Chapter 1: Overview

1A. Introduction

This is the “Theory of Design” document for the Plant Monitoring System. This document describes the entire system theory and design in great technical detail. In this document one will find theoretical descriptive text, figures, schematics, block diagrams, program code and much more. This is all provided so that a technical professional can, not only build this system from scratch but also modify it as they see fit.

This product is a plant monitoring system, which means that it constantly takes different measurements in order to make sure that the environment is compatible with the optimum environmental needs of any specific plant.

Note: *This device does not contain a database of optimal environmental needs for specific plants. This means that it is up to the user to properly define the environmental needs of the plant, that they wish to be monitored. Once the user has defined the specific needs for their plant, this product will monitor the environmental conditions.*

If the conditions that the user had defined are not met, then the product will set off an alarm to notify the user of what conditions are not being met. This plant monitor is part of the anti-noise pollution line of products. This means that instead of ringing a buzzer for the alarm, this product makes the display blink in order to get the attention of the user. The LCD display in the front panel will show the user the measurement values that triggered the alarm to go off, as well as the time that those measurements were taken.

This product uses the ATmega128 microcontroller, of the AVR 8-bit family. Our microcontroller was programed in IAR Embedded Workbench Integrated Design Environment (IDE). This is important to note because, as will be discussed later, some IAR specific functions were used in the code. We are able to monitor conditions because of the different sensor we have connected to the microcontroller.

1B. Purpose

This product was designed to monitor the environmental conditions of a plant. We are able to monitor the following conditions:

- Temperature: The user can decide to monitor temperature in Celsius (C) or Fahrenheit (F).
- Humidity:
- Carbon Dioxide (CO2) levels: CO2 levels are monitored using the unit of Parts Per Million (ppm)
- Time: we can monitor time with a resolution of 1 second.

In order to facilitate communication between the product and the user, the product has an LCD screen, and a keypad. The keypad allows the user to input data to the microcontroller and the LCD screen allows the microcontroller to output data to the user.

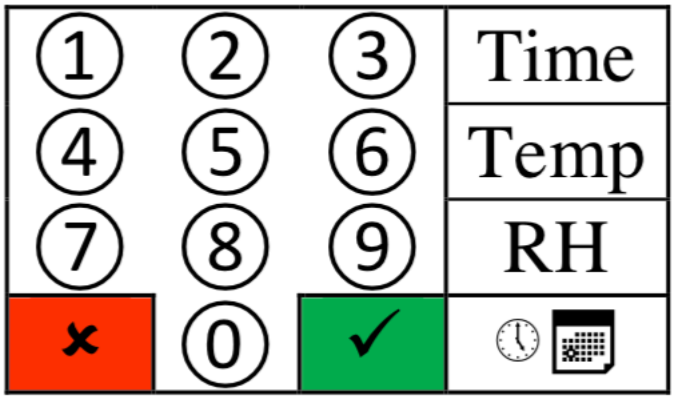
1C. Definitions

- **Sensor::=** A device that receives a signal or stimulus and responds by changing an electrical parameter or signal.
- **Stimulus::=** The quantity, property, or condition that is sensed and converted into an electrical signal.
- **Noise Pollution::=** The propagation of noise with harmful impact on the activity of human or animal life.

Chapter 2: UI and Interaction

A. Front View of Unit

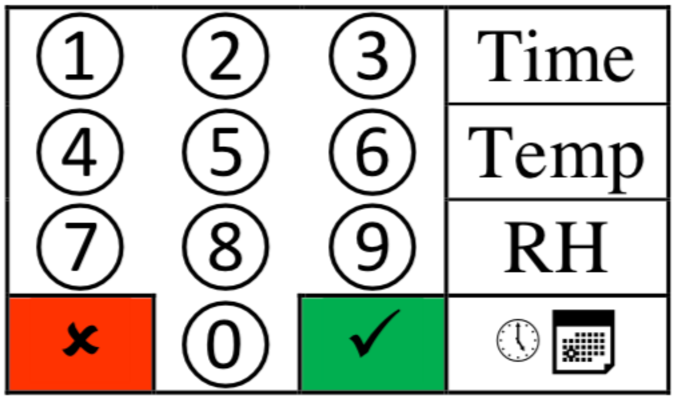
1	0	:	4	3	:	2	2	P	M		1	2	/	1	3
T	e	m	p	:		6	0	.	0	5	°	F			
R	H	:		2	3	.	0	4	%						



