Bowman Creek

Senior Design Project

Galen Harden, Frank Kuhny, Fernando Lozano
5/7/14

TABLE OF CONTENTS

3 Introduction

As a part of an initiative presented by Gary Gilot, the Bowman Creek restoration is an important step for South Bend. The initiative seeks to improve the overall quality of the watershed particularly by lowering pollutants and creating a livable environment for aquatic life to thrive in. The goal is to reclaim the creek and to transform its heavily urbanized form into one that is not only beautiful but also environmentally safe and sound. With this is mind some of the improvements that must be made are centered in the underground and piped paths of the creek. The creek must not only be clean-flowing above ground but also throughout if the overall impact of a cleaner and healthy watershed is expected.

We seek to solve the problems of the drastic change in flow rate due to obstructions caused by backflow in the underground sewer portions of the creek. The water flow rate as it is now is not well regulated and this leads to issues such a dry creek-bed when there is a lack of rain and also flooding during heavy rain periods. The flow must be monitored so that regulation systems including valves can appropriately allow a reasonable rate of water flow to not only prevent damage in the piping but also to have the creek have a steady flow throughout all its portions. With these sensors some other issues come to light, like powering such devices throughout the piping systems, whether they are to be independently powered or tied to the grid and whether the source of power is to be renewable or not as to maintain environmental goals. Along with that came the problem of the sizing of the sensing equipment which must not obstruct the flow of water whatsoever. Also the equipment had to have some protection from water so to not have chance of failure due to damage to the equipment.

In order to monitor the water depths designed and build what will be a compact, affordable, and durable solution to the aforementioned problems. We planned to use solar panels as the primary power source to maintain the self-dependence of the sensors and batteries as a back-up power source which will help in cases of emergency as well as with maintenance. The unit will have a protective shell to prevent water damage. RF receivers and transmitters will allow communication between one sensory unit and a neighboring one. The two adjacent sensors will have their water levels compared to decide if cleanup or is necessary.

The overall system is separated into the sensing and communicating between sensors. The sensing subsystem will be controlled through the microcontroller to sense the pressure of the water through a pressure transducer and from the pressure determine the depth of the water, if there is an obstacle the pressure and water depth will drop the change due to the obstruction. This will then notify that there is there is an obstruction on the terminal. Though, this is done through the communication subsystem by comparing it to the different water level from the other sensor. As it is compared, if the difference amongst both sensors is a large discrepancy this will be displayed on the terminal for which action should be taken up afterwards. The communication system will be RF to allow the communication between sensors for a decent range within the subterranean setup of pressure sensors.

The other crucial system for the project is the powering system. The system will be powered through replaceable rechargeable batteries which will in turn be charged by the solar panel if that is the path we take. The enclosure will allow access to replace the

batteries when necessary. The power will be monitored and if the batteries are too low it will indicate need to replace or that charging will need to take place. Also the batteries will have a charge monitoring and protection circuit so as to have the batteries have longer life with more cycles.

The design overall met our teams expectations with the issues that arose in the process. Due to the expense of the pressure sensors rather than the original ultrasonic sensor, the feasibility of creating another sensor to compare the two levels did not work out so some of the intended comparison system could not make it into the final design. The sensing system was particularly more accurate than the expected ultrasonic sensor that had been tested in the past, also it took the approach of relying on the actual water level data to determine the condition that the creek was in. The other route of having the ultrasonic sensor would have been more likely to fail due to water damage from proximity or get a false reading if there was anything that may stick out above the water. The sensor was robust and strong enough to withstand way beyond the needs of the Bowman Creek which would assure use that it would be prepared for the worst and best case scenarios.

Then comes our communication the communication went through several stages in the process with just a simple RF transceiver then we attempted to find a way for the EmNet communications to be integrated into the larger Bowman Creek system, and then finally our final 900 MHz RF communication. We abandoned the EmNet path due to the fact that it would require proprietary circuits and software to enable us to use them in our system this would have been too expensive and would have not proven to be that helpful for our purposes. And so after refining which method we would use to communicate between our sensors we decided upon using the terminals to monitor our water levels. This would allow us to display our readings on a computer nearby and have room to build upon rather than having the level on an LCD because we'd want access to the data without having to enter the creek. It also ended up letting us know where the levels came from as they were forwarded along through the use of each device id.

The power system turned out a bit better than expected, it went from a crude solar to cell to sensor, to a more refined system to monitor the batteries through charging and allowed a range of voltage inputs to supply our rechargeable batteries. This allowed us to be able to charge the battery not only more efficiently but also quite safe in an array of situations that may present themselves in the process. This allows us to make sure that we can make the most out of the cells for the longest periods possible. The power draws was minimal due to our setup of awake and sleep cycles because the sensor did not need to sense always this was a portion of the design to help out the power. The batteries took quite some time to lose charge with the setup done this way, which is useful when the South Bend sun is not cooperating to charge the batteries.

Overall, even though the overall design changed the final product turned out to be a bit better than expected in performing the tasks that were planned as parts of the subsystems.

4 System Requirements

Requirement	Description	Result
General Purpose	Must be able to communicate with the user to monitor the water depths between points	Completed
User Implementation	Must be able to operate on its own once installed	Completed
Expected Life of Product	Must be able to withstand water, periods without sunlight, and long battery life	Completed
Cost	System prototype must be within $500 budget to design and produce	Completed

SUBSYSTEM REQUIREMENTS

Power		
General	-Must be able to provide rated power to each necessary subsystem (microcontroller, communication, and pressure transducer) -Must monitor battery charge while charging/discharging -Must be able to have variable voltage at input	Completed Completed Completed
Send/Receive Module		
General	-Send depth and location data to other devices in network; receive similar input from other devices	Completed
Power	-Peak power draw must be limited, should enter low power mode when not transmitting/receiving	Completed
Distance	-Must be able to transmit over sizable distances underground in order to minimize number of devices	Completed
Software	-Synchronize timing with other devices to transfer data	Completed
Pressure Sensor		

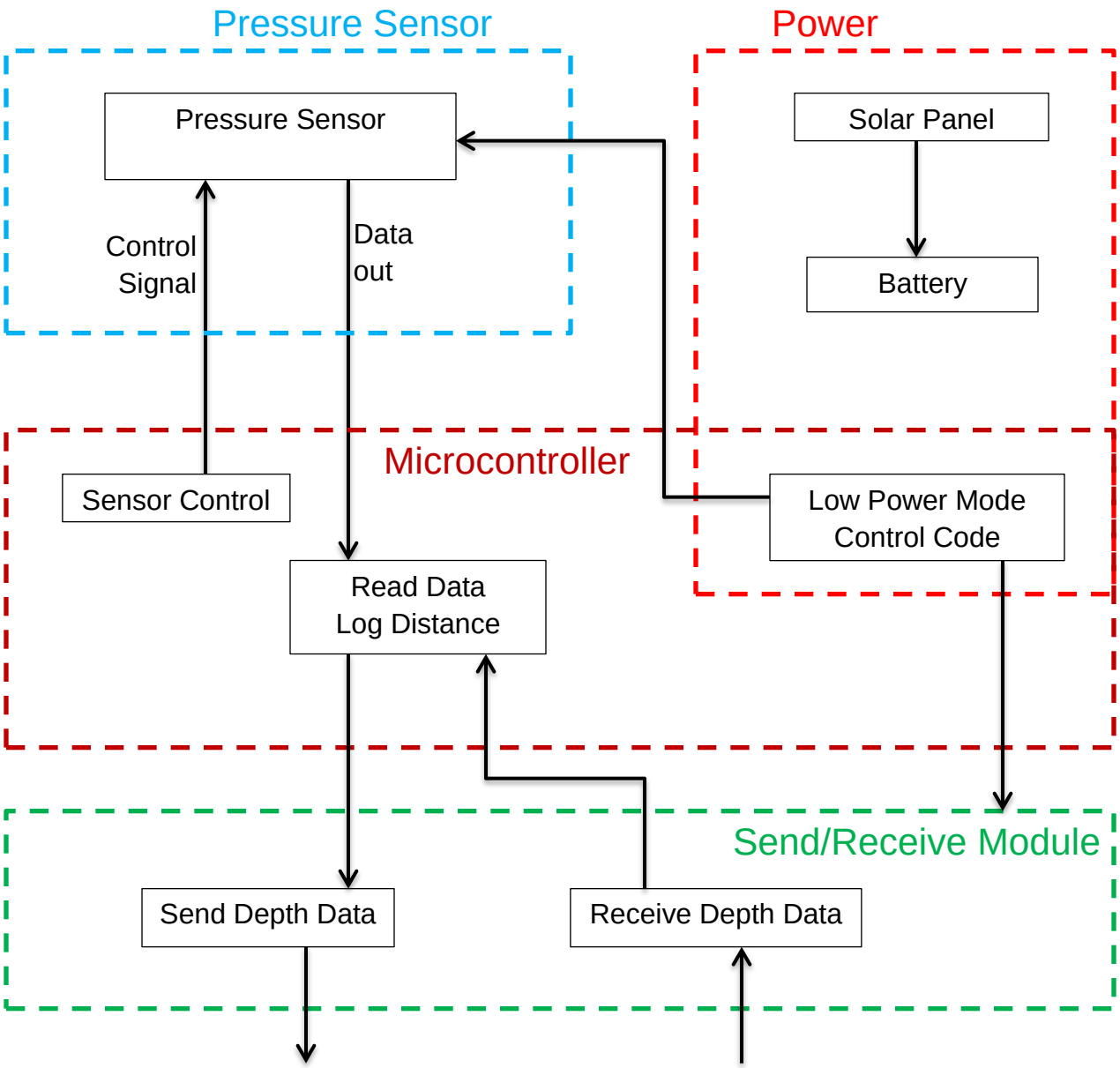
Power		
General	- Must be able to withstand worst case pressures in Bowman Creek - Must be well insulated protected from water - Must	Completed Completed

5 Detailed Project Description

5.1 System Theory of Operation

The entire system all works together from the power subsystem, then to the sensing, then to the process of communication. The charger has a 9 volt solar panel attached to the power circuit which is already programmed to monitor the charge of the two 3.7 Volt Lithium Ion cells, therefore a total of 7.4 Volts. There are two LEDs that will indicate whether the batteries themselves are charging or not. From this the voltage of the battery is dropped to 5 Volts by a 5 volt regulator which will serve as the source for the pressure transducer and the main microcontroller board. The microcontroller takes the 5 Volts and drops it to 3.3 Volts from a 3.3 volt regulator to power the PIC that is the brains of interpreting the water level and then sending measurement the to the next sensor in the network. The sensor output is an analog signal so we use an analog to digital converter and scale to determine the water depth from the pressure measured by the pressure transducer. Once this is done the programmed PIC sends the level to the next sensor along the network with a device id to identify it with the sensor in the network that it is reading. All these levels are accessed by the main microcontroller that is connected to a computer through a mini USB connection. The levels are displayed on the terminal through putty as the measurements are forward amongst the sensors in the network. The sensing only occurs every 5 minutes, but 16 seconds to demonstrate, because the water is not expected to change so quickly. This occurs through a cycle of awaking and sleeping that occurs throughout the entire process.

5.2 System Block Diagram



5.3 Detailed Design/Operation of Subsystem 1 – Power

The requirements for the power circuitry were that it must be able to provide rated power to each necessary subsystem (microcontroller, communication, and

pressure transducer), that it must monitor battery charge while charging/discharging, and that it must be able to have variable voltage at input.

It was decided that because the entire system would be enclosed and placed underground where the Bowman Creek needed to be monitored. In order to limit the need of human manipulation of the sensor it was decided to use a 9 volt solar panel to charge a system of batteries, two 3.7 volt Lithium Ion cells, a total of 7.4V at 2Ah which would then supply the power to the needed components as the pressure transducer and the main microcontroller. We decided to also allow a 2.1mm DC input for the input not only to allow testing without sunlight, but to allow another way of charging if need be. The input voltage is then used by the central component, the bq24650 Synchronous Switch-Mode Battery Charge Controller. The bq24650 is a highly integrated switch-mode battery charge controller that provides input voltage regulation, which reduces charge current when input voltage falls below a programmed level. When the input is powered by a solar panel, the input regulation loop lowers the charge current so that the solar panel can provide maximum power output. The bq24650 offers a constant-frequency synchronous PWM controller with high accuracy current and voltage regulation, charge preconditioning, charge termination, and charge status monitoring. The LEDs on the board indicate whether the batteries are being charged or not, first on charging second, second on done charging, otherwise there is a fault. The bq24650 charges the battery in three phases: pre-conditioning, constant current and constant voltage. Charge is terminated when the current reaches 1/10 of the fast charge rate. The pre-charge timer is fixed at 30 minutes. The bq24650 automatically restarts the charge cycle if the battery voltage falls below an internal threshold and enters a low quiescent current sleep mode when the input voltage falls below the battery voltage. There also exists an internal battery detection logic. Once the device has powered up, a 6-mA discharge current is applied to the SRN terminal. If the battery voltage falls below the LOWV threshold within 1 second, the discharge source is turned off, and the charger is turned on at low charge current. If the battery voltage gets up above the recharge threshold within 500ms, there is no battery present and the cycle restarts. If either the 500ms or 1 second timer time out before the respective thresholds are hit, a battery is detected and a charge cycle is initiated. We set the charge current with a current sensing resistor of 20mΩ to have a charge current of 2A. The bq24650, then has a buck converter to drop the output to our desired voltage of 7.4 Volts.

