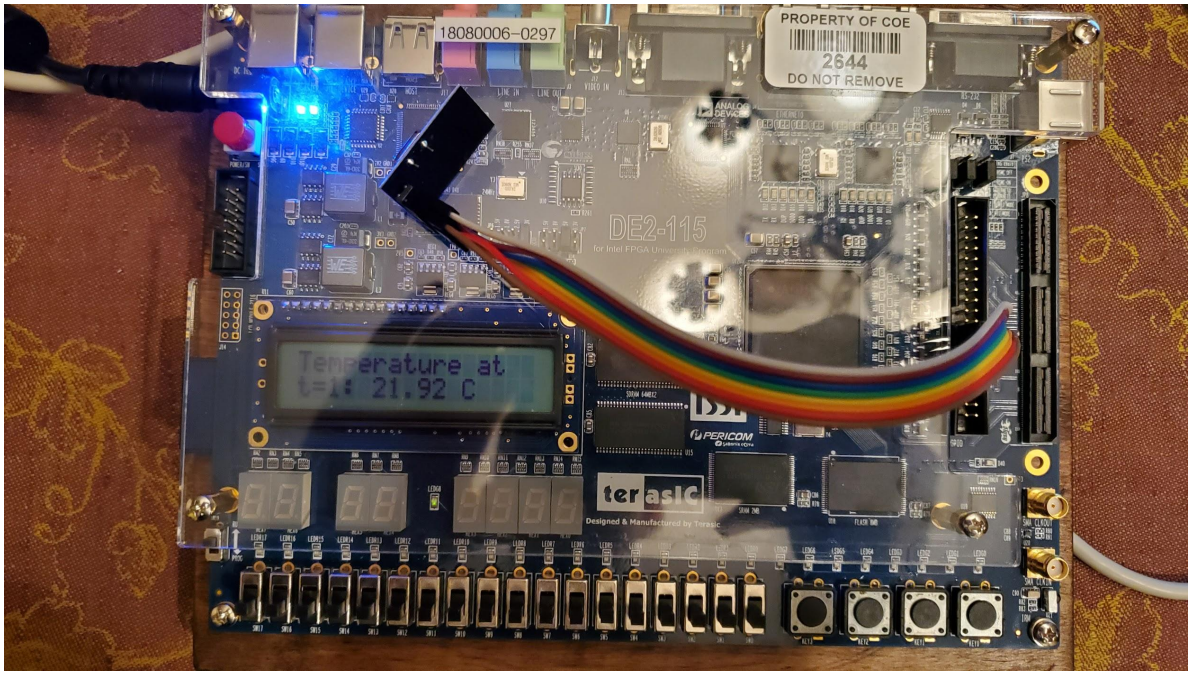

Figure 4.22: Display board showing temperature recorded at t = 1

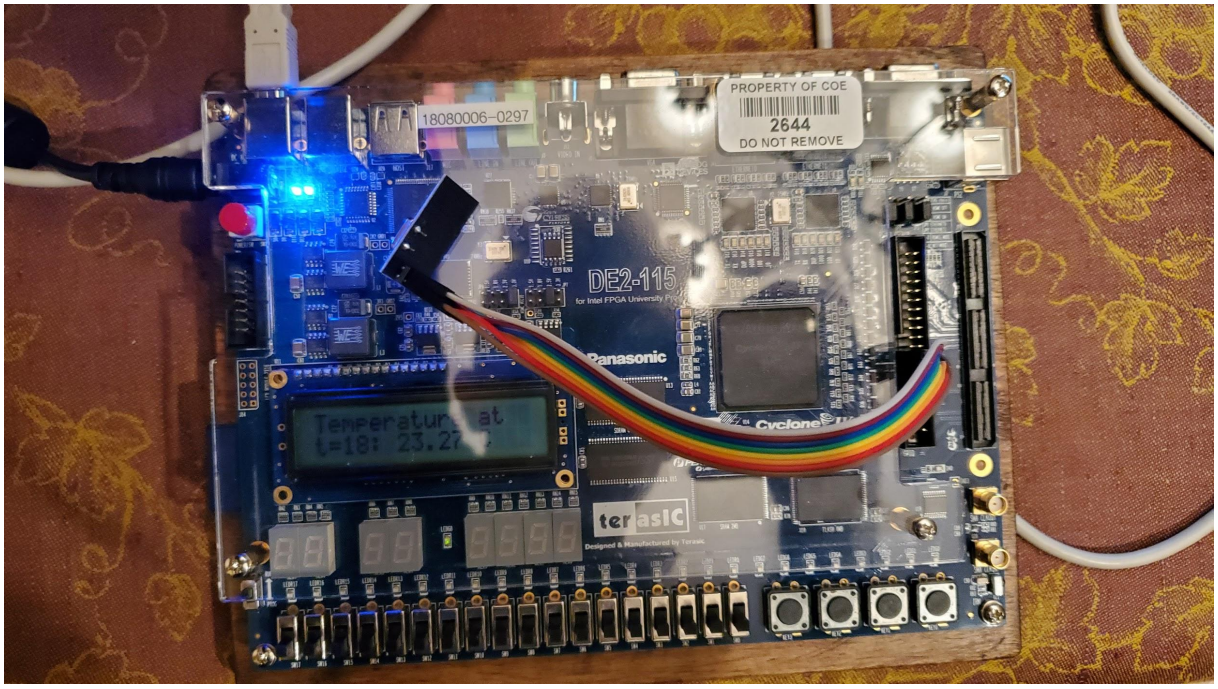

Figure 4.23: Display board showing temperature recorded at t = 18

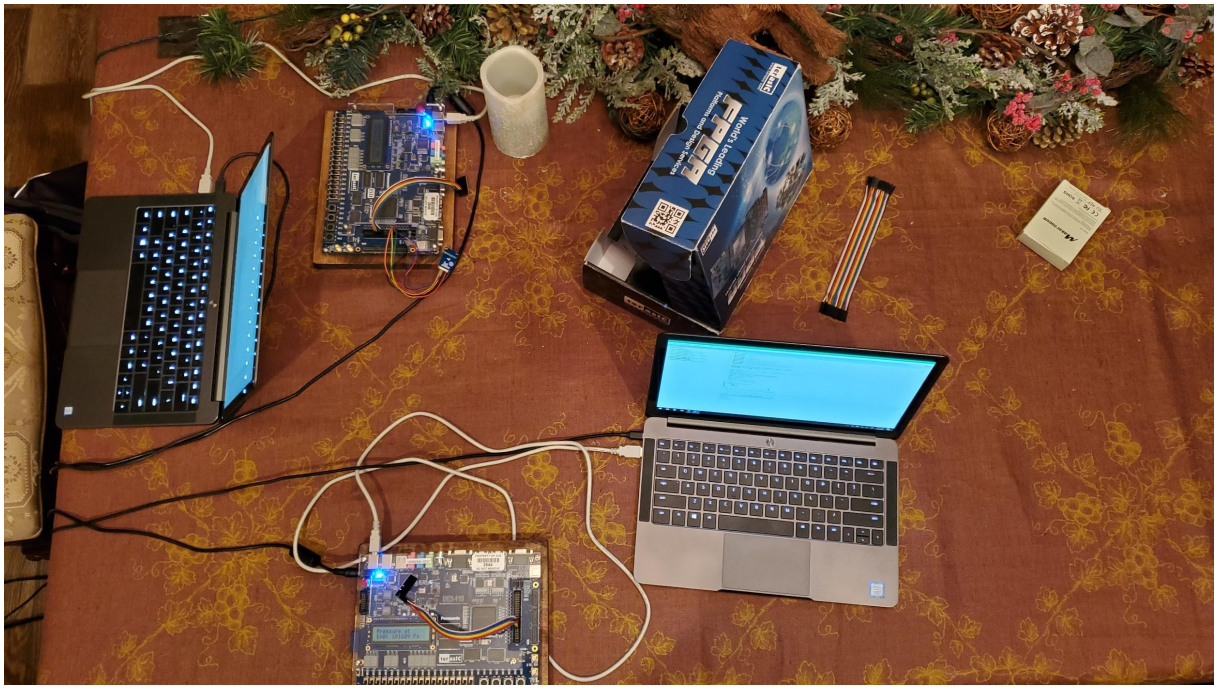

Figure 4.24: Overall setup with the collector board shown towards the top and the display board shown at the bottom

Note that not all of the source code is being shown in this section. Refer to the Appendices A through D to see the complete Verilog and C source code.

4.2 Analysis and Discussion of Results

From the results, it can be seen that all parts of both systems were successfully implemented to create a working greenhouse monitoring system. In Figure 4.12 to 4.14, three console outputs from the data collector are shown. These values are retrieved by first reading the raw data from the sensor, then converting it into floating point values. These values are initially retrieved using a function which allows the board to communicate with the sensor using the SPI interface. This function can be seen in Figure 4.1, and it essentially works by manually driving the CSN, SCK, and MOSI outputs to communicate and receive the bytes it needs. The first line of the output shows the raw ADC data that was retrieved from the sensor (ignoring the first bytes, from right to left it is: 3 bytes of pressure data, 3 bytes of temperature data, and 2 bytes of humidity data). It should be noted that although the pressure and temperature data are read as 3 bytes (24 bits), the pressure and temperature data are only represented by 20 bits (refer to Table 2.4). The next three lines show the environmental data after it has been passed through the compensation functions. These compensation functions (one is seen in Figure 4.2) are obtained through the BME280 datasheet. Their purpose is to take in the raw ADC data and run a compensation function using the calibration data from the calibration registers (refer to Table 2.4). The calibration data is read during the set up phase of the collector system and stored into variables of the format: dig_T#, dig_P#, and dig_H#. The output of the compensation functions is not the exact format that is desired. For example, if the output was “2105”, to get the actual output, one would need to multiply by 0.01 to get “21.05”. For pressure and humidity their results are in a fixed point number (Q24.8 for pressure and Q22.10 for humidity), so their results need to casted to a float and divided by 256 (2^8) and 1024 (2^10) respectively. The line after the compensated sensor data in the console output is the data that is going to be sent to the transceiver TX FIFO. The sensor data is split into 12 bytes (4 for each of the