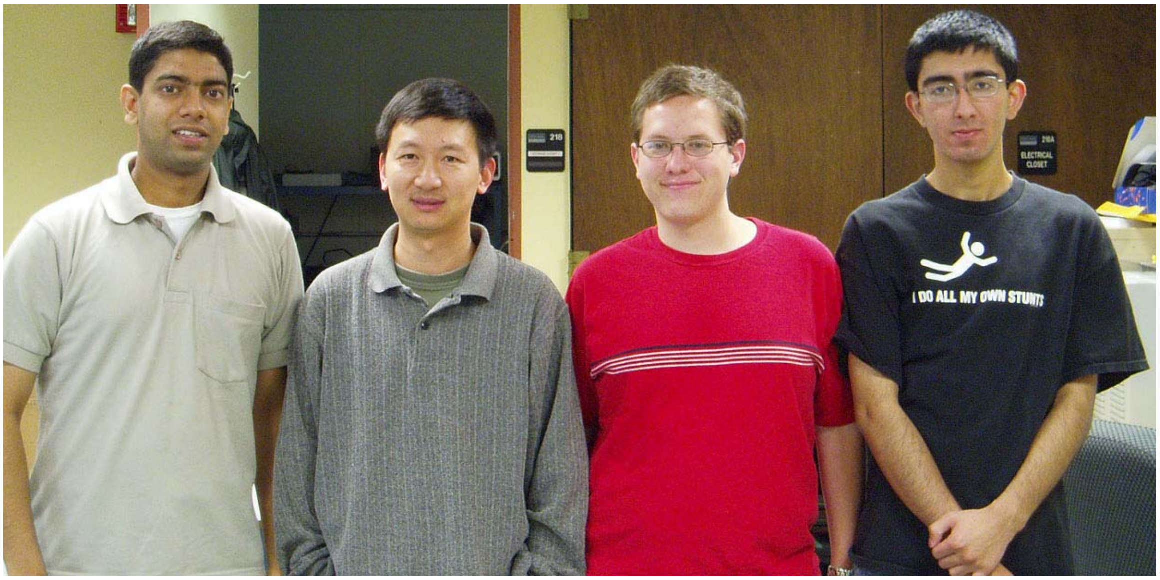
Wireless Drifter

Summer Scholars 2004



(From Left to Right)
Graduate Assistant: Nishant Kumar,
Advisor: Professor Yu-Dong Yao,
Robert Hudson,
Humza Shahid

Abstract

This report will describe the wireless drifter project done during the summer of 2004. It will contain information about the individual hardware components as well as the software written to control them. The software aspect will contain information about how the program was written as well as how it works. It will also explain the packaging of the drifter and the overall results of the project. The report will contain information on competitors' designs as well as the advantages of our design over theirs. A final section will include recommendations for future development of the project.

Table of Contents

Introduction

The research for summer scholars 2004 involved the development of a wireless drifter system. Wireless drifters can be used to study ocean currents and other trends as sea. The drifter developed consists of a GPS unit, a microcontroller, and a wireless transceiver. Together they can be used for scientific purposes.

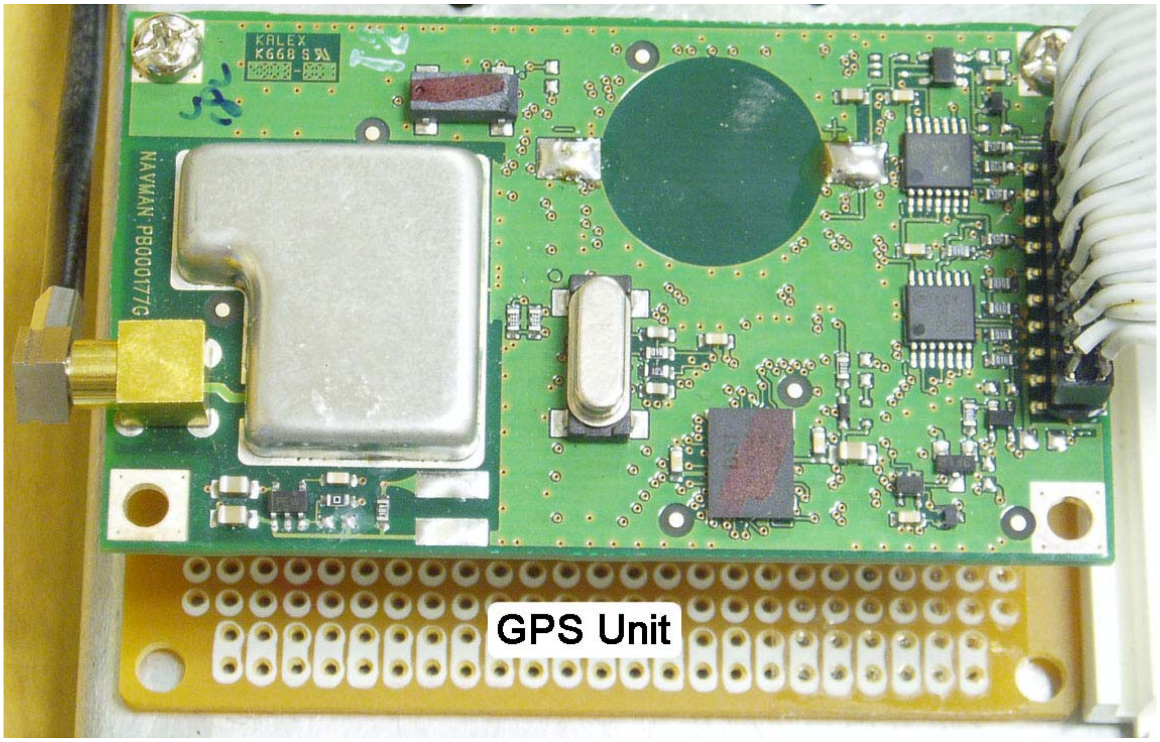
Our drifter was designed to be made less expensive than any existing drifters on the market with similar features and functions. Since we were using pre-made components, the primary task was to interface them together properly. Using less expensive components then allows us to market the product for a lower price than competitors. This low price point is important, because these drifters are meant to literally be thrown out to sea and may not be recoverable.