From the charging system we charge our 7.4 V battery, here we have pins to check the voltage while testing as well as to make sure the voltage is around this value to make sure everything is working according to expectations. From this the voltage of the battery is dropped to 5 Volts by a 5 volt regulator which will serve as the source for the pressure transducer and the main microcontroller board this is connected via crimp pins to a 2.1 mm DC input jack on the main board. The microcontroller takes the 5 Volts and drops it to 3.3 Volts from a 3.3 volt regulator to power the PIC that is the brains of interpreting the water level and then sending measurement the to the next sensor in the network.

To test the functionality of the charging system took various readings with a voltmeter across most of the components to be assured the entire system was working as in the input voltage, that output for the battery and the output to the board. This was definitely an extensive process of checking components and fixing any errors that

existed. The charging was checked after the batteries were drained with a low load for a while to see if charging was occurring or not even besides seeing the LEDs.

5.4 Detailed Design/Operation of Subsystem 2 – Wireless Network

Subsystem requirements: Send/Receive Module	
General	Send depth and location data to other devices in network; receive similar input from other devices
Power	Peak power draw must be limited, should enter low power mode when not transmitting/receiving
Distance	Must be able to transmit over sizable distances underground in order to minimize number of devices
Software	Synchronize timing with other devices to transfer data

The communication system was largely successful in fulfilling the requirements that were set out for it. Each device is capable of sending the depth of water that it measures to the other devices in the system, and the other devices are able to receive and interpret the sent data. Peak power draw was set to the maximum transmit power allowed by the transceiver for maximum range, but this power draw is not excessive (we were unable to measure the power draw of the transceiver because of the integrated nature of the device, but the current specifications are listed in the attached datasheet). Additionally, the transceiver is put to sleep by the software when the microcontroller goes to sleep, significantly lowering the power draw. The devices will transmit over a reasonable (but not impressive) distance of about 200 feet in buildings, which we thought would be comparable to their performance in the underground sections of Bowman Creek. Alternatively, if the devices can be mounted such that the link between them has few obstacles and is relatively line of sight, their transmit distance will be increased significantly. Finally, the devices are able to synchronize their sleep cycles in order to allow extended periods of time without powering the wireless receiver, greatly increasing the power efficiency of the transmit/receive module.

Subsystem Design and Operation

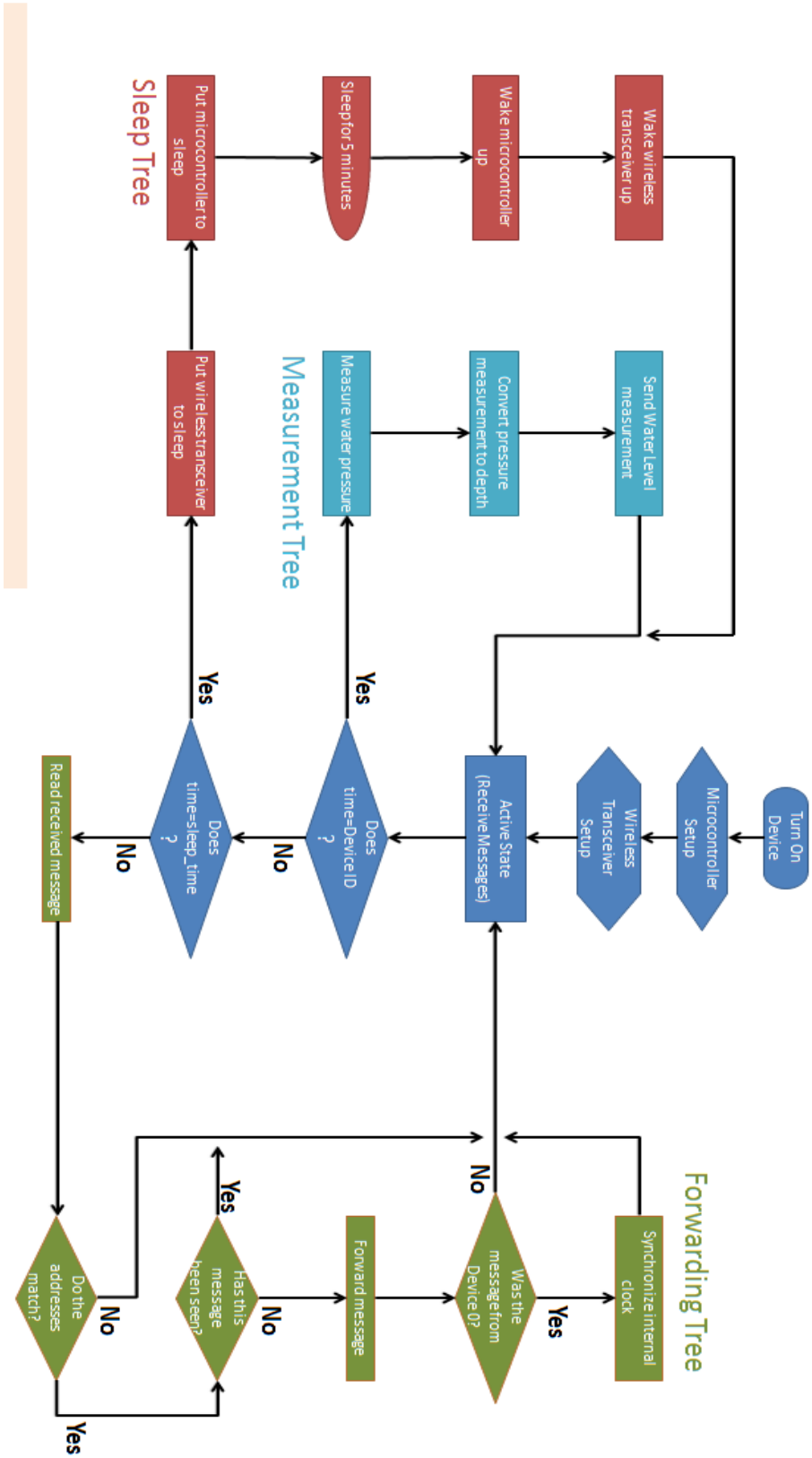
Each of the devices contains a wireless daughterboard, which is connected to the main device board and microcontroller by a 2x5 right angle connector. The wireless daughterboard is designed around the AT86RF212, a 700/800/900 MHz band wireless transceiver produced by Atmel. We chose to use the 900 MHz band rather than the more common 2.4 GHz band for wireless communication in order to take advantage of the greater range of communication that is allows, and to maintain a high level of similarity with the EmNet system upon which this system is based. The 2.4GHz band allows higher data rates, but a high data rate was not critical in this application, and would have required extra power, which was at a premium in our design. Communication between the microcontroller and the wireless daughterboard is by SPI, with additional pins for reset, sleep, and interrupt flagging. The schematic and board design are included in the appendix.

The wireless communication in the network is structured according to the standards set out in IEE 802.15.4. The microcontroller initiates a write to the ATRF212 frame buffer, where it then writes the message to be sent. Upon finishing this write, the microcontroller sends the transmit command, and the ATRF212 begins the transmit sequence. It first sends the preamble sequence, which consists of 5 octets which serve as the synchronization header. This header is generated automatically by the ATRF212. The message that was previously written to the frame buffer is then sent. This is referred to as the payload, and is what will be written to the receiving frame buffer of any receiving devices. The payload consists of the following components:

- PHR – between 1 and 127, signifying the length of the message to be sent (18 bytes for these devices)
- FCF – the frame control field, which signifies the format of the following message
- Sequence number – indicates the device that the message originates from
- Addressing fields – 8 bytes for these devices, containing in order the destination PAN ID, the destination address, the source PAN ID, and the source address.
- IEEE 802.15.4 allows a security header to follow the addressing fields, but this was deemed unnecessary for this application, as the data being transmitted is publically available, and transmission is highly power consuming
- MAC Service Data Unit – the user specified data, which contains an additional 32 bit header with the intent of screening out unwanted messages, as well as two bytes containing the voltage level measured on the pressure sensor.
- FCS – the frame check sequence, which includes a checksum for the message for transmit/receive error screening

The devices are designed to self-organize into a partially connected mesh network, where each node (device) will connect to all other devices within its range. This network topology was deemed the most practical for this application, as a fully connected mesh would require that each node would be able to connect to every other node in the network. While this topology offers better redundancy and resilience to error, it was impractical to transmit wirelessly over the total length of the network, which could be miles between the most distant nodes. Since the network will exist along Bowman Creek (forming a long and thin network), a daisy chain structure was considered, but the high dependency of the chain on the integrity of each device (i.e. if one device was to fail, no devices beyond that device would be accessible) made it less attractive. The partially connected mesh allowed flexibility in the construction of the network, since if one device fails, the next device in the line may still be able to maintain connection in the network. It should be noted, however, that if the devices are placed such that the distance between them is close to the range of transmission that they allow, the network will default to a daisy chain topology. Additionally, this type of network will allow new devices to be added in between existing devices without modifying those that are already in place, rather than requiring that new devices be added to the end of the line.

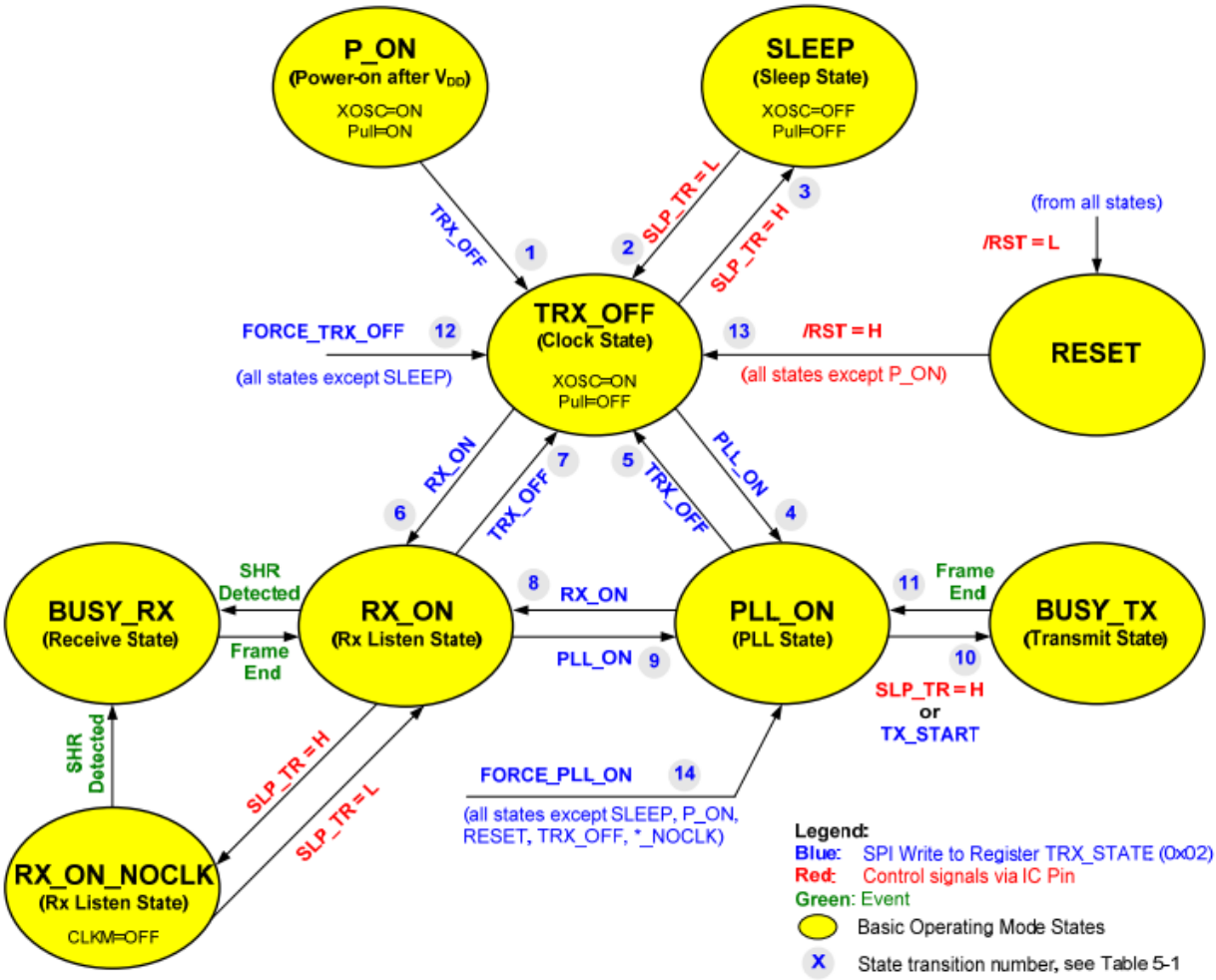
Software Description



The individual devices are controlled by the central microcontroller, which acts according to the software flowchart. Descriptions of the actions taken by the software for each box in the flowchart are provided below.

Microcontroller Setup – Upon turning on the device, the microcontroller moves into the initial setup phase. It first sets all interrupt settings and sets up the SPI and UART for interfacing with the wireless transceiver and computer terminal. It sets all of the appropriate pins to the necessary input/output and analog/digital state, sets up the watchdog timer (without enabling it) and timer 1, and sets up the analog to digital converter (ADC).

Wireless Transceiver Setup – After completing the microcontroller setup, the software sets up the wireless transceiver via SPI. The state diagram for this setup and the operation of the transceiver in general is shown below. It first moves into the TRX_OFF state, sets the communication channel (channel 8, 920 MHz), sets the device addresses, and enables the appropriate transceiver interrupts.