metrics). It should be noted that a “union” struct is used to turn the float values of the pressure and humidity into an integer that represents the floating point format of the float value. This is done so that the float point value can be sent through the transmitter in a readable format as 4 bytes. These are sent to the transmitter using a function similar to the one shown in Figure 4.1, just configured to use the transceiver I/O instead, as the transceiver also communicates through SPI. Finally, the last line in the console represents the status register of the transceiver. This was used to gauge whether or not the data was properly being transmitted. In the output, 2e means that the TX data has been sent and an ACK has been received.

On the receiving end, upon an interrupt, the transceiver interrupt handler shown in Figure 4.11 is called. This handler reconstructs the received data from bytes back into float values, and then stores it to the SD card by calling the corresponding function shown in Figure 4.7. When the push buttons are pressed and the data needs to be updated accordingly, the interrupt handler shown in Figure 4.9 is called, which handles the logic of changing the display or index variables according to button pressed, and then updates the display by first reading the SD card by calling

the function shown in Figure 4.8, and then displaying the retrieved data using the function shown in Figure 4.10. The data received at these three points in time, shown in the collector end in Figures 4.12 to 4.14, corresponds exactly to what has been displayed on the display board in Figures 4.19 to 4.23. This indicates that the process of wirelessly communicating the data, storing it, retrieving it, and then displaying it is all executing as it was originally intended.

5. CONCLUSION

This project sought to provide a method of monitoring a greenhouse using two embedded systems that can wirelessly communicate with each other. The system that was developed in this project had one board that was able to read three metrics: temperature, pressure, and humidity. This board would then transmit the data every 5 seconds to the display board. This board would be used by whoever would need to monitor the system. This board can either display the current data or store the data into an SD card. This would allow the person monitoring the greenhouse data to also see previous data and perhaps run an analysis of the data and create predictions for their future plant yield. This project provided valuable experience in developing embedded systems with external sensors. Additionally, it utilized concepts that have been discussed throughout the ECE 178 course, such as: SPI, SD cards, interrupts, timers, and PIOs.