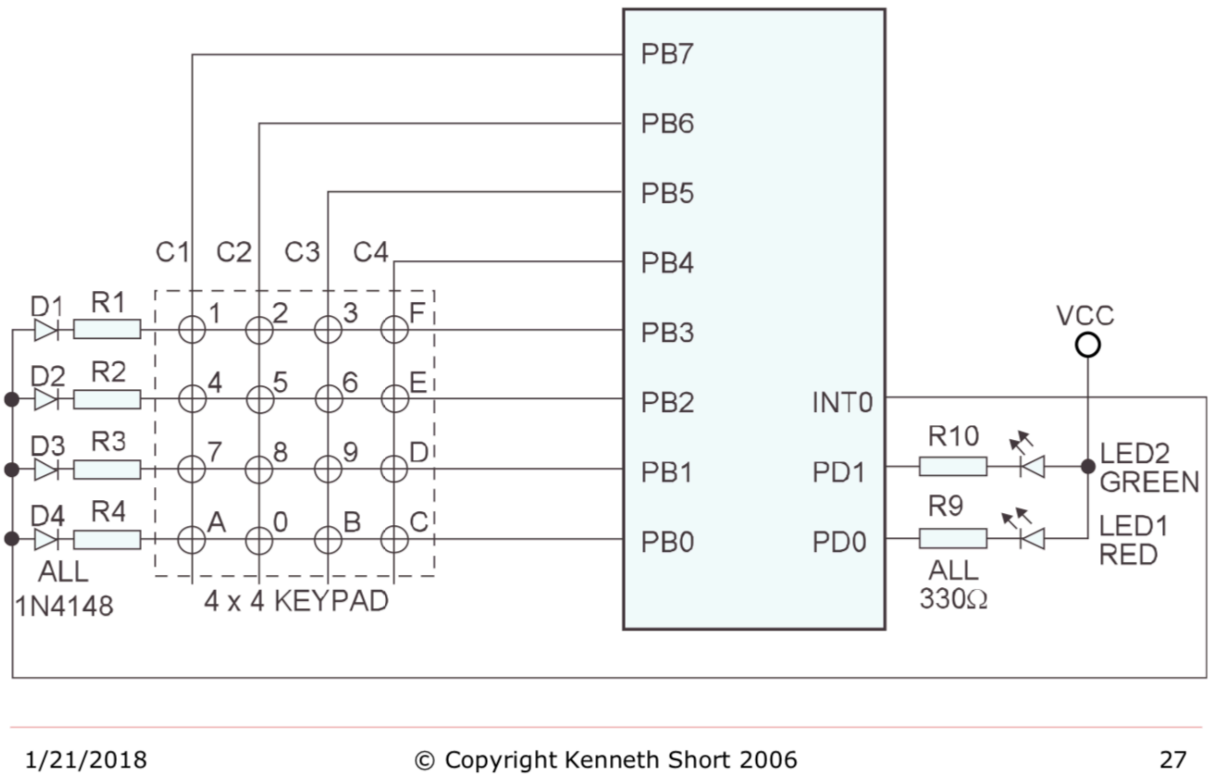
Here we can see the front panel of the product. On top we have the LCD which is composed of 3 lines of 16 characters each. Below the LCD is the 4x4 Keypad which is the main way for the user to input data into the product. Here the number buttons represent numbers and the buttons on the right most columns starting from the top represent, time, temperature, humidity and alarms.

B. Keypad

The keypad found in this product is essentially a 4x4 matrix which results in 16 buttons. It is important that we have a 4x4 matrix because it allows us to minimize the number of pins we need to use.