Active State (Receive Messages) – This is the default state for the state machine, which loops continuously. This state sets the wireless transceiver in the RX_ON state and checks the frame buffer for messages. If a message is in the frame buffer, it stores it in the array "payload," which is then accessible to other functions in the software.

Does time=Device_ID – Each device is programmed with a unique device ID (stored in the variable device_ID), which is the only difference between the devices (except for device 0, which is discussed later). Each device also keeps time with the timer Timer1 on the microcontroller. The timer is set to trigger an interrupt and increment the global time every time that it reaches its maximum value. If the time is equal to the device ID of an individual device, that device will branch into the measurement tree of the flowchart (discussed below). All other devices will continue downwards to the next decision.

Does time=sleep_time – When the time reaches the time that has been designated as the sleep time, the device will enter the sleep tree. The sleep time is changed by the software according to the messages that the device has received in order to allow scaling of the network. The sleep time is always set to three greater than the greatest device number that has sent a message over the network. This means that some energy is wasted during each cycle waiting for new messages, but new devices will be added to the network seamlessly, as any number of devices can be added, and sleep time will be incremented as needed. This system avoids waiting with the transceiver on for too long, which causes a high power draw, while also avoiding setting limits on the number of nodes in the network or worse, requiring alteration to already installed devices.

Read received message – This state checks the payload array and makes a number of decisions accordingly (listed below). It also begins the forwarding tree of the device.

Do the addresses match? – The software checks the relevant words of the payload array for address matching and to make sure that the 32 bit header is identical to what is expected from devices on this network. If all of the correct addresses and header sequence is read, the forwarding tree continues, otherwise, the state returns to receiving messages.

Has this message been seen? – The software checks if this device has already forwarded this packet. A large array (receivedPackets) has all bits cleared when the time is reset after each sleep cycle. Following this, the software sets the bit with index equal to the device's ID, and then sets the bit for the device ID of each packet that it receives. This ensures that no time or power is wasted forwarding packets that have already been seen (and forwarded) by this device. There was a conscious choice made during the design to not specify any other requirements for forwarding. Conceivably, one could only forward packets to devices upstream of this one (e.g. higher sequence number), or some similar criteria, but we wanted to be able to receive all of the data in

the network from any point along the network. Additionally, this allows new devices to be placed anywhere along the creek, regardless of their ID (though the user must never program devices with IDs in a non-sequential way, as devices with an ID more than three greater than the highest existing device on the network will be ignored until the empty space is filled)

Forward message – The device then forwards the entire message as it was received. To do this, the message is rewritten to the frame buffer and the transceiver is put into the state TX_ON until the message is sent.

Was the message from device 0? – The message is still stored in the payload array, so the software checks which device it originated from. If the message was originally sent by device 0, the global time is set to 0, synchronizing the time across the network (to within the necessary limits). The device then returns to the Active State to receive messages. This part of the code is different for device 0, as it does not receive its own messages. Instead of resetting the time upon receiving a signal, device 0 resets its own clock. After waking up from sleep, device 0 waits for ten more increments of the global time, and then resets its own clock to time 0. It then sends out its message, which acts as the synchronization signal for the entire network. In this way, all of the devices are synchronized to the clock of device 0.

Measurement tree – The device measures the water pressure, converts the measured pressure to a depth, and displays the measured value. It then sends the measured pressure to the other devices. This is described in more detail in the measurement subsystem.

Put wireless transceiver to sleep – The device sends the appropriate commands to change the state of the wireless transceiver chip into the TRX_OFF state, and then into the SLEEP state. This significantly reduces the power draw of the wireless transceiver by disabling all functions and communication (except by the wake up trigger pin).

Put microcontroller to sleep – The software then puts the microcontroller to sleep. To do this, the watchdog timer (WDT) is serviced and then enabled, and then the assembly instruction “wait” is sent to the microcontroller. It should be noted that devices which have been newly added to the network will not go to sleep, and instead will remain awake until they receive the synchronization message from device zero. This is necessary in order to make sure that new devices are not added out of synchronization with the network, and will always connect themselves if they are in range.

Sleep for 5 minutes – The microcontroller will enter sleep mode when the “wait” instruction is given. While it is in this state, the microcontroller disables most of its modules and significantly lowers its power consumption. The main oscillator (the fast RC onboard oscillator) is turned off, and only the low power oscillator (LPRC) remains on. The LPRC runs the WDT, which is set to wait for 262144 milliseconds (about 4.4 minutes) before waking up the microcontroller.

Wake up the microcontroller – Upon reaching its maximum value, the watchdog timer wakes up the microcontroller, and the code resumes from where the “wait” command was asserted.

Wake wireless transceiver up – Following waking up the microcontroller, the software wakes up the wireless transceiver by sending the commands to wake it up from the sleep state, and then to place it in the TRX_OFF state, from which it will be able to enter the RX_ON state when the device returns to the active state.

Subsystem Testing

The wireless communication subsystem was tested extensively in order to ensure that it would operate continuously without interruption, without requiring any input from the user. In the initial phases of testing, communication was established between devices operating at 2.4 GHz, as these were readily available and very similar to the 900 MHz devices. Two devices were provided by professor Schafer, one of which was configured to receive all messages, and one was configured to constantly send a stream of messages. The sending code was tested first, sending a single packet to the receiving device. The receiving code was written separately and tested with the provided transmit module. Upon receiving the message, the code would output the message to the computer terminal. The final testing done with the 2.4 GHz boards was a rudimentary forwarding scheme, which would receive a message from the transmit board, switch into transmit mode, and send the message (with some alteration) to the receive board, which would then output both the original and the altered messages to the computer terminal.

Following the testing on the 2.4 GHz boards, the 900 MHz boards were constructed. These were initially tested with the same code that the 2.4 GHz were tested with. As the main project code progressed, one device was set up with the pure receive and output code, while the project code was tested on two other devices. This setup allowed all of the messages sent over the network to be viewed in one place, which was important, because each device was supposed to receive and forward each message only once. If one or multiple devices forwarded messages without displaying them, it could lead to excessive crowding of the network and power usage. Once the devices were forwarding properly, without repeating messages unnecessarily, the code was tested for long periods of time. Whenever the code stopped executing, it was analyzed to find the source of the problem, and then modified to prevent this stoppage. Once the backbone of the code worked without any problems, minor changes were made, and it was confirmed that they worked properly by outputting to the terminal when the associated code had executed.

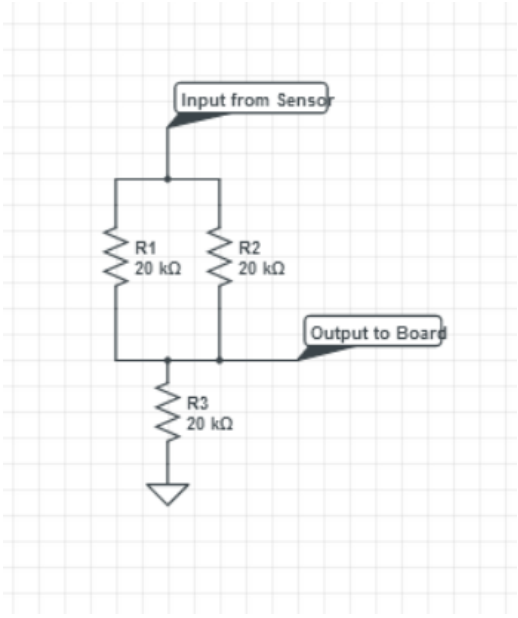
The sleep function was written and tested once the transmit/receive functionality was established. The transmit/receive code includes a global time that constantly increments and is displayed to the board LEDs, so it was easy to test when the board was sleeping, as this time would stop for the amount of time specified in the watchdog timer setup, then resume where it had taken off.

5.5 Detailed Operation of Sensor Subsystem

The purpose of this subsystem is to be able to accurately detect the level of water up to a certain depth. The design requirements for this subsystem were to be able to be powered by the main board that we were using and to measure water depth accurately enough as well as to a certain depth of water. Through discussions throughout the semester, we figured that being able to measure any depth up to about 30 feet would be adequate for the final design. This would accommodate the sensor being placed at the bottom of the creek in times of flooding.

For the actual sensor, we chose to use the Honeywell PX2AN1XX030PAAAX pressure transducer. A pressure transducer measures a pressure from its environment and converts that into an analog voltage on the output pin of the sensor. We chose this sensor for our system because it fit all of the requirements that we needed. It is powered by 5V which is perfect for the main board we were using. It also measures up to 30 PSI which is approximately 28 feet of water, just under our desired value, but very close nonetheless and definitely the closest we could come to the desired value.

The sensor has three pins on it: one power pin, one output pin, and a pin to connect to ground. The connector that is used is a Delphi Metri-Pack 150 which just allows easier connection to the three pins to other wires. We put a silicon coating around the connection to the wires to make them waterproof to the water that would inevitably touching them otherwise since the entire sensor would be immersed in water and would be shorted if this were not present. For our testing purposes also, we connected wires that were about 7-8 feet long so they would be able to reach from our board all the way down the PVC pipe we used as a water column.



The above circuit is the depiction of the connection from the sensor to the board. We experienced some amount of difficulty with the voltage drop we were trying to achieve. When we originally designed this circuit, we were expecting a voltage drop of 1/3 of the sensor output voltage because of the voltage divider we had designed. We had not thought about the fact that the connection to the board would put some sort of load on the overall circuit. Instead of a 1/3 drop, the actual drop was approximately a 60% drop. We thought about putting a common collector between the voltage divider and board to isolate the voltage, but the current gain would have been too high for the microprocessor we were using. Instead, we just kept this circuit which helped lower the voltage to a point that was accessible for our greater design.

6 System Integration Testing

6.1 Testing

The main part of the testing of the subsystems together involved getting the sensor to output the correct data to the terminal window we had set up. Originally, the code we had written was outputting negative numbers for depth which obviously was not correct since the lowest reading that should exist should be a depth of zero meters.

The bulk of what we had to do to best integrate them was calibrating the sensor within the program. This involved setting up the correct equation based on the particular sensor we had. The first problem that arose was that the voltage the sensor was outputting was too high and had to be scaled down. We did this by creating a voltage divider so that the output voltage from the sensor would be multiplied by 2/3. While the configuration we had did not put exactly 2/3 the output voltage into the input pin on the main board, we were able to scale it within the program.

Because we were having difficulty with getting this voltage to be linear in scale, we thought of creating a common collector circuit where the input would be 2/3 of the voltage of the output pin of the sensor (done by the voltage divider) and the output pin of the common collector circuit would be the input to the pin on the board. This would have correctly isolated. We decided to not go with this because the current gain would have been too high with the board we were using. Instead, we figured out that the input to the pin would scale approximately linearly with the voltage divider we were using. Therefore, we only had to correctly program the board to calculate the depth. The equation we ended up using was *depth = ((30*(voltage / 432 * 5 - 0.5)/4) - 13.905) * 74/3;* where “depth” is the calculated depth and “voltage” is the input voltage based from the sensor. The rest of the numbers are correction factors for atmospheric pressure and for the scaling of the sensor.

Once we had the correct scaling, all we had to do was output the value of the voltage to the terminal which was relatively easy. There was little to no issue with integrating the

battery charging circuit to the main board as the design for this system was robust and accounted for connecting it already.

6.2 Testing Met Design Requirements

The testing that we did on the system did meet the design requirements laid out to us. We were able to correctly measure the water to within 1-1.5 inches of the actual depth. Since this was the main purpose of the project, sensing water depth, being able to accurately do this was a great success. We were able to attain every depth from the sensor that we could achieve. Given the constraints of our testing system, the PVC pipe we used as a column of water only went up to four and a half feet. We attained a reading at every single inch within this column though from having the sensor completely out of the water to having the sensor at the very bottom of the column when it was completely filled. We achieved this both before demonstration day and on demonstration day.

We were also able to successfully send data along the line of sensors using ZigBee communications. When our sensor read in data to board 00 from the sensor we had working, it successfully sent it down the line of sensor. We set this up to first test just sending it from board 00 to board 01, and it worked. We then set up board 01 to send data it received from board 00 to board 02 which it successfully did. Our system was not designed to send data from every board to every board. The coding of our boards allowed this not to happen, so we were excited to see the system work. We thought it would be no problem to send data along any arbitrary number of points as well.

Finally, the testing of our battery subsystem proved to meet our design requirements. We were able to demonstrate a gain in voltage all the way up to the expected value when the system had been charging for a long time. We were not able to test it using a solar cell, but the battery we had hooked up to our lithium-ion cells showed a gain in voltage across the cells after a great deal of charging. We were able to test that there was voltage across the solar cell we were planning on using, so there would be no problem with charging it if the solar cell were actually implemented into the overall system instead of the battery. The lithium-ion cells also did successfully power the sensor system once we had everything hooked up and working.

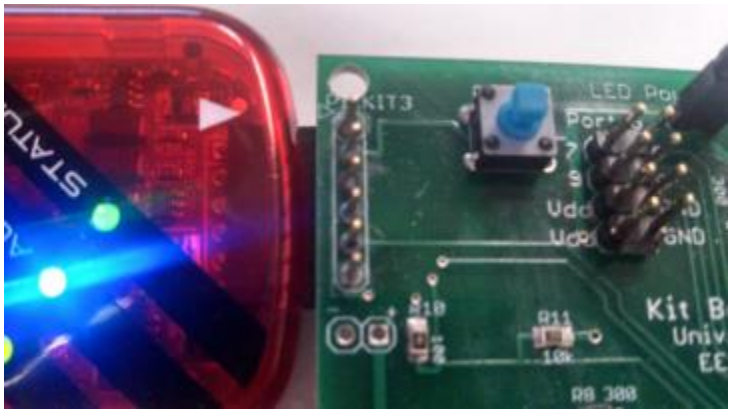
7 User's Manual

7.1 Setup

The setup of this device is relatively simple. If the device has been provided with the device number already set, not setup is needed. If not, the device will need to be programmed with a device number. To do this, follow these steps:

1. Plug the device into a computer via the micro USB port on the device.
2. Plug the PicKit 3 programmer into the computer via the mini USB connector on the PicKit.