6. REFERENCES

[1] Nordic Semiconductor, “nRF24L01+ Single Chip 2.4GHz Transceiver Preliminary Product specification v1.0, 2008”. [Online]. Available: https://www.sparkfun.com/datasheets/Components/SMD/nRF24L01Pluss_Preliminary_Product_Specification_v1_0.pdf

[2] Bosch Sensortec, “Final data sheet BME280 Combined humidity and pressure sensor”, 2014. [Online]. Available: https://www.waveshare.com/w/upload/9/91/BME280_datasheet.pdf

[3] Altera, “16x2 Character Display for Altera DE2-Series Boards”, 2011. [Online]. Available: https://wiki.eecs.yorku.ca/course_archive/2014-15/W/3215/_media/laboratory:character_lcd.pdf

[4] Intel, “Intel FPGA University Program Secure Data Card IP Core”. [Online]. Available: https://ftp.intel.com/Pub/fpgaup/pub/Intel_Material/15.1/University_Program_IP_Cores/Memory/SD_Card_Interface_for_SoPC_Builder.pdf

[5] Intel, “Altera DE2-115 User Manual”, 2013. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf. [Accessed: 22-Nov-2021]

Appendix A

Top Verilog Module of Collector System

```
module final_project (CLOCK_50, KEY, SW, LEDR, LEDG, HEX0, HEX1, SDRAM_ADDR,
SDRAM_BA, SDRAM_CAS_N, SDRAM_CKE, SDRAM_CS_N, SDRAM_DQ, SDRAM_DQM, SDRAM_RAS_N,
SDRAM_WE_N, SDRAM_CLK,BME_CSN, BME_IRQ, BME_MISO, BME_MOSI, BME_SCK,NRF_CE,
NRF_CSN, NRF_IRQ, NRF_MISO, NRF_MOSI, NRF_SCK);

      input CLOCK_50;
      input [31:0] SW;
      input  [3:0] KEY;
      output [31:0] LEDR;
      output  [7:0] LEDG;
      output [31:0] HEX0;
      output [31:0] HEX1;
      output [11:0] SDRAM_ADDR;
      output [1:0]  SDRAM_BA;
      output        SDRAM_CAS_N;
      output        SDRAM_CKE;
      output        SDRAM_CS_N;
      inout  [31:0] SDRAM_DQ;
      output [3:0]  SDRAM_DQM;
      output        SDRAM_RAS_N;
      output        SDRAM_WE_N;
      output        SDRAM_CLK;

      output BME_CSN;
      input BME_IRQ;
      output BME_MOSI;
      output BME_SCK;
      input BME_MISO;

      output        NRF_CE;
      output        NRF_CSN;
      input               NRF_IRQ;
      input               NRF_MISO;
      output        NRF_MOSI;
      output        NRF_SCK;



      nios_system NiosII (
            .clk_clk(CLOCK_50),
            .reset_reset(),
            .switches_export(SW),
            .keys_export(KEY),
            .leds_r_export(LEDR),
            .leds_g_export(LEDG),
            .seven_seg_0_export(HEX0),
            .seven_seg_1_export(HEX1),
            .sdram_addr(SDRAM_ADDR),
            .sdram_ba(SDRAM_BA),
            .sdram_cas_n(SDRAM_CAS_N),
            .sdram_cke(SDRAM_CKE),
```

```
            .sdram_cs_n(SDRAM_CS_N),
            .sdram_dq(SDRAM_DQ),
            .sdram_dqm(SDRAM_DQM),
            .sdram_ras_n(SDRAM_RAS_N),
            .sdram_we_n(SDRAM_WE_N),
            .sdram_clk_clk(SDRAM_CLK),
            .bme_csn_export(BME_CSN),
            .bme_irq_export(BME_IRQ),
            .bme_miso_export(BME_MISO),
            .bme_mosi_export(BME_MOSI),
            .bme_sck_export(BME_SCK),
            .nrf_ce_export(NRF_CE),
            .nrf_csn_export(NRF_CSN),
            .nrf_irq_export(NRF_IRQ),
            .nrf_miso_export(NRF_MISO),
            .nrf_mosi_export(NRF_MOSI),
            .nrf_sck_export(NRF_SCK),
      );
endmodule
```

Appendix B

Top Verilog Module of Display System

```
module final_project (CLOCK_50, KEY, SW, LEDR, LEDG, HEX0, HEX1, SDRAM_ADDR,
SDRAM_BA, SDRAM_CAS_N, SDRAM_CKE, SDRAM_CS_N, SDRAM_DQ, SDRAM_DQM, SDRAM_RAS_N,
SDRAM_WE_N, SDRAM_CLK, NRF_CE, NRF_CSN, NRF_IRQ, NRF_MISO, NRF_MOSI, NRF_SCK,
SD_CMD, SD_DAT, SD_DAT3, SD_CLK, RESET, LCD_DATA, LCD_ON, LCD_EN, LCD_RS,
LCD_RW);

	input CLOCK_50;
	input [31:0] SW;
	input  [3:0] KEY;
	output [31:0] LEDR;
	output  [7:0] LEDG;
	output [31:0] HEX0;
	output [31:0] HEX1;
	output [11:0] SDRAM_ADDR;
	output [1:0]  SDRAM_BA;
	output        SDRAM_CAS_N;
	output        SDRAM_CKE;
	output        SDRAM_CS_N;
	inout  [31:0] SDRAM_DQ;
	output [3:0]  SDRAM_DQM;
	output        SDRAM_RAS_N;
	output        SDRAM_WE_N;
	output        SDRAM_CLK;
	output        NRF_CE;
	output        NRF_CSN;
	input               NRF_IRQ;
	input               NRF_MISO;
	output        NRF_MOSI;
	output        NRF_SCK;
	inout               SD_CMD;
	inout               SD_DAT;
	inout               SD_DAT3;
	output        SD_CLK;
	input               RESET;
	inout  [7:0]  LCD_DATA;
	output        LCD_ON;
	output        LCD_EN;
	output        LCD_RS;
	output        LCD_RW;

	nios_system NiosII (
		.clk_clk(CLOCK_50),
		.reset_reset(RESET),
		.switches_export(SW),
		.keys_export(KEY),
		.leds_r_export(LEDR),
		.leds_g_export(LEDG),
		.seven_seg_0_export(HEX0),
		.seven_seg_1_export(HEX1),
		.sdram_addr(SDRAM_ADDR),
		.sdram_ba(SDRAM_BA),
		.sdram_cas_n(SDRAM_CAS_N),
```

```
                .sdram_cke(SDRAM_CKE),
                .sdram_cs_n(SDRAM_CS_N),
                .sdram_dq(SDRAM_DQ),
                .sdram_dqm(SDRAM_DQM),
                .sdram_ras_n(SDRAM_RAS_N),
                .sdram_we_n(SDRAM_WE_N),
                .sdram_clk_clk(SDRAM_CLK),
                .nrf_ce_export(NRF_CE),
                .nrf_csn_export(NRF_CSN),
                .nrf_irq_export(NRF_IRQ),
                .nrf_miso_export(NRF_MISO),
                .nrf_mosi_export(NRF_MOSI),
                .nrf_sck_export(NRF_SCK),
                .sd_card_b_SD_cmd(SD_CMD),
                .sd_card_b_SD_dat(SD_DAT),
                .sd_card_b_SD_dat3(SD_DAT3),
                .sd_card_o_SD_clock(SD_CLK),
                .lcd_DATA(LCD_DATA),
                .lcd_ON(LCD_ON),
                .lcd_BLON(),
                .lcd_EN(LCD_EN),
                .lcd_RS(LCD_RS),
                .lcd_RW(LCD_RW));

endmodule
```