In a 4x4 matrix we only need to use 8 pins if we connect the keypad to the microcontroller using the matrix approach. This is because when we use an unencoded matrix the number of ports needed to support all buttons in the matrix is equal to the addition of the number of rows and the number of columns. Compare this to a linear unencoded approach, which would result in using 1 pin per button. This would mean that in order to support 16 buttons a linear unencoded approach would need 16 pins of the microcontroller, on the other hand the matrix approach only needs 8 pins to support all 16 buttons.



The way a matrix approach on a 4x4 keypad is depicted above, courtesy of Professor Kenneth Short. Out of the 8 pins we are provided at Port B we assign have of the pins to be outputs PB4,PB5,PB6,PB7 and the other half as inputs PB3, PB2, PB1, PB0. We can see that the columns are connected to the input pins and the rows are connected to output pins. The intersections on this matrix are where the buttons would be.

When a user presses down on a button one of the input pins will register a HIGH signal. Let us say that we pressed down on button 9. This would result in PB1 registering a HIGH, so we have identified the row in which the button that was pressed resides. So how do we identify the column in which the button resides. This can simply be done by only allowing one of the output ports to be HIGH at a time. This way, when we register a HIGH in the input we know what row we are in as well as which columns since only one column is allowed to be HIGH at a time. The microcontroller much faster than the user so it will never not register that the user has pressed a button been though only one column is HIGH at any given time.

A problem that was addressed in the design of this product in regard to the keypad was the phantom key problem. In the phantom key problem is as follows, let us say that buttons 6, E, and D are pressed, and that the column C3 is HIGH. The microcontroller will read PB2 and PB1 are both HIGH. This is because buttons 6 is connected to button E which is connected to D and hence both rows will drive a HIGH. However, because only column C3 is high the microcontroller will assume that buttons 6 and 9 are being pressed. Then when it goes to make C4 HIGH it will register that buttons E and D are HIGH. Resulting in the phantom key 9 being detected when it was not pressed. The way to solve this problem is to add a diode at every intersection of the buttons to prevent the current from flowing to buttons that have not been pressed.

C. LCD display

The Liquid Crystal Display (LCD) found on this product consists of 3 lines, of 16 characters per line, LCD panels and a CMOS controller driver which is mounted to the pc board that the LCD is mounted to. LCDs are among the most cost-effective ways to add a display output to an embedded system.

1	0	:	4	3	:	2	2	P	M		1	2	/	1	3
T	e	m	p	:		6	0	.	0	5	°	F			
R	H	:		2	3	.	0	4	%						

The specific LCD module found on this product is the Electronic Assembly (EA) DOGM163W-A. This module has a Sitronix ST7036 Dot Matrix Controller Driver incorporated into its pc board. This driver allows us to set up an SPI connection between the driver and the microcontroller. This SPI connection lets the microcontroller control the LCD and have it display characters without having to specifically define the dot matrix sequence of the characters. This is because the controller driver has all the dot matrix sequence for all alpha-numeric characters predefined. So when the microcontroller tells the controller driver to print “X” the controller driver search its memory for the dot matrix of “X” and sends this dot matrix sequence to the LCD which then displays the “X”.

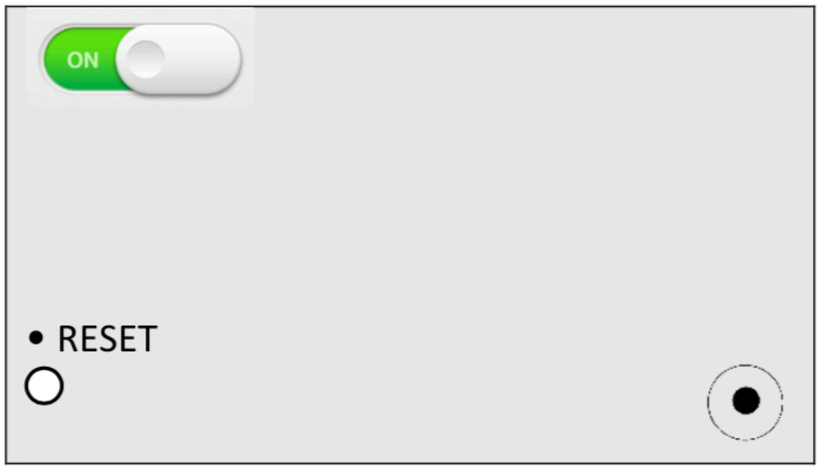
We connect the microcontroller and the controller driver by using a 10-pin header and a flat ribbon cable. This was done in order to avoid having messy wiring. We used the Serial Peripheral Interface (SPI) as the communication protocol between the microcontroller and the controller driver. SPI has the advantage of being supported by this LCD module as well as being a synchronous serial communication with full duplex. Meaning that we require a common clock between both devices and that both devices can transmit information simultaneously. Some may argue that SPI is too slow and that instead we should have used a parallel communication interface. However, one must note that the LCD module is very slow. For example, the execution time of an instructions can range from around 40 microseconds to about 1.64 milliseconds depending on the instruction. We can definitively say that using a parallel interface will not improve the speeds at which we can display new data on the LCD since the bottleneck is the ST7036 not the communication interface that we were using. However, if we had used a parallel interface then we would use more pins on the microcontroller which adds cost to the whole project. For this reason, we used SPI.