3. Attach the Pickit 3 programmer to the device programmer pins on the top left as shown below.



The PicKit 3 programmer attached to the device board. Image courtesy of R. Michael Schafer

4. Open MPLAB X IDE on the computer.
5. Open the project “CommProject8”
6. Under Source Files, open the c file mainComm8.c
7. Edit the device_ID variable to the desired device ID. Note that in order for the network to function properly, there must be no gaps in device IDs present in the network.
8. Press the button “Make and Program Device Main Project”
9. Repeat for each device to be programmed, incrementing device IDs

7.2 Installation

Installation of the device should be completed as follows for normal devices:

1. Find a suitable place for the device to be installed. This should be at the lowest point in the creek, so that any water in the creek will be measured.
2. Connect the solar panel to the device and place it somewhere where it is able to gather as much sunlight as possible
3. Press the reset button on the device. If it has been programmed properly, it will begin counting up in binary on the onboard LEDs. This should continue for a minute or so, and then stop. It will resume counting after the specified sleep time has passed, and shortly thereafter reset to a count of zero.
4. Once these steps have been completed, the device should be connected to the network.

If you want to read data from the network, follow these steps:

1. Connect your device to a computer via the micro USB port.
2. Open a terminal (with a program such as Putty) connected to the appropriate COM port, with the speed set to 57600
3. If the device is connected to the network, it will display measurements and messages that it receives

7.3 How to tell if the device is working

If the device is working and connected to the network, it will count up in binary on the onboard LEDs, and if it is connected to a computer, it will output to the terminal whenever it receives messages.

7.4 Troubleshooting

Device LEDs are not counting.	The device may be out of battery. Connect it to a working power source.
Device LEDs do not pause counting.	The device is not connected to the network. There may be a number of reasons for this. • Device 0 is not sending the synchronization message. • This device is out of communication range with the other devices. Move it closer or place more devices in between
One device on the network does not send data about the water level	It may be out of range or power.
Measured water level varies slightly between measurements.	This is normal, the device will only measure with an accuracy of about ±1 inch
Devices beyond a certain point do not connect to the network.	The network employs message forwarding to deliver messages. If the distance between two devices is very great, no messages from beyond that gap will be delivered.

8 To-Market Design Changes

The first change that would be made to the overall system would be the acquisition of a better sensor. The sensor that we were using was pretty much the only one we could find that fit within the budget the design constraints we had to deal with. There were other sensor that could have been much more accurate for a lot more money, up to the thousands of dollars, but obviously we did not have the budget for that. With the sensor that we got for our final design, we were able to measure depths accurately to the inch, but we were not able to do much beyond that. Given a more accurate sensor and a board that could accommodate this greater accuracy, we could have been able to measure depths to a much greater precision.

The other major design change that would be needed before this product could go out to market would be involving the sensor system. We actually got a lot of questions about this during our presentation. In its current state, our overall system simply outputs the reading to a terminal from the one sensor that we have. It can output this data to any number of terminals based on the number of boards that we have sending data to one another. Beyond that, we do not have any analytical capabilities for the data we are sensing. It would have been difficult since the data we would be receiving

would be need to be compared with other data, and since we only had one sensor we had nothing with which to compare that data.

If our product were to go out to market though, we would need to implement some sort of central data dump where every five minutes when the sensors woke up, all the data along the entire line of sensors would be dumped into some sort of central computer where the data from board 00 would be compared to that of board 01, board 01 would be compared to board 02, etc. This way, if there were a certain amount of discrepancy between any two points within the line, some sort of error could be sent to a technician and people could be dispatched to see if there were a blockage in the creek of any sorts.

The lithium-ion cells were the final part of the system that might need to be changed before the final product could be sent out to market. The ones that we used to not have the best current rating and are rather dangerous. Lithium-ion cells in general are flammable, and we would not want anything to be possibly flammable and dangerous, especially in an area where we are trying to preserve the habitat. We would need to look into safer cells for this to be possible. The charging PCB we designed might also need to be slightly changed to accommodate these changes in batteries should this change be made for the market.

9 Conclusions

Overall the final product for our Senior Design Project turned out well. We were able to implement well the system requirements that were laid out to us in the project. When we talked with Gary Gilot at the project presentation, he seemed pleased with what we were able to come up with as our final implementation of the design. Had we been able to communicate better with EmNet from the beginning of the semester, we might have been able to go a little bit farther with the project with ideas such as piggybacking our sensor onto their communication system. This was not viable because they wanted us to buy their product for a couple hundred dollars though, so we went with what we could do. We were able to sense well with the sensor we bought and had the power management and communication systems working as best they could. Given another year (and possibly more funds) we would be able to make a finalized device with most if not all of the ideas that were in the To-Market Design Changes Section.

10 Appendices

10.1 Relevant parts or component data sheets

Honeywell Heavy Duty Pressure Transducer PX2AN1XX030PAAAX
http://media.digikey.com/pdf/Data%20Sheets/Honeywell%20Sensing%20&%20Control%20PDFs/PX2_Series_DS~.pdf

AT86RF212 Wireless Transceiver

http://www.atmel.com/images/doc8168.pdf

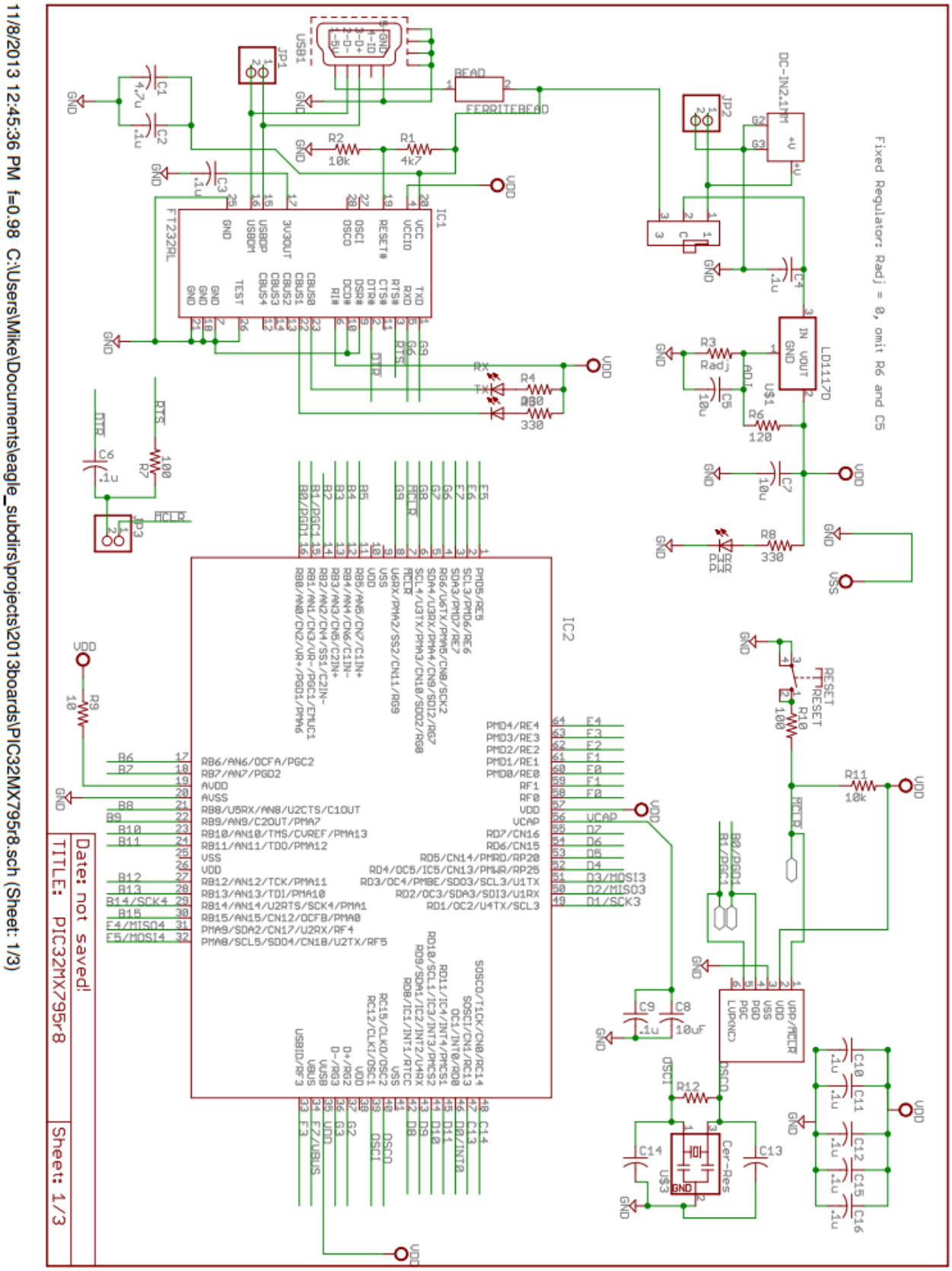
Balun/Filter combination 863-928 MHz (0805)
http://www.johansontechnology.com/images/stories/ip/balun-filters/Balun_Filter_Combo_0896FB15A0100_Prelim.pdf

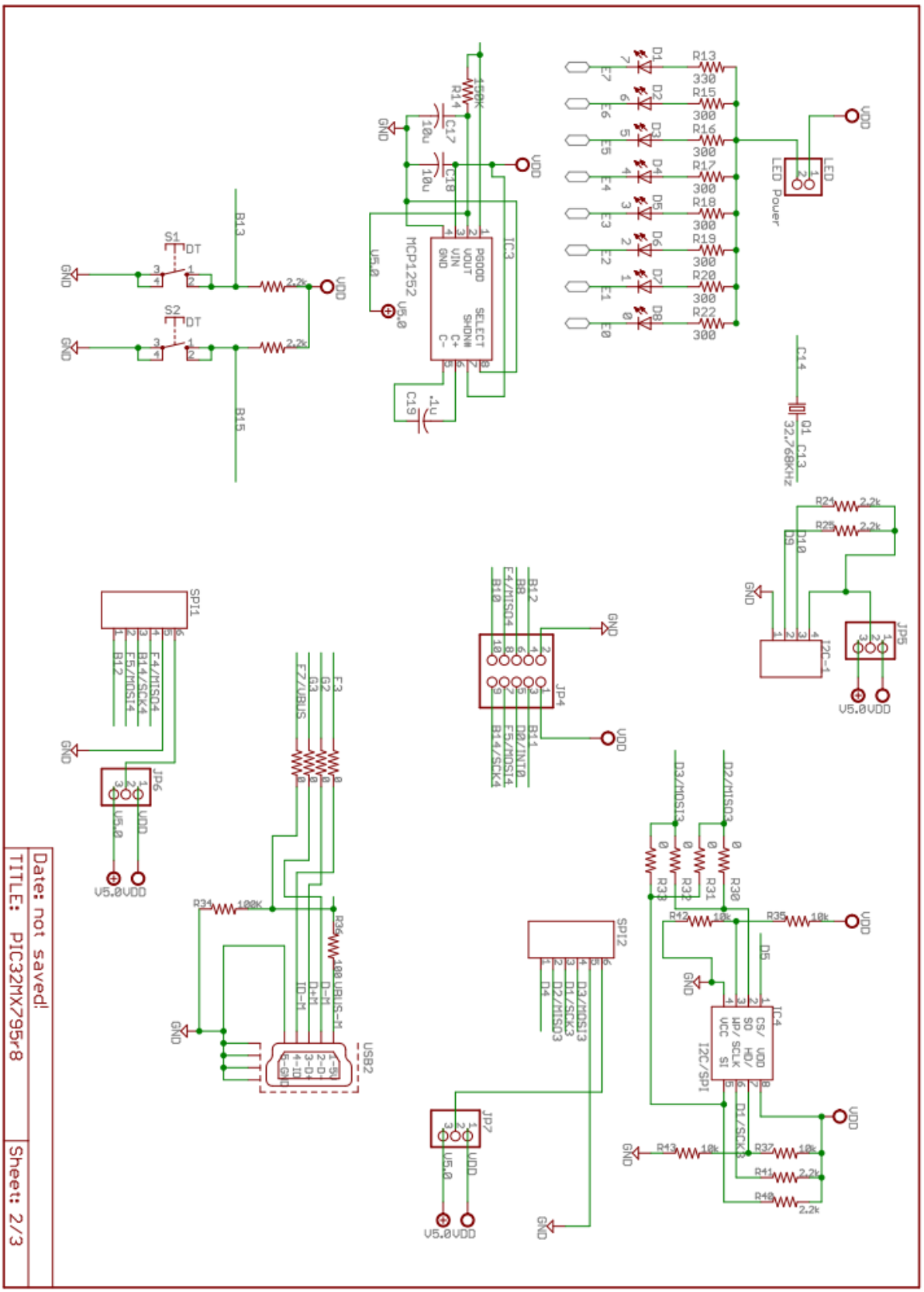
916 MHz Chip Antenna
https://www.linxtechnologies.com/resources/data-guides/ant-xxx-chp-x.pdf