Appendix C

C Source Code for Data Collector System

```
#include <stdio.h>
#include <stdint.h>
#include <io.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "sys/alt_irq.h"

// ****************************************************************
// Variables used for calibration and compensating sensor data
long signed int t_fine;
unsigned short dig_T1 = 0;
signed short dig_T2 = 0;
signed short dig_T3 = 0;

unsigned short dig_P1 = -1;
signed short dig_P2 = 0;
signed short dig_P3 = 0;
signed short dig_P4 = 0;
signed short dig_P5 = 0;
signed short dig_P6 = 0;
signed short dig_P7 = 0;
signed short dig_P8 = 0;
signed short dig_P9 = 0;

unsigned char dig_H1 = 0;
signed short dig_H2 = 0;
unsigned char dig_H3 = 0;
signed short dig_H4 = 0;
signed short dig_H5 = 0;
signed char dig_H6 = 0;
// ****************************************************************

// SPI input and output bytes
uint8_t test_input[20];
uint8_t test_output[20];

// PERIOD global values
int PERIOD_INIT = 250;
int PERIOD = 5000;
// SETUP state flag
int SETUP = 0;

// Sensor result data
long signed int t_result = 0;
long unsigned int p_result = 0;
float p_result_float = 0;
long unsigned int h_result = 0;
float h_result_float = 0;

// Convert float into IEEE754 format to be sent
union float_by_bit {
      int i;
```

```
      float f;
};

// Compensate ADC temperature input function (from data sheet)
long signed int BME280_compensate_T_int32(long signed int adc_T)
{
    long signed int var1, var2, T;
    var1 = ((((adc_T>>3) - ((long signed int)dig_T1<<1))) * ((long signed
int)dig_T2)) >> 11;
    var2 = (((((adc_T>>4) - ((long signed int)dig_T1)) * ((adc_T>>4) - ((long
signed int)dig_T1))) >> 12) * ((long signed int)dig_T3)) >> 14;
    t_fine = var1 + var2;
    T = (t_fine * 5 + 128) >> 8;
    return T;
}

// Compensate ADC pressure input from data sheet
long unsigned int BME280_compensate_P_int64(long unsigned int adc_P)
{
      long long signed int var1, var2, p;
      var1 = ((long long signed int)t_fine) - 128000;
      var2 = var1 * var1 * (long long signed int)dig_P6;
      var2 = var2 + ((var1*(long long signed int)dig_P5)<<17);
      var2 = var2 + (((long long signed int)dig_P4)<<35);
      var1 = ((var1 * var1 * (long long signed int)dig_P3)>>8) + ((var1 * (long
long signed int)dig_P2)<<12);
      var1 = (((((long long signed int)1)<<47)+var1))*((long long signed
int)dig_P1)>>33;
      if (var1 == 0)
      {
            return 0; // avoid exception caused by division by zero
      }
      p = 1048576-adc_P;
      p = (((p<<31)-var2)*3125)/var1;
      var1 = (((long long signed int)dig_P9) * (p>>13) * (p>>13)) >> 25;
      var2 = (((long long signed int)dig_P8) * p) >> 19;
      p = ((p + var1 + var2) >> 8) + (((long long signed int)dig_P7)<<4);
      return (long unsigned int)p;
}

// Compensate ADC humidity input from data sheet
long unsigned int bme280_compensate_H_int32(long signed int adc_H)
{
      long signed int v_x1_u32r;
      v_x1_u32r = (t_fine - ((long signed int)76800));
      v_x1_u32r = (((((adc_H << 14) - (((long signed int)dig_H4) << 20) -
(((long signed int)dig_H5) * v_x1_u32r)) +
      ((long signed int)16384)) >> 15) * (((((((v_x1_u32r * ((long signed
int)dig_H6)) >> 10) * (((v_x1_u32r *
      ((long signed int)dig_H3)) >> 11) + ((long signed int)32768))) >> 10) +
((long signed int)2097152)) *
      ((long signed int)dig_H2) + 8192) >> 14));
      v_x1_u32r = (v_x1_u32r - (((((v_x1_u32r >> 15) * (v_x1_u32r >> 15)) >> 7)
* ((long signed int)dig_H1)) >> 4));
      v_x1_u32r = (v_x1_u32r < 0 ? 0 : v_x1_u32r);
      v_x1_u32r = (v_x1_u32r > 419430400 ? 419430400 : v_x1_u32r);
      return (long unsigned int)(v_x1_u32r>>12);
```

```
}

// Set up BME sensor
void bme_init()
{
      IOWR(BME_SCK_BASE, 0, 0);      //Initializes clock to 0
      IOWR(BME_CSN_BASE, 0, 1);      //Initializes csn to 1
}

// SPI communication function for BME sensor
void spi_comm (uint8_t input[], uint8_t output[], int in_length, int
out_length)
{
      for(int i = 0; i < out_length; i++)  //Initializing output array to all
zeros
      {
            output[i] = 0;