It is important to keep in mind that the LCD module must be initialized every time we power on the device. For this reason, we created a software subroutine init_lcd_dog. This subroutine essentially sets up the SPI. We select the DOG module using the Slave Select pin, we set up MOSI in order to send data to the LCD. We set our common clock to be SCK.

Once we have initialized the LCD we have to create a subroutine to update the display, called update_lcd_dog. This subroutine allows us to display the contents of our SRAM memory arrays, dsp_buff_1, dsp_buff_2, dsp_buff_3 on the LCD display. We have 3 memory arrays because we have 3 rows on our LCD, and each of these arrays is of size 16 because each line in our LCD has 16 character spaces.

With these two methods we are able to send messages to the LCD from the microcontroller. Due to the nature of both of these methods, they can be used to display any message on the LCD. This is because neither method creates the message that is to be sent they simply set up communication. This means that one could re-use these methods and add to the product more sensors and have their values displayed on the LCD by simply passing the proper buffer array which is created in the microcontroller.

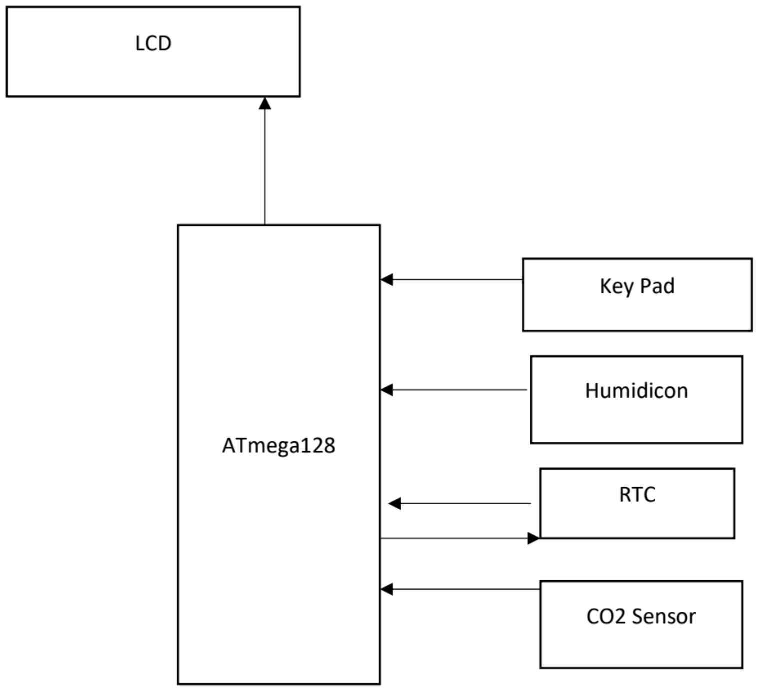
D. Back View of the Unit



Here we see the back panel of the product. It is composed of an On/Off switch at the top left corner. On the bottom left corner, we see a Reset button which is how the user can reset the system without having to power off the product. Lastly on the bottom right we have a power supply which connects to 120 V AC.