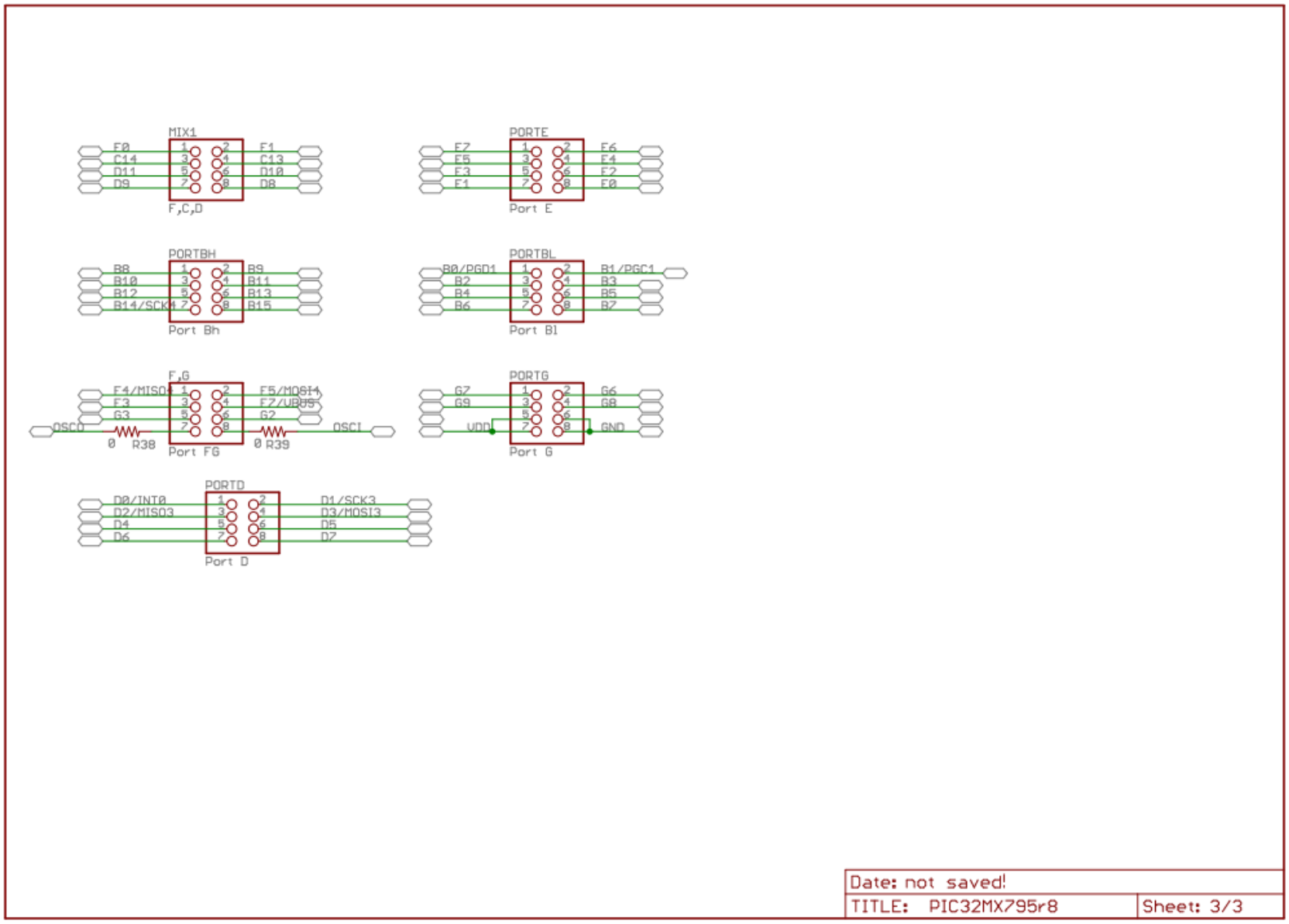
PIC32MX795f512h Microcontroller
http://ww1.microchip.com/downloads/en/DeviceDoc/61156H.pdf

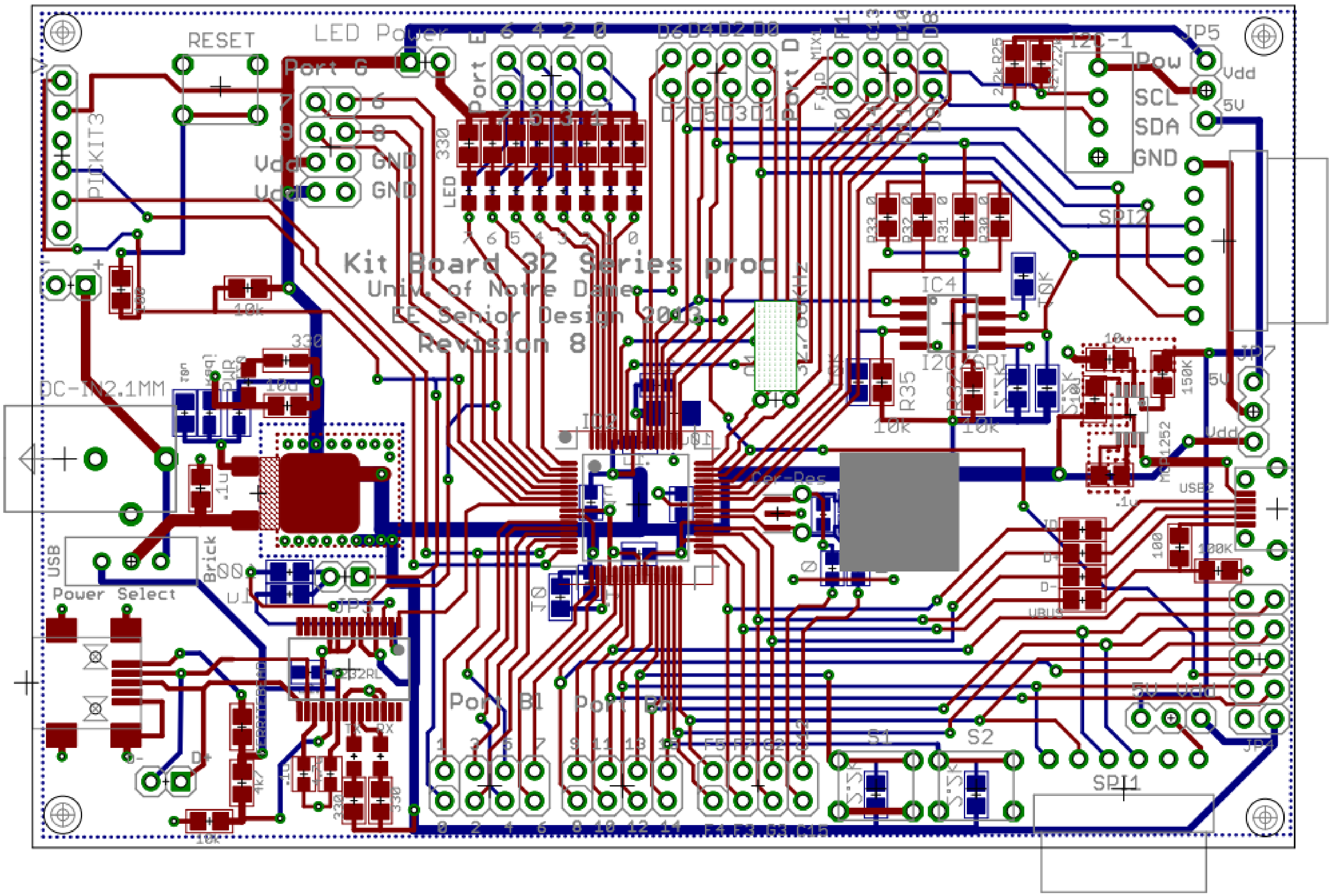
BQ24650 Sychronous Switch-Mode Battery Charge Controller
http://www.ti.com/lit/ds/symlink/bq24650.pdf

10.2 Complete Hardware Schematics

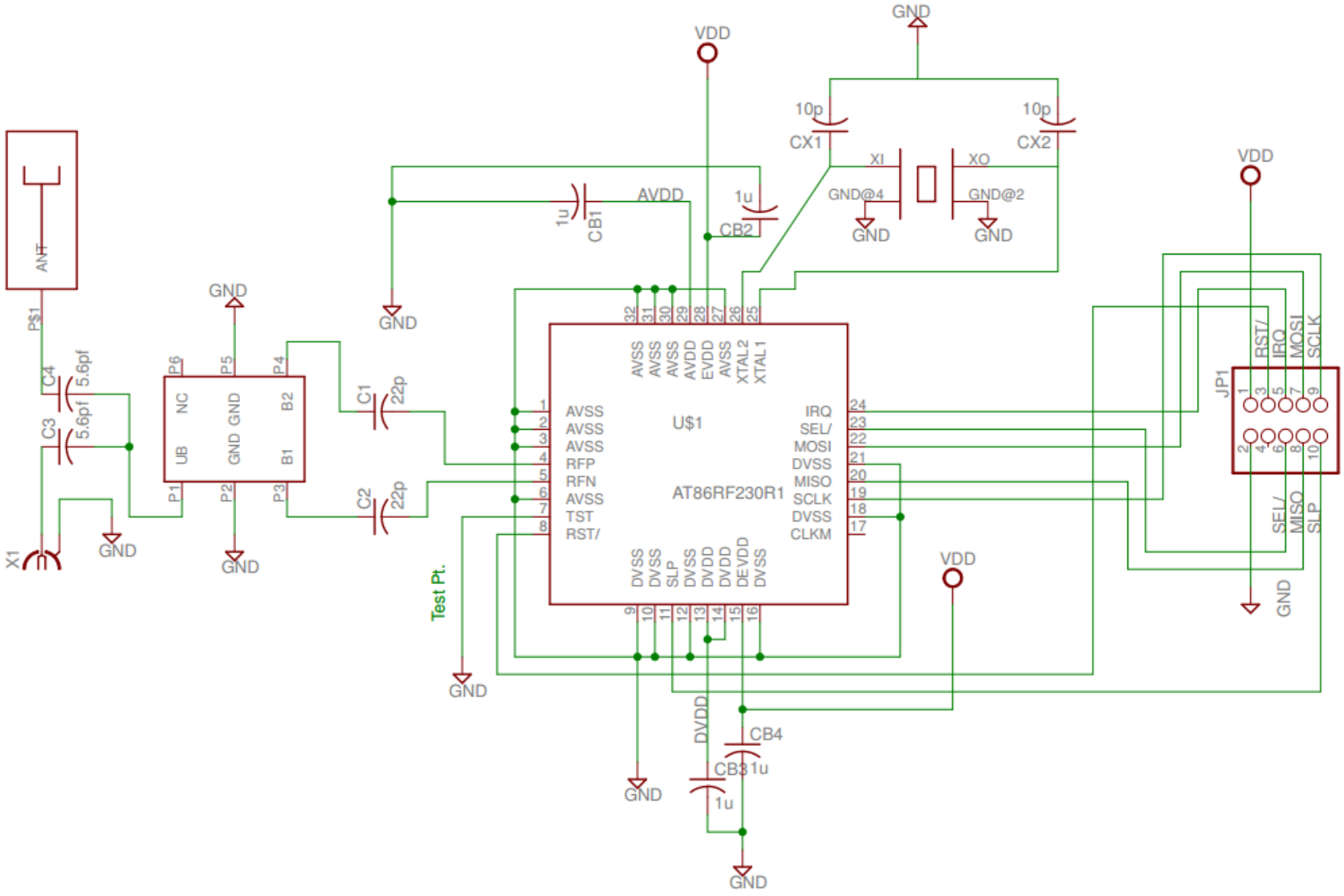



Date: not saved!
TITLE: PIC32MX795r8
Sheet: 2/3
11/8/2013 12:45:36 PM f=0.98 C:\Users\Mike\Documents\eagle_subdirs\projects\2013boards\PIC32MX795r8.sch (Sheet: 2/3)