      }

      IOWR(BME_CSN_BASE, 0, 0);      //CSN to 0 to begin comm
      for(int i = 0; i < ((in_length > out_length)?in_length:out_length); i++)
//Loop once for each byte of communication
      {
            for(int j = 7; j >= 0; j--)    //Loop 8 times for each byte
            {
                  IOWR(BME_SCK_BASE, 0, 0);      //Clock falling edge
                  if(i < in_length)  //Change input byte (if still inputting)
                        IOWR(BME_MOSI_BASE, 0, (input[i] & (1 << j))?1:0);
                  if(i < out_length) //Change output byte (if still outputting)
                        output[i] = output[i] | (IORD(BME_MISO_BASE, 0) << j);
                  IOWR(BME_SCK_BASE, 0, 1);      //Clock rising edge (when
input is read by transceiver
            }
      }
      IOWR(BME_SCK_BASE, 0, 0);      //Reset clock back to 0
      IOWR(BME_CSN_BASE, 0, 1);      //Reset CSN back to 1, ending communication
}

// SPI communication for NRF transmitter
void spi_comm_nrf(uint8_t input[], uint8_t output[], int in_length, int
out_length)
{
      for(int i = 0; i < out_length; i++)  //Initializing output array to all
zeros
      {
            output[i] = 0;
      }

      IOWR(NRF_CSN_BASE, 0, 0);      //CSN to 0 to begin comm
      for(int i = 0; i < ((in_length > out_length)?in_length:out_length); i++)
//Loop once for each byte of communication
      {
            for(int j = 7; j >= 0; j--)    //Loop 8 times for each byte
            {
                  IOWR(NRF_SCK_BASE, 0, 0);      //Clock falling edge
                  if(i < in_length)  //Change input byte (if still inputting)
                        IOWR(NRF_MOSI_BASE, 0, (input[i] & (1 << j))?1:0);
```

```
                  if(i < out_length) //Change output byte (if still outputting)
                        output[i] = output[i] | (IORD(NRF_MISO_BASE, 0) << j);
                  IOWR(NRF_SCK_BASE, 0, 1);      //Clock rising edge (when
input is read by transceiver
            }
      }
      IOWR(NRF_SCK_BASE, 0, 0);      //Reset clock back to 0
      IOWR(NRF_CSN_BASE, 0, 1);      //Reset CSN back to 1, ending communication
}

// Set up NRF transmitter
void nrf_init() {

      // setting CLK to 0 and setting CSN to 1
      IOWR(NRF_SCK_BASE, 0, 0);
      IOWR(NRF_CSN_BASE, 0, 1);

      // setting auto ACK
      test_input[0] = 0b00111101;
      test_input[1] = 0x6;
      spi_comm_nrf(test_input, test_output, 2, 0);

      // flushing TX FIFO
      test_input[0] = 0b11100001;
      spi_comm_nrf(test_input, test_output, 1, 0);

      // setting status register
      test_input[0] = 0x27;
      test_input[1] = 0x1E;
      spi_comm_nrf(test_input, test_output, 2, 0);

}

// Print sensor data and compensate the data
void print_array_v (int8_t a[], int size)
{
      if(size > 1)
      {
            printf("Received Sensor Data: ");
            for(int i = 0; i < size - 1; i++)
            {
                  alt_printf("%x, ", (a[i] & 0xFF));
            }
      }
      alt_printf("%x\n", (a[size - 1] & 0xFF));

      // temperature
      long signed int t = 0;
      t = (a[4] << 12) + (a[5] << 4) + a[6];
      t = t & 0x000FFFFF;
      t_result = BME280_compensate_T_int32(t);
      printf("compensated T: %g\n", t_result * 0.01);

      // pressure
      long signed int p = 0;
      p = (a[1] << 12) + (a[2] << 4) + a[3];
      p = p & 0x000FFFFF;
```

```
	p_result = BME280_compensate_P_int64(p);
	p_result_float = ((float)p_result/256);
	printf("compensated P: %g\n", p_result_float);

	// humidity
	long signed int h = 0;
	h = (a[7] << 8) + a[8];
	h = h & 0xFFFF;
	h_result = bme280_compensate_H_int32(h);
	h_result_float = ((float)h_result / 1024);
	printf("compensated H: %g\n", h_result_float);
	return;
}

// Timer interrupt handler
void timer_int_handler()
{
	if (SETUP == 1) {

		// secondary setup
		// writing to ctrl_hum
		// setting osrs_h to 100
		test_input[0] = 0x72;
		test_input[1] = 0x04;
		spi_comm(test_input, test_output, 2, 0);

		// reading calib data for temperature
		test_input[0] = 0x88;
		spi_comm(test_input, test_output, 1, 7);