E. High Level Block Diagram of the Circuit

Below is a high-level block diagram of the circuit and the components used to measure the different environmental conditions. In later chapters we will go more in depth as to how to use these sensors. For now, this diagram will serve to provide a general overview of the whole system.



Chapter 3: Key Components

A. ATmega128a

The ATmega128a is part of the AVR 8-bit family. This means that it uses 8 bits to represent integers. It also means that it is a reduced instruction set computer (RISC), so we have a reduced set of instructions we can use when programming this microcontroller. For our purposes this knowledge does not really matter since we programmed the microcontroller using embedded-C. The IAR Embedded IDE translated and optimizes our C programs into assembly.

Some key features are the following:

- 128K Bytes In-System Reprogrammable Flash
- 4K Bytes Internal SRAM
- 4K Bytes EEPROM
- Two Expanded 16-bit Timer/Counters with Separate Prescalers, Compare Mode, and Capture Mode
- Real Time Counter with Separate Oscillator
- Dual Programmable Serial USARTs
- 53 Programmable I/O Lines

B. Temperature/Humidity sensor

The sensor on this product has the ability to measure temperature and humidity. The sensor found on this product is the Honeywell HumidIcon HIH9131-000-001 Digital Humidity/Temperature Sensor. The HIH9131-000-001 has a serial interface which makes it easy to communicate with. When communicating with it we will receive a 14-bit resolution for humidity and for temperature. This sensor is close to ideal since it has a humidity range of 0 to 100% with ±1.7% accuracy. It also has a temperature measuring range of -40 to +125°C. The sensor can compensate for temperature over the range 5 to 50°C and for that range has an accuracy of ±0.6°C. Lastly it also has a Mean Time To Failure (MTTF) of 9,312,507 hours ensuring a lifetime guarantee for this product.

When receiving data from the HIH9131-000-001 we must parse it to get the proper values of temperature and humidity. The HIH9131-000-001 will send a packet of bits, [{S(1:0), C(13:8)}, {C(7:0)}, {T(13:6)}, {T(5:0),xx}], that can be split into the following categories.

- **S(1:0)::=** status bits of packets
- **C(13:8) ::=** upper six bits of 14 bit humidity data
- **C(7:0)::=** lower 8 bits of 14 bit humidity data
- **T(13:6)::=** corrected temperature data
- **T(5:0),xx::=** remaining bits of 14 bit temperature data

Typically, the sensor is normally in power down mode, this reduces the consumption of power of the whole system and makes it energy efficient. In order to wake up the sensor we must use a measurement request (MR) command. The MR command essentially reads stale data from the HIH9131-000-001 in order to wake it up and have the HIH9131-000-001 replace the stale data with valid data. We then use a data fetch command (DF) which will request a packet of bits as portrayed above. As stated above S(1:0) are the status bits of the packet. If the S1 and S0 equal 0,0 then the data being read is valid. However, is S1 is 0 and S0 is 1 then the data being read is stale data. Data is considered stale after the master has already read the data once. We can end up reading stale data when the master polls the data quicker than the sensor can update the output buffer.

After we have parsed the packet for the 14 bits that indicate temperature and the 14 bits that indicate humidity we must use the following formulas to convert these bits into meaningful numbers.

- Humidity (%RH) = (value of 14 bits Humidity output count/ (2^14 -2))*100
- Temperature (Celsius) = (value of 14 bits Temperature output count/ (2^14 -2))*165-40

We can perform these calculations using a float type variable, however in order to be cost effective we performed these calculations using only type integer variables. This allows us to save memory when performing these calculations.

Some functions specific to this sensor are the following:

- The humidicon_config function which unselects the HIH9131-000-001 and configures it for operation with the microcontroller running a clock at 16MHz.
- The read_humidicon_byte function which reads one data byte from the HIH9131-000-001 and only when the SPI transfer is complete does it return the byte of data to the microcontroller as an unsigned char type. One may wonder how this function knows when the SPI transfer is complete. The function knows when the SPI transfer is complete because it is constantly polling the SPI status flag to see if the transfer has been finished.
- The read_humidicon function selects the HIH9131-000-001and calls the read_humidicon_byte function 4 times in order to receive the complete data packet. Each time we call the read_humidicon_byte function we assign its return value to the global unsigned ints humidicon_byte1, humidicon_byte2, humidicon_byte3, and humidicon_byte4, respectively. After we have received the 4 bytes of data the function deselects the HIH9131-000-001. The function the parses the humidicon bytes for the temperature and humidity information. The newly parsed information is then assigned to the global unsigned int humidity_raw and the global unsigned int temperature_raw respectively. Note that, both of these raw values are right justified. Once this has been done the function returns.