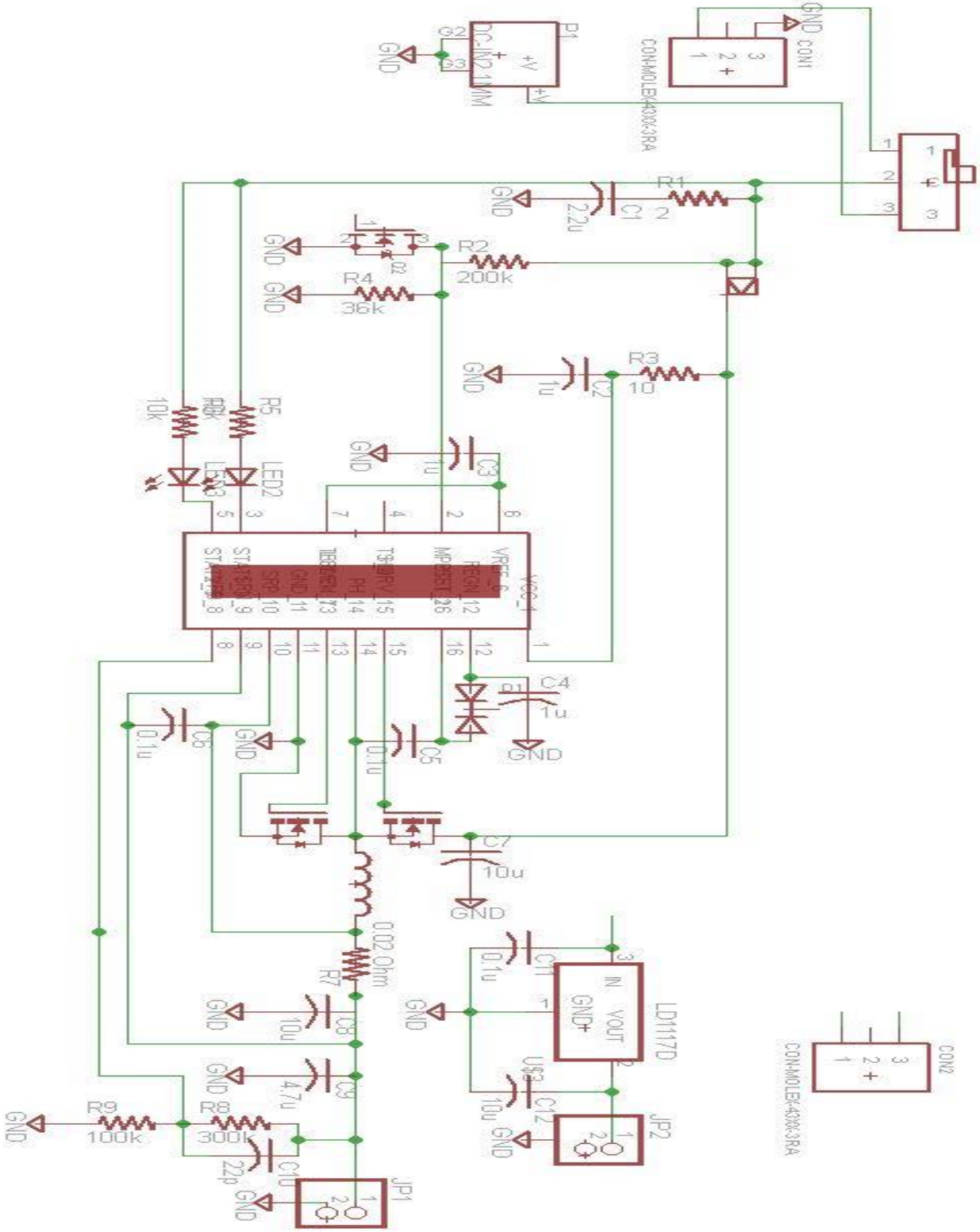

MIX1
F0
C14
D11
D9
F1
C13
D10
D8
F,C,D
PORTE
E7
E5
E3
E1
E6
E4
E2
E0
Port E
PORTBH
B8
B10
B12
B14/SCK4
B9
B11
B13
B15
Port Bh
PORTBL
B0/PGD1
B2
B4
B6
B1/PGC1
B3
B5
B7
Port Bl
F,G
F4/MISO4
F3
G3
F5/MOSI4
F2/USB5
G2
OSCO
OSCI
0 R38
0 R39
Port FG
PORTG
G7
G9
G6
G8
VDD
GND
Port G
PORTD
D0/INT0
D2/MISO3
D4
D6
D1/SCK3
D3/MOSI3
D5
D7
Port D
Date: not saved!
TITLE: PIC32MX795r8
Sheet: 3/3

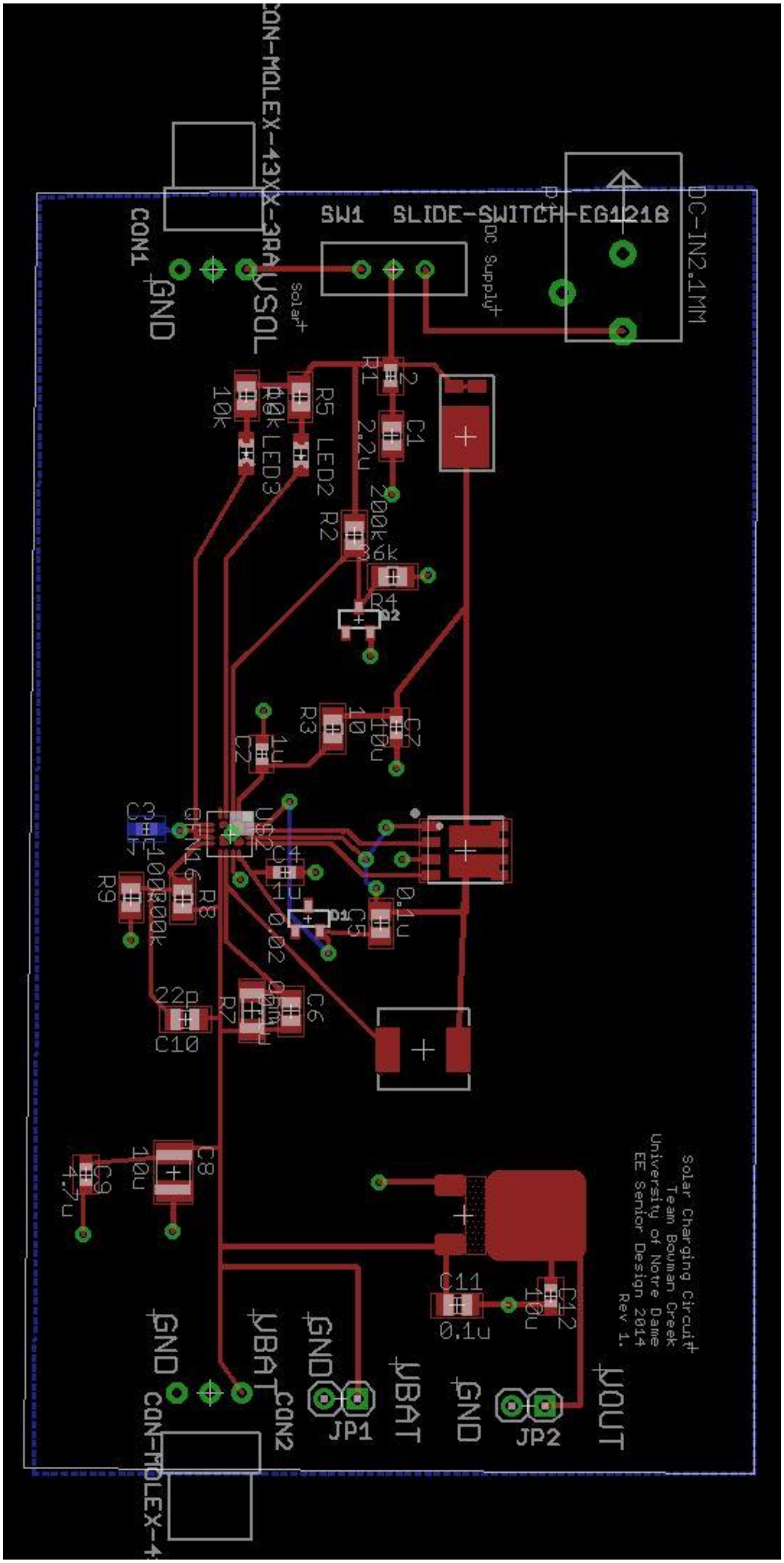


11/8/2013 12:46:43 PM f=2.34 C:\Users\Mike\Documents\eagle_subdirs\projects\2013boards\PIC32MX795r8.brd



1/23/2014 4:51:25 PM f=1.35 C:\Users\Mike\Documents\eagle_subdirs\projects\ATMEL ZiigBee\zigbee_xmtr-r3-flip.sch (Sheet: 1/1)


GND
CON1
CON-MOLEX-4030-3RA
P1
DC-JK2.1MM
+V
C1
2.2u
R1
2
Q2
R2
200k
R4
36k
C2
1u
R3
10
R5
10k
LED2
LED3
C3
1u
VCC_1
VREF_6
REGN_12
MP_RESET_16
TS_IBRV_15
PH_14
CHG_11
SRP_10
STAT/SDA_9
STAT/SCL_8
C4
1u
C6
0.1u
C5
0.1u
C7
10u
0.02_Ohm
R7
C11
0.1u
LD1117D
IN
VOUT
GND
C8
10u
C9
4.7u
C12
10u
JP2
R9
100k
R8
30
C10
22p
JP1
CON2
CON-MOLEX-4030-3RA


CON-MOLEX-43XX-3RA
CON1
GND
VSOL
SW1
SLIDE-SWITCH-EG1218
DC-IN2.1MM
Solar Charging Circuit
Team Bouman Creek
University of Notre Dame
EE Senior Design 2014
Rev 1.
GND
VBAT
CON2
JP1
GND
VBAT
GND
JP2
VOUT