		dig_T1 = (test_output[2] << 8) + (test_output[1]);
		dig_T2 = (test_output[4] << 8) + (test_output[3]);
		dig_T3 = (test_output[6] << 8) + (test_output[5]);

		// reading calib data for pressure
		test_input[0] = 0x8E;
		spi_comm(test_input, test_output, 1, 19);

		dig_P1 = ((test_output[2] << 8) + (test_output[1])) & 0xFFFF;
		dig_P2 = (test_output[4] << 8) + (test_output[3])& 0xFFFF;
		dig_P3 = ((test_output[6] << 8) + (test_output[5]))& 0xFFFF;
		dig_P4 = ((test_output[8] << 8) + (test_output[7]))& 0xFFFF;
		dig_P5 = ((test_output[10] << 8) + (test_output[9]))& 0xFFFF;
		dig_P6 = ((test_output[12] << 8) + (test_output[11]))& 0xFFFF;
		dig_P7 = ((test_output[14] << 8) + (test_output[13]))& 0xFFFF;
		dig_P8 = ((test_output[16] << 8) + (test_output[15]))& 0xFFFF;
		dig_P9 = ((test_output[18] << 8) + (test_output[17]))& 0xFFFF;


		// reading calib data for humidity
		test_input[0] = 0xA1;
		spi_comm(test_input, test_output, 1, 2);

		dig_H1 = test_output[1];
		test_input[0] = 0xE1;
		spi_comm(test_input, test_output, 1, 8);

		dig_H2 = (test_output[2] << 8) + (test_output[1])& 0xFFFF;
```

```
	dig_H3 = test_output[3];
	dig_H4 = (test_output[4] << 4) + (test_output[5] & 0x0F)& 0xFFF;
	dig_H5 = (test_output[6] << 4) + (test_output[5] & 0xF0)& 0xFFF;
	dig_H6 = test_output[7];

	SETUP = 2;
	IOWR(TIMER_0_BASE, 1, 8);
	IOWR(TIMER_0_BASE, 3, PERIOD);
	IOWR(TIMER_0_BASE, 1, 7);
} else if (SETUP == 2) {
	// tx mode
	test_input[0] = 0b00100000;
	test_input[1] = 0b00001010; // power up
	spi_comm_nrf(test_input, test_output, 2, 0);
	SETUP = 3;

	IOWR(TIMER_0_BASE, 3, 2);
	IOWR(TIMER_0_BASE, 1, 7);
} else if (SETUP == 3) {
	// get Sensor data
	IOWR(TIMER_0_BASE, 1, 8);
	test_input[0] = 0xF7;
	spi_comm(test_input, test_output, 1, 9);
	print_array_v(test_output, 9);


	IOWR(NRF_CE_BASE, 0, 1);
	int j =0;
	// delay at least 130us
	for (int i = 0; i < 10000; i++) {
		j = 1;
	}
	test_input[0] = 0b10100000;
	union float_by_bit p;
	p.f = p_result_float;
	union float_by_bit h;
	h.f = h_result_float;
	// Moving result data into bytes
	test_input[1] = (t_result >> 24) & 0xFF;
	test_input[2] = (t_result >> 16) & 0xFF;
	test_input[3] = (t_result >> 8) & 0xFF;
	test_input[4] = (t_result >> 0) & 0xFF;
	test_input[5] = (p.i >> 24) & 0xFF;
	test_input[6] = (p.i >> 16) & 0xFF;
	test_input[7] = (p.i >> 8) & 0xFF;
	test_input[8] = (p.i >> 0) & 0xFF;
	test_input[9] = (h.i >> 24) & 0xFF;
	test_input[10] = (h.i >> 16) & 0xFF;
	test_input[11] = (h.i >> 8) & 0xFF;
	test_input[12] = (h.i >> 0) & 0xFF;
	printf("Transmitted data: ");
	for (int i = 0; i < 13; i++) {
		printf("%x ", test_input[i]);
	}
	printf("\n");
	spi_comm_nrf(test_input, test_output, 13, 0);
```

```
            SETUP = 4;
            IOWR(TIMER_0_BASE, 3, 1);
            IOWR(TIMER_0_BASE, 1, 7);
      } else if (SETUP == 4) {
            IOWR(TIMER_0_BASE, 1, 8);
            IOWR(NRF_CE_BASE, 0, 0);
            IOWR(TIMER_0_BASE, 3, PERIOD);
            IOWR(TIMER_0_BASE, 1, 7);

            test_input[0] = 0xFF;
            spi_comm_nrf(test_input, test_output, 1, 1);
            printf("Status: %x\n\n", test_output[0]);

            test_input[0]=0x27;
            test_input[1]=0x2E;
            spi_comm_nrf(test_input, test_output, 2, 0);
            SETUP = 2;
      }
      IOWR(TIMER_0_BASE, 0, 0);
      return;
}

// Set up PIOs
static void pio_init()
{

      IOWR(SEVEN_SEG_0_BASE,0, -1);
      IOWR(SEVEN_SEG_1_BASE, 0, -1);
      IOWR(LEDS_R_BASE, 0, 0);
      IOWR(LEDS_G_BASE, 0, 0);
      IOWR(TIMER_0_BASE, 1, 8);
      IOWR(TIMER_0_BASE, 3, PERIOD_INIT);
      IOWR(TIMER_0_BASE, 1, 7);
      alt_irq_register(TIMER_0_IRQ, TIMER_0_BASE, timer_int_handler);

      return;
}

// Main function
int main ()
{

      // initializing PIOs
      pio_init();
      bme_init();
      nrf_init();

      // initial setup
      // writing to crtl_meas to set up osrs_t and osrs_p and set mode
      // osrs_t = 001, osrs_p = 001, mode = 11
      test_input[0] = 0x74;
      test_input[1] = 0x27;
      spi_comm(test_input, test_output, 2, 0);
      SETUP = 1;

      return 0;
}
```