This is essentially how we are able to control and request data from the the HIH9131-000-001.

C. RTC

The DS1306 is a Real-Time Clock(RTC) IC by Maxim Integrated. This RTC uses an external 32.768 KHz crystal oscillator to count seconds, minutes, hours, date of the month, month, day of the week, and year with leap-year compensation that is valid up to 2100. It supports SPI for communication and has two interrupt pins that are controlled by two programmable alarms.

Alarms can be set to trigger on the combination of seconds, minutes, hours, and day of the week. An alarm's interrupt is activated when, the parameters set for that alarm match the time. The Maxim's RTC also provides a 1 Hz and a 32 KHz output pin for other ICs to use. Furthermore, it provides dual-power supply pins for primary and backup power supplies and an optional trickle charge output to the backup supply.

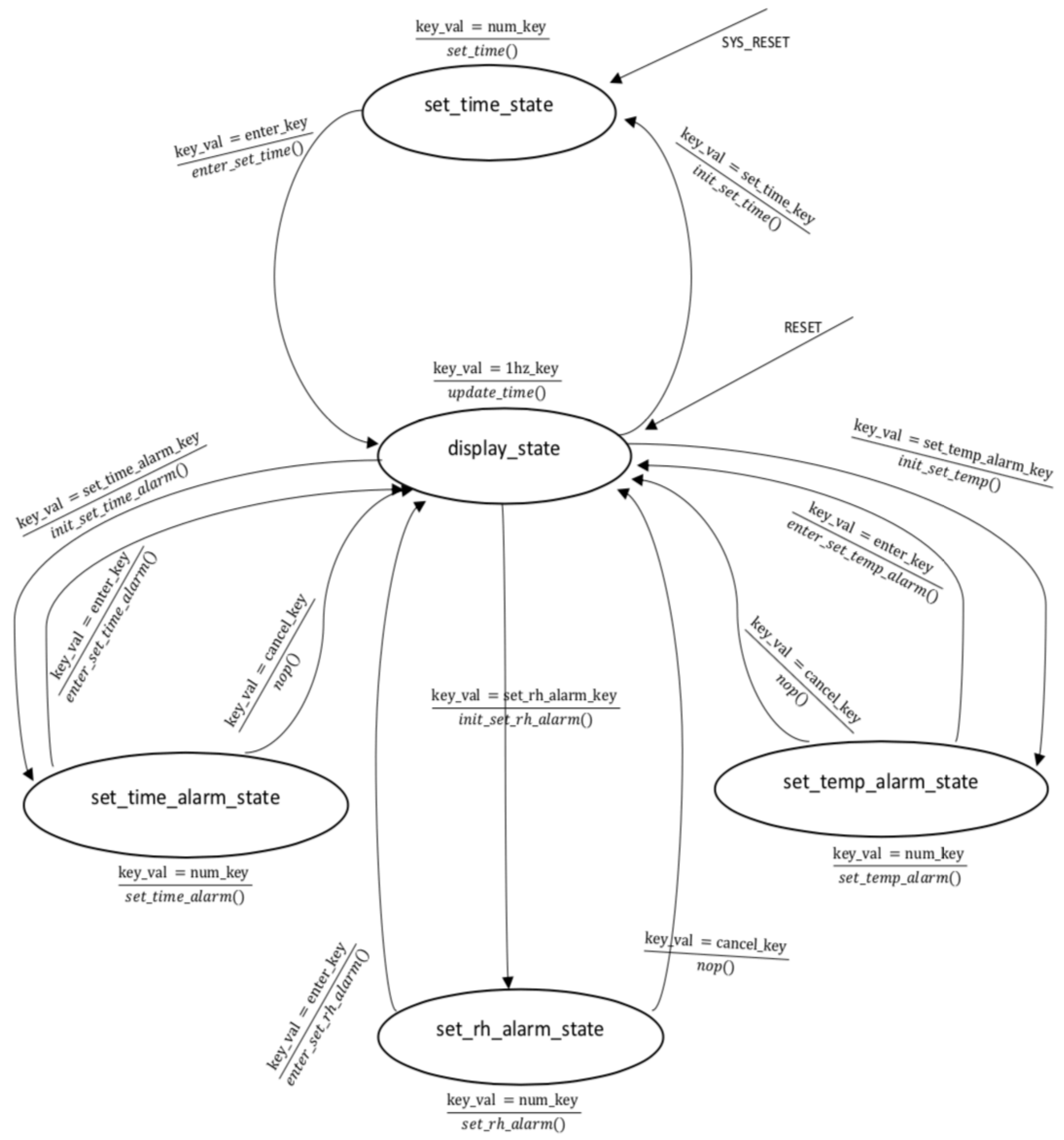
The RTC was essential to our design so that we could keep track of time without having to write a lot of software that would waste a lot of clock cycles on the ATmega. Since our ATmega is running significantly faster that 32 KHz.