10.3 Complete Software Listing

```
/*
 * File:   mainComm8BK.c
 * Author: Galen
 *
 * Created on April 20, 2014, 9:40 PM
 */


#include <stdio.h>
#include <stdlib.h>
#include <xc.h>
#include "configbitsrev8.h"
#include <plib.h>
#include <sys/attribs.h>


/*
 *
 */

int time = 0;
int maxTime = 968,sleepTime = 1000;
int waterLevel;

void __ISR(_TIMER_1_VECTOR, IPL6AUTO) Timer1Hand(void) {


    INTClearFlag(INT_T1);
    IFS0bits.T1IF = 0;

    time++;
    LATE = 0xFF - time;

    if (time == sleepTime){

        // write SLEEP

putu(10);putu(13);putu(83);putu(76);putu(69);putu(69);putu(80);
        putu(10);putu(13);

        atrf_sleep();
        WDTCONSET = 0x01;       // service WDT
        WDTCONSET = 0x8000;     // enable WDT
        asm volatile("wait");
```

```
        time = maxTime-10;
        WDTCONCLR = 0x8000;     // disable WDT
        atrf_wake();

        // write WAKE UP

putu(10);putu(13);putu(87);putu(65);putu(75);putu(69);putu(32);p
utu(85);putu(80);


        waterLevel = measure_Pressure();
    }


    /*
    INTClearFlag(INT_T1);
    IFS0bits.T1IF = 0;
    time++;
    if (time < 10){
        LATE = LATE - 0x1;
    }
    if (time == 20){
        time = 0;
    }*/
}

int * receive_Loop(){
    int
IRQdata,rData,PHR,FCF1,FCF0,seq,addr0,addr1,pan0,pan1,word;
    char dispChar[10];
    static int payload[127];
    int i = 0;

    // Begin receiving signals
    // Send RX_ON command

    // command write to TRX_STATE
    // write 0x06 (state change to RX_ON)
    atrf_SPI(0xC2,0x06);


    // command read from IRQ_STATUS
    // send empty byte, store status in IRQdata
    IRQdata = atrf_SPI(0x8F,0x00);
    //LATE = IRQdata;         // show IRQ_STATUS on LED array

    // Check for address match
```

```
    if ((IRQdata & 0x20) == (0x20)){
        putu(10);
        putu(13);
        //putu(0x6D); // put m if addresses match
    }

    // Check for RX frame end and output frame
    if ((IRQdata & 0x08) == (0x08)){
        //putu(0x65); // put e if end of transmission reached
        LATBbits.LATB12 = 0;     // set chip select to 0
(active)

        rData = do_SPI(0x20);   // command read from frame
buffer
        payload[0] = do_SPI(0x00);   // send empty byte, store
PHR in PHR
        payload[1] = do_SPI(0x00);   // send empty byte, store
FCF1
        payload[2] = do_SPI(0x00);   // send empty byte, store
FCF0
        payload[3] = do_SPI(0x00);   // get sequence number
        payload[4] = do_SPI(0x00);   // get address0 (lower 8
bits)
        payload[5] = do_SPI(0x00);   // get address1 (upper 8
bits)
        payload[6] = do_SPI(0x00);   // get pan0 (lower 8 bits)
        payload[7] = do_SPI(0x00);   // get pan1 (upper 8 bits)
        payload[8] = do_SPI(0x00);   // get address0 (lower 8
bits)
        payload[9] = do_SPI(0x00);   // get address1 (upper 8
bits)
        payload[10] = do_SPI(0x00);   // get pan0 (lower 8 bits)
        payload[11] = do_SPI(0x00);   // get pan1 (upper 8 bits)


        i=0;
        while (i<(payload[0]-11)){
//////////////////////////////
            payload[i+12] = do_SPI(0x00);   // get next word


            i++;
        }
        LATBbits.LATB12 = 1;     // set chip select to 1
(inactive)
```

```
    }

    return (payload);
}

int main(int argc, char** argv) {

    // see if code is running
    TRISE = 0x0;
    LATE = 0xFF;

    int i=0, device_ID=2, packetSent=0;
    int *payload;
    int maxDevices=device_ID;

    // Set up initial settings for SPI, UART, Timer1, stuff
    pic_SPI_setup();

    // Set up AT86RF212 state, addresses, channel
    atrf_setup();



    int receivedPackets[1000] = {0,0,0,0,0,0,0,0,0,0};
    receivedPackets[device_ID] = 1;

    while(1){



        if ((time == (3*device_ID)) && (!packetSent)){
            //
            // do measurement here? (if it is fast)
            //

            send_measurement(device_ID,waterLevel);
            packetSent = 1;
        }

        else {
```

```
            payload = receive_Loop();

            int received = 0;

            for(i=0;i<sizeof(receivedPackets);i++){
                if (receivedPackets[payload[3]]){
                    received = 1;
                    //LATE = payload[3];
                }
            }

            // ((FCF0 & 0x33) != 0)
            //  && (payload[3] != 0x88)
            // (payload[3] != oldseq) &&
            //  && (payload[12] == 0x23) && (payload[13] ==
0x57) && (payload[14] == 0x11) && (payload[15] == 0x13)

            // this condition requires that things be addressed
in a certain way
            if ((payload[0] > 0) && ((payload[2] & 0x08) ==
0x08) && (!received) && (payload[12] == 0x23) && (payload[13] ==
0x57) && (payload[14] == 0x11) && (payload[15] == 0x13)){

                // mark this packet as received
                receivedPackets[payload[3]] = 1;
                //oldseq = payload[3];

                // Format and print received data
                receive_Print(payload);

                // Write to frame buffer
                write_frame_buffer(payload);

                // Send Payload
                send_Payload(payload);

                // if data is from device 0, synchronize time,
reset sent/received
                //  && (oldseq != 0)
                if ((payload[3] == 0) && (time != 0)){
                    //putu(122);
                    //putu(10);
                    //putu(13);
                    packetSent = 0;
                    for (i=0;i<sizeof(receivedPackets);i++){
                        receivedPackets[i] = 0; // clear
received packets
```

```
                    }
                    receivedPackets[device_ID] = 1;
                    time = 0;
                }

                if (payload[3] > maxDevices){
                    maxDevices = payload[3];
                }

            }

        }

        sleepTime = 3*(maxDevices+3);

    }

    //IFS2bits.U6RXIF = 0;

    return (EXIT_SUCCESS);
}

int do_SPI(int sendData) {
    IFS1bits.SPI4RXIF = 0;
    SPI4BUF = sendData;
    while(!IFS1bits.SPI4RXIF){}
    return SPI4BUF;
}

int atrf_SPI(int command,int what){
    int rData;
    LATBbits.LATB12 = 0;     // set chip select to 0 (active)

    rData = do_SPI(command);   // command write/read to
register/frame buffer
    rData = do_SPI(what);   // write state/data

    LATBbits.LATB12 = 1;     // set chip select to 1 (inactive)
    return(rData);
}

int pic_SPI_setup(){

    int rData;
    // Set up UART stuff
    serial_init(57600UL);
    set_output_device(1);
```

```
    //INTClearFlag(INT_T2);
    //INTSetVectorPriority(INT_T2, 2);
    IFS2bits.U6RXIF = 0;
    IPC12bits.U6IP = 1;
    IPC12bits.U6IS = 1;
    IEC2bits.U6RXIE = 1;


    // Disable JTAG (on pins B10, B11, B12)
    DDPCONbits.JTAGEN = 0;
    //Set Pins to digital
    AD1PCFG = 0xFFFE;

    // set pins to output
    TRISBbits.TRISB8 = 0;
    TRISBbits.TRISB12 = 0;
    TRISBbits.TRISB10 = 0;
    TRISBbits.TRISB11 = 0;
    TRISBbits.TRISB14 = 0;
    TRISBbits.TRISB0 = 1;
    TRISDbits.TRISD0 = 1;   // set IRQ pin to input

    LATBbits.LATB10 = 0;    // set SLP_TR to low
    LATBbits.LATB11 = 0;    // set RST/ to 0 (reset ATRF231)
    LATBbits.LATB12 = 1;     // set chip select to 1 (active
low)
    //

    OSCCONbits.PBDIV1 = 0;  // PBCLK is SYSCLK divided by 2
    OSCCONbits.PBDIV0 = 1;
    OSCCONbits.SLPEN = 1;   // sleep when WAIT instruction is
given
    //WDTCONCLR = 0x0002;     // disable WDT window mode
    //WDTCONSET = 0x8000;     // Enable WDT

    // Set up Timer1
    T1CONbits.ON = 0;   // timer is off
    T1CONbits.TCKPS = 0x03; // prescale by 256 (0x03)
    T1CONbits.TWDIS = 1;
    TMR1 = 0x0;         // set timer value to 0
    PR1 = 0x2710;       //
    IFS0bits.T1IF = 0;      // clear T1 flag
    IPC1bits.T1IP = 0x06;   // second highest priority
    IPC1bits.T1IS = 0x00;   // lowest subpriority
    IEC0bits.T1IE = 1;
    T1CONbits.ON = 1;   // turn timer on
```

```
    // Set up Watchdog Timer (WDT)
    OSCCONSET = 0x10;       // Set power saving mode to sleep
    WDTCONCLR = 0x0002;     // disable WDT window mode

    // Set up ADC
    IFS1SET = 0x02;
    IPC6bits.AD1IP = 1;         // Priority
    IPC6bits.AD1IS = 1;         // Subpriority
    //IEC1bits.AD1IE = 0;
    TRISBbits.TRISB0 = 1;       // set AN0 to input (RB0)
    AD1CON1bits.SIDL = 1;       // turn ADC off during sleep
    // AD1CON2bits.VCFG = 0x01; // add this and change many
things for vref version
    AD1CSSL = 0x0001;           // only look at AN0
    AD1CON1bits.ASAM = 1;       // automatic sampling

    // Set up SPI4
    IEC1CLR=0x0700; // disable all interrupts for SPI4
    SPI4CON = 0; // Stops and resets the SPI4.
    rData=SPI4BUF; // clears the receive buffer for SPI4
    SPI4CONbits.ENHBUF = 0; // do not use enhanced buffer mode
    IFS1CLR=0x0700; // clear any existing event for SPI4
    IPC8CLR=0x1f; // clear the priority for SPI4
    IPC8SET=0x0d; // Set IPL=3, Subpriority 1 for SPI4
    //IEC1SET=0x0700; // Enable RX, TX and Error interrupts for
SPI4

    SPI4BRG=0x1F; // use FPB/2 clock frequency
    SPI4STATCLR=0x40; // clear the Overflow

    SPI4CONbits.CKE = 1; // 0 in previous?
    SPI4CONbits.CKP = 0; // 1 in previous?
    SPI4CONbits.SMP = 1;
    SPI4CONbits.MSTEN = 1;
    SPI4CONbits.MODE32 = 0;
    SPI4CONbits.MODE16 = 0;
    SPI4CONbits.ON = 1;

    LATBbits.LATB11 = 1;    // end reset of atrf 231



    rData=SPI4BUF;          // clears the receive buffer

    // enable interrupts
    INTCONbits.MVEC = 1;
```

```
    asm("ei");

    return;
}

int atrf_setup(){

    int IRQdata;
    // set chip select to 0 (active)
    //LATBbits.LATB12 = 0;

    // send test SPI
    // rData = do_SPI(0x9C); // command read part number
    // rData = do_SPI(0x00); // send empty byte, store part no.
in rData

    // set chip select to 1 (inactive)
    //LATBbits.LATB12 = 1;


    // command write to TRX_STATE
    // write 0x08 (state change to TRX_OFF)
    atrf_SPI(0xC2,0x08);

    // command write to TRX_CTRL_1
    // write 0x24 (automatic checksum, monitor TRX_STATUS)
    atrf_SPI(0xC4,0x24);

    // command read from TRX_STATUS
    // send empty byte, store status in rData
    atrf_SPI(0x81,0x00);
    //LATE = rData;           // set SPI return to LED array


    // command write to PHY_CC_CCA
    // write 0x28 (channel 8 (920 MHz), no CCA)
    atrf_SPI(0xC8,0x28);

    // Set Addresses
    // command write to SHORT_ADDR_0
    // write 0xCD (lower 8 bits)
    atrf_SPI(0xE0,0xCD);

    // command write to SHORT_ADDR_1
    // write 0xAB (upper 8 bits)
    atrf_SPI(0xE1,0xAB);
```

```
    // command write to PAN_ID_0
    // write 0xEF (lower 8 bits)
    atrf_SPI(0xE2,0xEF);

    // command write to PAN_ID_1
    // write 0xEF (upper 8 bits)
    atrf_SPI(0xE3,0xEF);

    // command write to TRX_STATE
    // write 0x09 (state change to PLL_ON)
    atrf_SPI(0xC2,0x09);

    // command read from TRX_STATUS
    // send empty byte, store status in rData
    atrf_SPI(0x81,0x00);
    //LATE = rData;           // set SPI return to LED array

    // command read from IRQ_STATUS
    // send empty byte, store status in rData
    IRQdata = atrf_SPI(0x8F,0x00);
    //LATE = IRQdata;           // show IRQ_STATUS on LED array

    // command write to IRQ_MASK
    // enable RX_START and TRX_END interrupts
    atrf_SPI(0xCE,0x6C);


    // End Initial Setup of AT86RF212

}

int receive_Print(int payload[127]){

    int PHR, FCF1, FCF0, addr0, addr1, pan0, pan1, seq, word;
    int voltageInt, pressureInt, depthInt, depth_cm;
    float voltage, pressure, depth;
    int i=0;
    char dispChar[10];

    putu(10);
    putu(13);

    PHR = payload[0];
            FCF1 = payload[1];
            FCF0 = payload[2];
            //LATE = FCF1;
            /*
```

```
if ((FCF0 & 0x33) != 0){

    return;
}
*/
sprintf(dispChar, "%d", PHR-11);
if (PHR-11 < 0x10){
    dispChar[1] = dispChar[0];
    dispChar[0] = '0';
}
if (PHR-11 < 0x100){
    dispChar[2] = dispChar[1];
    dispChar[1] = '0';
}

putu(dispChar[0]);
putu(dispChar[1]);
putu(dispChar[2]);
putu(0x20);

// display FCF first half (TYPE SFAP)

for(i=0;i<8;i++){
    if ((FCF1 & (0b01<<i)) == (0x01<<i)){
        dispChar[i] = '1';
    }
    else {
        dispChar[i] = '0';
    }

}

putu(dispChar[3]);
putu(dispChar[2]);
putu(dispChar[1]);
putu(dispChar[0]);
putu(0x20);
putu(dispChar[7]);
putu(dispChar[6]);
putu(dispChar[5]);
putu(dispChar[4]);
putu(0x20);

// display FCF second half (DM SM)
i = 0;
for(i=0;i<8;i++){
    if ((FCF0 & (0b01<<i)) == (0x01<<i)){
```

```
        dispChar[i] = '1';
    }
    else {
        dispChar[i] = '0';
    }

}

putu(dispChar[3]);
putu(dispChar[2]);
putu(0x20);
putu(dispChar[7]);
putu(dispChar[6]);
putu(0x20);

// get and display sequence number

seq = payload[3];
sprintf(dispChar, "%x", seq);

if (seq < 0x10){
    dispChar[1] = dispChar[0];
    dispChar[0] = '0';
}


putu(dispChar[0]);
putu(dispChar[1]);
if (seq>99){
    putu(dispChar[2]);
}
if (seq>999){
    putu(dispChar[3]);
}
putu(0x20);

// get and display destination addresses

addr0 = payload[4];
addr1 = payload[5];
sprintf(dispChar, "%x", addr1);
if (addr1 < 0x10){
    dispChar[1] = dispChar[0];
    dispChar[0] = '0';
}

putu(dispChar[0]);
```

```
putu(dispChar[1]);
sprintf(dispChar, "%x", addr0);
if (addr0 < 0x10){
    dispChar[1] = dispChar[0];
    dispChar[0] = '0';
}

putu(dispChar[0]);
putu(dispChar[1]);
putu(0x20);

// get and display destination PAN addresses

pan0 = payload[6];
pan1 = payload[7];
sprintf(dispChar, "%x", pan1);
if (pan1 < 0x10){
    dispChar[1] = dispChar[0];
    dispChar[0] = '0';
}

putu(dispChar[0]);
putu(dispChar[1]);
sprintf(dispChar, "%x", pan0);
if (pan0 < 0x10){
    dispChar[1] = dispChar[0];
    dispChar[0] = '0';
}

putu(dispChar[0]);
putu(dispChar[1]);
putu(0x20);

// get and display source addresses

addr0 = payload[8];
addr1 = payload[9];
sprintf(dispChar, "%x", addr1);
if (addr1 < 0x10){