Appendix D

C Source Code for Data Display System

```
#include <stdio.h>
#include <stdint.h>
#include <io.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "sys/alt_irq.h"
#include <altera_up_sd_card_avalon_interface.h>
#include "altera_up_avalon_character_lcd.h"

#define SD_ASR (SD_CARD_INTERFACE_BASE + 564*8)
#define FILE_NAME "data.txt"

int display = 0; // 0 = temp, 1 = humidity, 2 = pressure
int index = 0;
int max_index = 0;
float temperature, humidity, pressure;
int use_data_prompt = 0;
int no_response = 1;
int response;
int waiting = 0;
uint8_t nrf_input[33];
uint8_t nrf_output[33];

union float_by_bit {
      int i;
      float f;
};

alt_up_character_lcd_dev * char_lcd_dev;

//NRF_CSN_BASE, NRF_SCK_BASE, NRF_MOSI_BASE, NRF_MISO_BASE, NRF_CE_BASE,
NRF_IRQ_BASE

void spi_comm(uint8_t input[], uint8_t output[], int in_length, int out_length)
{
      for(int i = 0; i < out_length; i++)  //Initializing output array to all
zeros
      {
            output[i] = 0;
      }

      IOWR(NRF_CSN_BASE, 0, 0);      //CSN to 0 to begin comm
      for(int i = 0; i < ((in_length > out_length)?in_length:out_length); i++)
//Loop once for each byte of communication
      {
            for(int j = 7; j >= 0; j--)    //Loop 8 times for each byte
            {
                  IOWR(NRF_SCK_BASE, 0, 0);      //Clock falling edge
                  if(i < in_length)  //Change input byte (if still inputting)
                        IOWR(NRF_MOSI_BASE, 0, (input[i] & (1 << j))?1:0);
                  if(i < out_length) //Change output byte (if still outputting)
                        output[i] = output[i] | (IORD(NRF_MISO_BASE, 0) << j);
```

```
                  IOWR(NRF_SCK_BASE, 0, 1);      //Clock rising edge (when
input is read by transceiver
            }
      }
      IOWR(NRF_SCK_BASE, 0, 0);      //Reset clock back to 0
      IOWR(NRF_CSN_BASE, 0, 1);      //Reset CSN back to 1, ending communication
}

void print_array(int8_t a[], int size)
{
      if(size > 1)
      {
            for(int i = 0; i < size - 1; i++)
            {
                  alt_printf("%x, ", (a[i] & 0xFF));
            }
      }
      alt_printf("%x\n", (a[size - 1] & 0xFF));
}

void lcd_print(char string0[], char string1[])
{
      alt_up_character_lcd_init(char_lcd_dev);
      alt_up_character_lcd_string(char_lcd_dev, string0);
      alt_up_character_lcd_set_cursor_pos(char_lcd_dev, 0, 1);
      alt_up_character_lcd_string(char_lcd_dev, string1);
}

void sd_write_data()
{
      union float_by_bit n;
      max_index++;
      short int handle = alt_up_sd_card_fopen(FILE_NAME, false);
      alt_up_sd_card_write(handle, max_index);
      for(int i = 0; i < max_index * 12; i++) {alt_up_sd_card_read(handle);}
      n.f = temperature;
      for(int i = 3; i >= 0; i--) {alt_up_sd_card_write(handle, (n.i >>
(i*8)));}
      n.f = humidity;
      for(int i = 3; i >= 0; i--) {alt_up_sd_card_write(handle, (n.i >>
(i*8)));}
      n.f = pressure;
      for(int i = 3; i >= 0; i--) {alt_up_sd_card_write(handle, (n.i >>
(i*8)));}
      alt_up_sd_card_fclose(handle);

      if(index == max_index - 2)
      {
            index++;
            update_lcd();
      }

      if(max_index == 1)
            update_lcd();
}

void sd_read_data()
```

```
{
	union float_by_bit n;
	short int handle = alt_up_sd_card_fopen(FILE_NAME, false);
	alt_up_sd_card_read(handle);
	for(int i = 0; i < index * 12 + 12; i++) {alt_up_sd_card_read(handle);}
	n.i = 0;
	for(int i = 3; i >= 0; i--) {n.i = n.i | (alt_up_sd_card_read(handle) <<
(i*8));}
	temperature = n.f;
	n.i = 0;
	for(int i = 3; i >= 0; i--) {n.i = n.i | (alt_up_sd_card_read(handle) <<
(i*8));}
	humidity = n.f;
	n.i = 0;
	for(int i = 3; i >= 0; i--) {n.i = n.i | (alt_up_sd_card_read(handle) <<
(i*8));}
	pressure = n.f;
	alt_up_sd_card_fclose(handle);
}

void update_lcd()
{
	if(max_index == 0)
	{
		lcd_print("No data", "available yet.");
		return;
	}
	char *temp0 = (char*)malloc(16 * sizeof(char));
	char *temp1 = (char*)malloc(16 * sizeof(char));
	switch(display) {
		case 0:
			sprintf(temp0, "Temperature at");
			sprintf(temp1, "t=%d: %g C", index, temperature);
			lcd_print(temp0, temp1);
			break;
		case 1:
			sprintf(temp0, "Humidity at");
			sprintf(temp1, "t=%d: %g%%", index, humidity);
			lcd_print(temp0, temp1);
			break;
		case 2:
			sprintf(temp0, "Pressure at");
			sprintf(temp1, "t=%d: %g Pa", index, pressure);
			lcd_print(temp0, temp1);
			break;
	}
}

void key_int_handler()
{
	int state = IORD(KEYS_BASE, 0);
	if(!(state & 2))
	{
		//alt_printf("KEY1\n");
		char *temp = (char*)malloc(16 * sizeof(char));
		if(display == 2)
			display = 0;
```

```
			else
				display++;
			sd_read_data();
			update_lcd();
		}
		else if(!(state & 4))
		{
			//alt_printf("KEY2\n");
			if(use_data_prompt)
			{
				no_response = 0;
				response = 0;
				return;
			}
			if(index < max_index - 1)
			{
				index++;
				sd_read_data();
				update_lcd();
			}
		}
		else if(!(state & 8))
		{
			//alt_printf("KEY3\n");
			if(use_data_prompt)
			{
				no_response = 0;
				response = 1;
				return;
			}
			if(index > 0)
			{
				index--;
				sd_read_data();
				update_lcd();
			}
		}
		IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE + 3, 0x0);
}

void timer_int_handler()
{
		waiting = 0;
		IOWR(TIMER_0_BASE, 0, 0);
}

void nrf_int_handler()
{
		union float_by_bit n;

		//Reading RX payload
		nrf_input[0] = 0b01100001;
		spi_comm(nrf_input, nrf_output, 1, 13);
		//print_array(nrf_output, 13);

		unsigned int temp = 0;