D. CO_2 Sensor

This is an analog infrared CO2 sensor, with an effective measuring range from 0 to 5000ppm. This sensor integrates temperature compensation and support DAC output. It needs a 5V power supply and can output a range of analog voltages from 0V to 2V. This sensor does need a 3 minute preheating time so the user must allow the product to preheat for 3 minutes before taking any data outputted by the product seriously.

Chapter 4: System Design

A. FSM (Finite State Machine)



B. FSM explained

Most of the system is implemented as an FSM. In the diagram above can see that all of our interrupt subroutines do some minimal work to interpret the input and then call the FSM function to do all the heavy lifting.

Our FSM has a total of fives states. They are:

1) display_state: While in this state the system is idle and waiting for user input on the key pad. This state is also interrupted by the 1Hz output of our RTC. This interrupt will not change the present state of the FSM but has the FSM update the time, humidity, and temperature readings and display this on the LCD.

2) set_time_state: In this state users are prompted to input the current time and date.

3) set_time_alarm_state: In this state users are prompted to input their desired alarms for time.

4) set_temp_alarm_state: In this state users are prompted to input a range for desired temperature.

5) set_rh_alarm_state: In this state users are prompted to input a range for desired humidity.

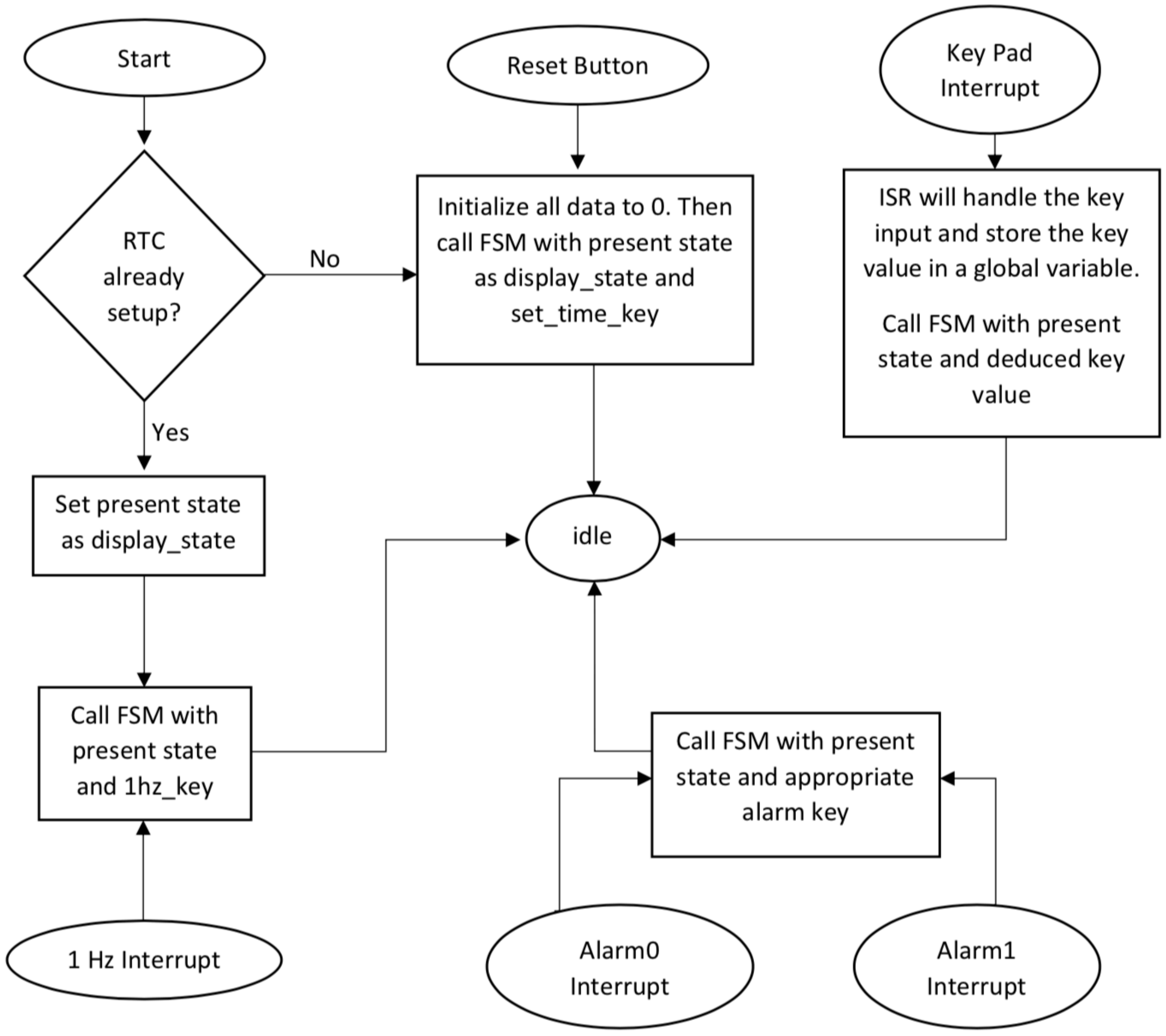
When the device is powered up for the first time ever or the user resets the device, the present state will be set to display state and the system will call the FSM as if the set_time_key was pressed. This will cause our system to go into the set_time_state. At this point the user should input appropriate values for the time and date. If the system is powered of and then turned back on, then the system will remember that it already had been initialize and therefore time and date do not need to be set again.