    dispChar[1] = dispChar[0];
    dispChar[0] = '0';
}

putu(dispChar[0]);
putu(dispChar[1]);
sprintf(dispChar, "%x", addr0);
if (addr0 < 0x10){
```

```
            dispChar[1] = dispChar[0];
            dispChar[0] = '0';
        }

        putu(dispChar[0]);
        putu(dispChar[1]);
        putu(0x20);

        // get and display source PAN addresses

        pan0 = payload[10];
        pan1 = payload[11];
        sprintf(dispChar, "%x", pan1);
        if (pan1 < 0x10){
            dispChar[1] = dispChar[0];
            dispChar[0] = '0';
        }

        putu(dispChar[0]);
        putu(dispChar[1]);
        sprintf(dispChar, "%x", pan0);
        if (pan0 < 0x10){
            dispChar[1] = dispChar[0];
            dispChar[0] = '0';
        }

        putu(dispChar[0]);
        putu(dispChar[1]);
        putu(0x20);

        // set output first entry to length
 ///    payload[0] = PHR-11;
////////////////////////////////
 ///    LATE = payload[0];
        i=0;
        while (i<(payload[0]-11)){
////////////////////////////////
            word = payload[i+12];
            sprintf(dispChar, "%x", word);
            if (word < 0x10){
                dispChar[1] = dispChar[0];
                dispChar[0] = '0';
            }

            putu(dispChar[0]);
            putu(dispChar[1]);
            putu(0x20);
```

```
            i++;
        }


        LATBbits.LATB12 = 1;     // set chip select to 1
(inactive)

        if (1){          //seq%10 == 1
            putu(10);
            putu(13);
            char legend[51] =
{'L','E','N','|','T','Y','P','E','|','S','F','A','P','|','D','M'
,'|','S','M','|','S','Q','|','D','P','A','N','|','D','A','D','D'
,'|','S','P','A','N','|','S','A','D','D','|','P','a','y','l','o'
,'a','d'};
            int i = 0;
            for (i = 0;i<sizeof(legend);i++)
            {
                putu(legend[i]);
            }
            putu(10);
            putu(13);
        }
/////////
        // Device (payload[3]) measured ((payload[16] << 8)
+ payload[17]) feet.

        sprintf(dispChar, "%d", payload[3]);

        if (payload[3] < 10){
            dispChar[1] = dispChar[0];
            dispChar[0] = '0';
        }

        char legend2[20] = {'D','e','v','i','c','e','
',dispChar[0],dispChar[1],' ','m','e','a','s','u','r','e','d','
'};
        i = 0;
        for (i = 0;i<sizeof(legend2);i++)
        {
            putu(legend2[i]);
        }

        voltage = (payload[16] << 8) + payload[17];
```

```
            depth = ((30*(voltage / 432 * 5 - 0.5)/4) - 13.905)
* 74/3;
            depthInt = depth;
            sprintf(dispChar, "%d", depthInt);

            // print depth
            if (depthInt < 0){
                putu(48);
                putu(46);
                putu(48);
            }
            else {
                putu(dispChar[0]);
                putu(46);
                putu(dispChar[1]);
                putu(dispChar[2]);
                putu(dispChar[3]);
            }


/*
            if (depth > 1000){
                putu(dispChar[0]);
                putu(dispChar[1]);
                putu(46);
                putu(dispChar[2]);

            }
            else if (depth > 100){
                putu(dispChar[0]);
                putu(46);
                putu(dispChar[1]);
                putu(dispChar[2]);
            }
            else if (depth < 0){
                putu(48);
                putu(46);
                putu(48);
            }
            else {
                putu(48);
                putu(46);
                putu(dispChar[0]);
                putu(dispChar[1]);
            }
*/
            // feet of water.
```

```
putu(32);putu(102);putu(101);putu(101);putu(116);putu(32);putu(1
11);putu(102);putu(32);putu(119);putu(97);putu(116);putu(101);pu
tu(114);putu(46);

            putu(10);putu(13);

}

int write_frame_buffer(int payload[127]) {

    int rData, IRQdata;
    int i=0;

    // command write to TRX_STATE
    // write 0x09 (state change to PLL_ON)
    atrf_SPI(0xC2,0x09);

    LATBbits.LATB12 = 0;     // set chip select to 0 (active)

    rData = do_SPI(0x60);   // command write to frame buffer
    rData = do_SPI(payload[0]);   // send PHR (frame length) >=9
    rData = do_SPI(0x81);   // FCF first half
    rData = do_SPI(0x88);   // FCF second half
    rData = do_SPI(payload[3]);  // Sequence #
    rData = do_SPI(0xFE);   // Destination PAN
    rData = do_SPI(0xFE);
    rData = do_SPI(0xBA);   // Destination Address
    rData = do_SPI(0xDC);
    rData = do_SPI(0x54);   // Source PAN
    rData = do_SPI(0x76);
    rData = do_SPI(0x10);   // Source Address
    rData = do_SPI(0x32);

    //rData = do_SPI(0x0E);   // send 1
    //rData = do_SPI(0x0D);   // send 2
    //rData = do_SPI(0x0C);   // send 3
    //rData = do_SPI(0x0B);   // send 4
    //rData = do_SPI(0x0A);   // send 5
    //rData = do_SPI(0xAB);   // send 6
    //rData = do_SPI(0xCD);   // send 7
    //rData = do_SPI(0x0F);   // send 6
    //rData = do_SPI(0x01);   // send 7
    //rData = do_SPI(0x11);   // send 8
    //sendCount++;
```

```
    // Send message content
    i=12;
    while (i<(payload[0])){
//////////////////////
        rData = do_SPI(payload[i]);

        i++;
    }
    LATBbits.LATB12 = 1;     // set chip select to 1 (inactive)

    // command read from IRQ_STATUS
    // send empty byte, store status in IRQdata
    IRQdata = atrf_SPI(0x8F,0x00);
    //LATE = IRQdata;         // show IRQ_STATUS on LED array

    // Check for frame buffer problems
    if ((IRQdata & 0x40) == (0x40)){
        putu(10);
        putu(13);
        putu(69);
        putu(82);
        putu(82);
        putu(79);
        putu(82);
        //putu(0x6D); // put m if addresses match
    }

}

int send_Payload(int payload[127]){

    int IRQdata;
    int endTX=0, i=0;
    char dispChar[10];
    int device_ID = payload[3];

    // command write to TRX_STATE
    // write 0x02 (send TX_START)
    atrf_SPI(0xC2,0x02);

    // Wait until transmission finishes
    while (!endTX){
        IRQdata = atrf_SPI(0x8F,0x00);

        if ((IRQdata & 0x08) == (0x08)){
            endTX = 1;
```

```
        }
    }

            //oldseq = payload[3];



    //putu(10);
    //putu(13);

    sprintf(dispChar, "%d", device_ID);

    if (device_ID < 10){
        dispChar[1] = dispChar[0];
        dispChar[0] = '0';
    }

    char legend[45] = {'F','o','w','a','r','d','e','d',' '
','p','a','c','k','e','t',' ','w','i','t','h',' ','s','e','q',' '
',' ','n','u','m','b','e','r',':',' ',dispChar[0],dispChar[1],' '};
    i = 0;
    for (i=0;i<sizeof(legend);i++)
    {
        putu(legend[i]);
    }

    /*
    sprintf(dispChar, "%d", device_ID);

    if (device_ID < 10){
        dispChar[1] = dispChar[0];
        dispChar[0] = '0';
    }

    putu(dispChar[0]);
    putu(dispChar[1]);
    if (device_ID>99){
        putu(dispChar[2]);
    }
    if (device_ID>999){
        //putu(dispChar[3]);
    }*/

    //putu(10);
    //putu(13);
```

```
    //putu(10);
    //putu(13);

    /*
    char legend[29] = {'S','e','n','t','
','p','a','c','k','e','t',' ','n','u','m','b','e','r',':',' '};
    int l = 0;
    //for (l = 0;l<sizeof(legend);l++)
    //{
    //    putu(legend[l]);
    //}
    while (l<sizeof(legend)){
        //putu(legend[l]);
        l++;
    }
    */

    /*

    sprintf(dispChar, "%d", payload[3]);

    if (payload[3] < 10){
        dispChar[1] = dispChar[0];
        dispChar[0] = '0';
    }


    putu(dispChar[0]);
    putu(dispChar[1]);
    if (payload[3]>99){
        putu(dispChar[2]);
    }
    if (payload[3]>999){
        //putu(dispChar[3]);
    }

    //putu(10);
    //putu(13);

    */
    return;
}

int send_measurement(int device_ID, int waterLevel){

    int rData, IRQdata;
    int i=0, endTX = 0;
```

```
    char dispChar[10];

    // command write to TRX_STATE
    // write 0x09 (state change to PLL_ON)
    atrf_SPI(0xC2,0x09);

    LATBbits.LATB12 = 0;     // set chip select to 0 (active)

    rData = do_SPI(0x60);   // command write to frame buffer
    rData = do_SPI(0x13);   // send PHR (frame length) >=9
    rData = do_SPI(0x81);   // FCF first half
    rData = do_SPI(0x88);   // FCF second half
    rData = do_SPI(device_ID);  // Sequence #
    rData = do_SPI(0xFE);   // Destination PAN
    rData = do_SPI(0xFE);
    rData = do_SPI(0xBA);   // Destination Address
    rData = do_SPI(0xDC);
    rData = do_SPI(0x54);   // Source PAN
    rData = do_SPI(0x76);
    rData = do_SPI(0x10);   // Source Address
    rData = do_SPI(0x32);

    rData = do_SPI(0x23);
    rData = do_SPI(0x57);
    rData = do_SPI(0x11);
    rData = do_SPI(0x13);
    rData = do_SPI((waterLevel >> 8) & 0x03);
    rData = do_SPI(waterLevel & 0xFF);
    //rData = do_SPI(0x0E);   // send 1
    //rData = do_SPI(0x0D);   // send 2
    //rData = do_SPI(0x0C);   // send 3
    //rData = do_SPI(0x0B);   // send 4
    //rData = do_SPI(0x0A);   // send 5
    //rData = do_SPI(0xAB);   // send 6
    //rData = do_SPI(0xCD);   // send 7
    //rData = do_SPI(0x0F);   // send 6
    //rData = do_SPI(0x01);   // send 7
    //rData = do_SPI(0x11);   // send 8
    //sendCount++;



    // Send message content
    //i=12;
    //while (i<(payload[0])){
//////////////////////
    //    rData = do_SPI(payload[i]);
```

```
//
    //    i++;
    //}


    LATBbits.LATB12 = 1;     // set chip select to 1 (inactive)

    // command read from IRQ_STATUS
    // send empty byte, store status in IRQdata
    IRQdata = atrf_SPI(0x8F,0x00);
    //LATE = IRQdata;         // show IRQ_STATUS on LED array

    // Check for frame buffer problems
    if ((IRQdata & 0x40) == (0x40)){
        putu(10);
        putu(13);
        putu(69);
        putu(82);
        putu(82);
        putu(79);
        putu(82);
        //putu(0x6D); // put m if addresses match
    }



    // command write to TRX_STATE
    // write 0x02 (send TX_START)
    atrf_SPI(0xC2,0x02);

    // Wait until transmission finishes
    while (!endTX){
        IRQdata = atrf_SPI(0x8F,0x00);

        if ((IRQdata & 0x08) == (0x08)){
            endTX = 1;
        }
    }

            //oldseq = payload[3];




    putu(10);
    putu(13);
```

```
    char legend[51] = {'S','e','n','t','
','p','a','c','k','e','t',' ','w','i','t','h',' ','s','e','q','
','n','u','m','b','e','r',':',' '};
    i = 0;
    for (i = 0;i<sizeof(legend);i++)
    {
        putu(legend[i]);
    }


    sprintf(dispChar, "%d", device_ID);

    if (device_ID < 10){
        dispChar[1] = dispChar[0];
        dispChar[0] = '0';
    }

    putu(dispChar[0]);
    putu(dispChar[1]);
    if (device_ID>99){
        putu(dispChar[2]);
    }
    if (device_ID>999){
        //putu(dispChar[3]);
    }

    //putu(10);
    //putu(13);

}

int atrf_sleep(){

    int stateChange = 1,rData;

    // command write to TRX_STATE
    // write 0x08 (state change to TRX_OFF)
    atrf_SPI(0xC2,0x08);

    while(stateChange){
        // command read from TRX_STATUS
        // send empty byte, store status in rData
        rData = atrf_SPI(0x81,0x00);

        // when state changes to TRX_OFF, exit loop
        if ((rData & 0x1F) == 0x08){
            stateChange = 0;
```

```
        }
    }

    // put atrf212 to sleep
    LATBbits.LATB10 = 1;    // set SLP_TR to high

}

int atrf_wake(){

    int stateChange = 1,rData;

    // wake atrf212 up
    LATBbits.LATB10 = 0;    // set SLP_TR to low

    while(stateChange){
        // command read from TRX_STATUS
        // send empty byte, store status in rData
        rData = atrf_SPI(0x81,0x00);

        // when state changes to TRX_OFF, exit loop
        if ((rData & 0x1F) == 0x08){
            stateChange = 0;
        }
    }

    // command write to TRX_STATE
    // write 0x09 (state change to PLL_ON)
    atrf_SPI(0xC2,0x09);

    while(stateChange){
        // command read from TRX_STATUS
        // send empty byte, store status in rData
        rData = atrf_SPI(0x81,0x00);

        // when state changes to TRX_OFF, exit loop
        if ((rData & 0x1F) == 0x09){
            stateChange = 0;
        }
    }

}

int measure_Pressure(){

    float voltage, pressure, depth, depth_cm;
    int voltageInt, pressureInt, depthInt;
```

```
    int i=0;
    char dispChar[10];

    AD1CON1bits.ON = 1;          // turn ADC on


        i=0;
        voltage = 0;
        for(i=0;i<100;i++){
            AD1CON1SET = 0x0002; // start sampling ...
            delay_ms(10); // for 10 ms
            AD1CON1CLR = 0x0002; // start Converting
            while (!(AD1CON1 & 0x0001)){}// conversion done?
            voltage += ADC1BUF0; // yes then get ADC value


        }
        voltage = voltage/100;
        voltageInt = voltage;
        //voltage = voltage * 330 / 1023;
        //pressure = 30*(voltage)/2.66666;
        //depth = (pressure-1460) * 0.43351492;
        depth = ((30*(voltage / 432 * 5 - 0.5)/4) - 13.905) *
74/3;
        depthInt = depth;
        sprintf(dispChar, "%d", depthInt);



        putu(10);
        putu(13);
        char legend[51] = {'M','e','a','s','u','r','e','d',' '};
        i = 0;
        for (i = 0;i<sizeof(legend);i++)
        {
            putu(legend[i]);
        }

        if (depthInt < 0){
            putu(48);
            putu(46);
            putu(48);
        }
        else {
            putu(dispChar[0]);
            putu(46);
            putu(dispChar[1]);
```

```
            putu(dispChar[2]);
            putu(dispChar[3]);
        }



        //LATE = 0xFF - (voltageInt >> 2);




    // feet of water.

putu(32);putu(102);putu(101);putu(101);putu(116);putu(32);putu(1
11);putu(102);putu(32);putu(119);putu(97);putu(116);putu(101);pu
tu(114);putu(46);

        /*
        char legend2[20] = {'f','e','e','t',' ','o','f','
','w','a','t','e','r','.'};
        i = 0;
        for (i = 0;i<sizeof(legend2);i++)
        {
            putu(legend2[i]);
        }*/


    return voltageInt;
}
```