		for(int i = 0; i < 4; i++)
```

```
	{
		temp = temp | (nrf_output[i + 1] << (3-i)*8);
	}
	temperature = temp * 0.01;
	//alt_printf("Temperature: %x\n", temp);
	temp = 0;
	for(int i = 0; i < 4; i++)
	{
		temp = temp | (nrf_output[i + 5] << (3-i)*8);
	}
	n.i = temp;
	pressure = n.f;
	//alt_printf("Pressure: %x\n", temp);
	temp = 0;
	for(int i = 0; i < 4; i++)
	{
		temp = temp | (nrf_output[i + 9] << (3-i)*8);
	}
	n.i = temp;
	humidity = n.f;
	//alt_printf("Humidity: %x\n", temp);

	//Resetting status register
	nrf_input[0] = 0x27;
	nrf_input[1] = 0x4E;
	spi_comm(nrf_input, nrf_output, 2, 1);

	printf("%g, %g, %g\n", temperature, pressure, humidity);

	sd_write_data();

	IOWR_ALTERA_AVALON_PIO_EDGE_CAP(NRF_IRQ_BASE + 3, 0x0);
}

static void irq_init()
{
	IOWR_ALTERA_AVALON_PIO_IRQ_MASK(KEYS_BASE, 0xF);
	IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE + 3, 0x0);
	alt_irq_register(KEYS_IRQ, KEYS_BASE + 3, key_int_handler);

	/*IOWR(TIMER_0_BASE, 1, 8);
	IOWR(TIMER_0_BASE, 0, 0);
	alt_irq_register(TIMER_0_IRQ, TIMER_0_BASE, timer_int_hander);
	IOWR(TIMER_0_BASE, 3, 250);
	IOWR(TIMER_0_BASE, 1, 7);
	alt_printf("IRQ initialized\n");*/

	return;
}

void nrf_init()
{
	IOWR(NRF_SCK_BASE, 0, 0);	//Initializes clock to 0
	IOWR(NRF_CSN_BASE, 0, 1);	//Initializes csn to 1

	//Turning on auto acknowledge
	nrf_input[0] = 0b00100001;
```

```
	nrf_input[1] = 1;
	spi_comm(nrf_input, nrf_output, 2, 1);

	//Resetting status register
	nrf_input[0] = 0x27;
	nrf_input[1] = 0x4E;
	spi_comm(nrf_input, nrf_output, 2, 1);

	//Resetting data pipe 0 byte count
	nrf_input[0] = 0x31;
	nrf_input[1] = 12;
	spi_comm(nrf_input, nrf_output, 2, 1);

	//Flushing RX FIFO
	nrf_input[0] = 0b11100010;
	spi_comm(nrf_input, nrf_output, 1, 1);

	//Enabling RX mode
	nrf_input[0] = 0b00100000;
	nrf_input[1] = 0b110010;
	spi_comm(nrf_input, nrf_output, 2, 1);

	IOWR(TIMER_0_BASE, 3, 2);
	IOWR(TIMER_0_BASE, 1, 7);
	waiting = 1;
	while(waiting) {}
	IOWR(TIMER_0_BASE, 1, 8);
	alt_printf("Standby-I mode\n");

	IOWR(NRF_CE_BASE, 0, 1);
	nrf_input[0] = 0b00100000;
	nrf_input[1] = 0b110011;
	spi_comm(nrf_input, nrf_output, 2, 1);
	alt_printf("RX mode\n");
}

void lcd_init()
{
	// open the Character LCD port
	char_lcd_dev = alt_up_character_lcd_open_dev ("/dev/character_lcd");
	if ( char_lcd_dev == NULL)
		alt_printf ("Error: could not open character LCD device\n");
	else
		alt_printf ("Opened character LCD device\n");

	/* Initialize the character display */
	alt_up_character_lcd_init (char_lcd_dev);
}

void sd_init()
{
	alt_up_sd_card_dev *device_reference = NULL;
	device_reference = alt_up_sd_card_open_dev("/dev/sd_card_interface");
	if (device_reference != NULL) {
		if (alt_up_sd_card_is_Present()) {
			alt_printf("Card connected\n");
			if (alt_up_sd_card_is_FAT16()) {
```

```
                        alt_printf("FAT16 file system detected\n");
                  } else {
                        alt_printf("Unknown file system\n");
                  }
            } else if (!alt_up_sd_card_is_Present()) {
                  lcd_print("Please insert an", "SD card.");
                  while(!alt_up_sd_card_is_Present()) {}
                  lcd_print("", "");
                  alt_printf("Card connected\n");
                  if (alt_up_sd_card_is_FAT16()) {
                        alt_printf("FAT16 file system detected\n");
                  } else {
                        alt_printf("Unknown file system\n");
                  }
            }
      }

      short int handle = alt_up_sd_card_fopen(FILE_NAME, false);
      int start = alt_up_sd_card_read(handle);
      if(start > 0 && start < 65535)
            {
                  lcd_print("Existing data", "found. Use it?");
                  use_data_prompt = 1;
                  while(no_response) {}
                  use_data_prompt = 0;
                  if(response)
                  {
                        alt_printf("Existing data used\n");
                        max_index = start;
                        index = max_index - 1;
                        alt_up_sd_card_fclose(handle);
                        sd_read_data();
                  }
                  else
                  {
                        alt_up_sd_card_fclose(handle);
                        handle = alt_up_sd_card_fopen(FILE_NAME, false);
                        alt_up_sd_card_write(handle, 0);
                        max_index = 0;
                        alt_up_sd_card_fclose(handle);
                  }
            }
            else
            {
                  alt_up_sd_card_fclose(handle);
                  handle = alt_up_sd_card_fopen(FILE_NAME, false);
                  alt_up_sd_card_write(handle, 0);
                  max_index = 0;
                  alt_up_sd_card_fclose(handle);
            }
            update_lcd();
}

int main()
{

      IOWR(SEVEN_SEG_0_BASE, 0, 0xFFFFFFFF);
```

```
	IOWR(SEVEN_SEG_1_BASE, 0, 0xFFFFFFFF);
	IOWR(LEDS_R_BASE, 0, 0);
	IOWR(LEDS_G_BASE, 0, 0);

	IOWR(TIMER_0_BASE, 1, 8);
	IOWR(TIMER_0_BASE, 0, 0);
	alt_irq_register(TIMER_0_IRQ, TIMER_0_BASE, timer_int_handler);

	IOWR_ALTERA_AVALON_PIO_IRQ_MASK(NRF_IRQ_BASE, 1);
	IOWR_ALTERA_AVALON_PIO_EDGE_CAP(NRF_IRQ_BASE + 3, 0);
	alt_irq_register(NRF_IRQ_IRQ, NRF_IRQ_BASE + 3, nrf_int_handler);

	alt_printf("Waiting for nrf startup\n");
	IOWR(TIMER_0_BASE, 3, 150);
	IOWR(TIMER_0_BASE, 1, 7);
	waiting = 1;
	while(waiting){}
	alt_printf("Nrf started\n");

	irq_init();
	nrf_init();
	lcd_init();
	sd_init();
}
```