From the display_state we can transition to the set_time_state if a user presses the set_time_key. Here the user is asked to reset the time and date. The alarms will not be reset as they are when the system is reset or powered up for the first time.

From the display_state we can transition into any of the set_xxxx_alarm_state (in this case xxxx can be either time, temp, or rh) by pressing the appropriate key for that alarm. These three states operate in a very similar fashion as the set_time_state. Two functions oversee prompting the user for data and storing that data. To keep these functions in sync we use a static global variable so that the two functions know what data is being prompted for. Once the state has prompted for all the data, the enter_key becomes valid and can be pressed to confirm data entry. At any point the user can press the cancel_key and return to the display_state. Since we are using a global variable to sync our functions when users press the set_time_key or any of the set_xxxx_alarm_keys and initialization function is called to make sure this variable is reset. This way even if the user cancels midway and then tries to set the time or alarms again the prompts will start from the beginning again.

Chapter 5: Software

A. Flowchart of how it all works


Start
RTC already setup?
No
Yes
Set present state as display_state
Call FSM with present state and 1hz_key
1 Hz Interrupt
Reset Button
Initialize all data to 0. Then call FSM with present state as display_state and set_time_key
idle
Key Pad Interrupt
ISR will handle the key input and store the key value in a global variable.
Call FSM with present state and deduced key value
Call FSM with present state and appropriate alarm key
Alarm0 Interrupt
Alarm1 Interrupt

Chapter 6: Reference Documents