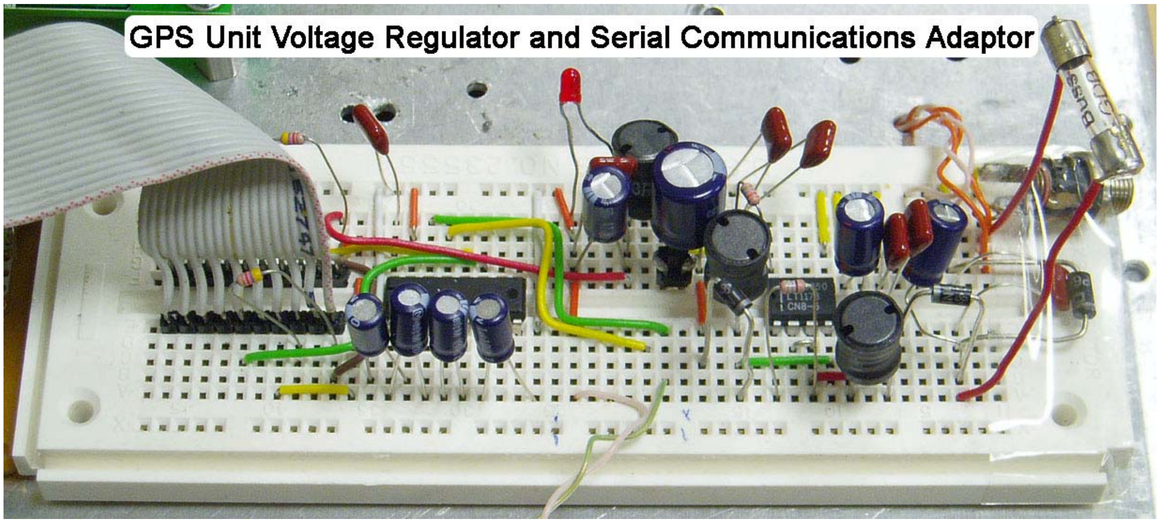
For a basic overview of how a wireless drifter works, there are two main components. The main body of the unit, which in our case contains the GPS unit, microcontroller, transceiver, and any sensors, is thrown out into the ocean and will be dragged along by the flow of the ocean currents. The microcontroller then gathers the data from the GPS unit and sensors and then creates a packet and then sends it via the transceiver back to the base. The second component of the drifter is the base unit. This unit consists of another wireless transceiver connected to a computer which decodes and logs the data from the packet.

The system developed includes both of the main units listed. The software for the microcontroller and the base computer interact to gather the data from the sensors and GPS unit. The software was designed to eventually allow input from more than one drifter at a time. This will allow study of larger areas of the ocean.

System and Hardware

GPS Unit


GPS Unit


GPS Unit Voltage Regulator and Serial Communications Adaptor

An integral part of the wireless drifter is the GPS unit. The GPS unit was the most difficult and time consuming attachment to program. The unit itself is not very large, and as with all other pieces requires a power source. It also needs an antenna so that it can communicate with satellites and determine its position, the time, and some other information. All of the data sent is from the GPS is in the format of “high byte, low byte.” It must reversed (low byte, high byte) before the computer can translate this hexadecimal data into binary. The unit was designed to transmit several different packets, all of which start with the message header “FF81.” The unit is able to send a variety of different codes which are shown in the following tables. For the purposes of the drifter, only the geodetic position information is needed. When programming the microcontroller to utilize the GPS, only the specific information that is necessary is stored and transmitted, the rest of the information is ignored. The data that we deemed important from the Geodetic Position Status Output are words 19-24, and 27-32. They are the date, time, and location data bits, which are the only pieces of information which the drifter needs from the GPS.

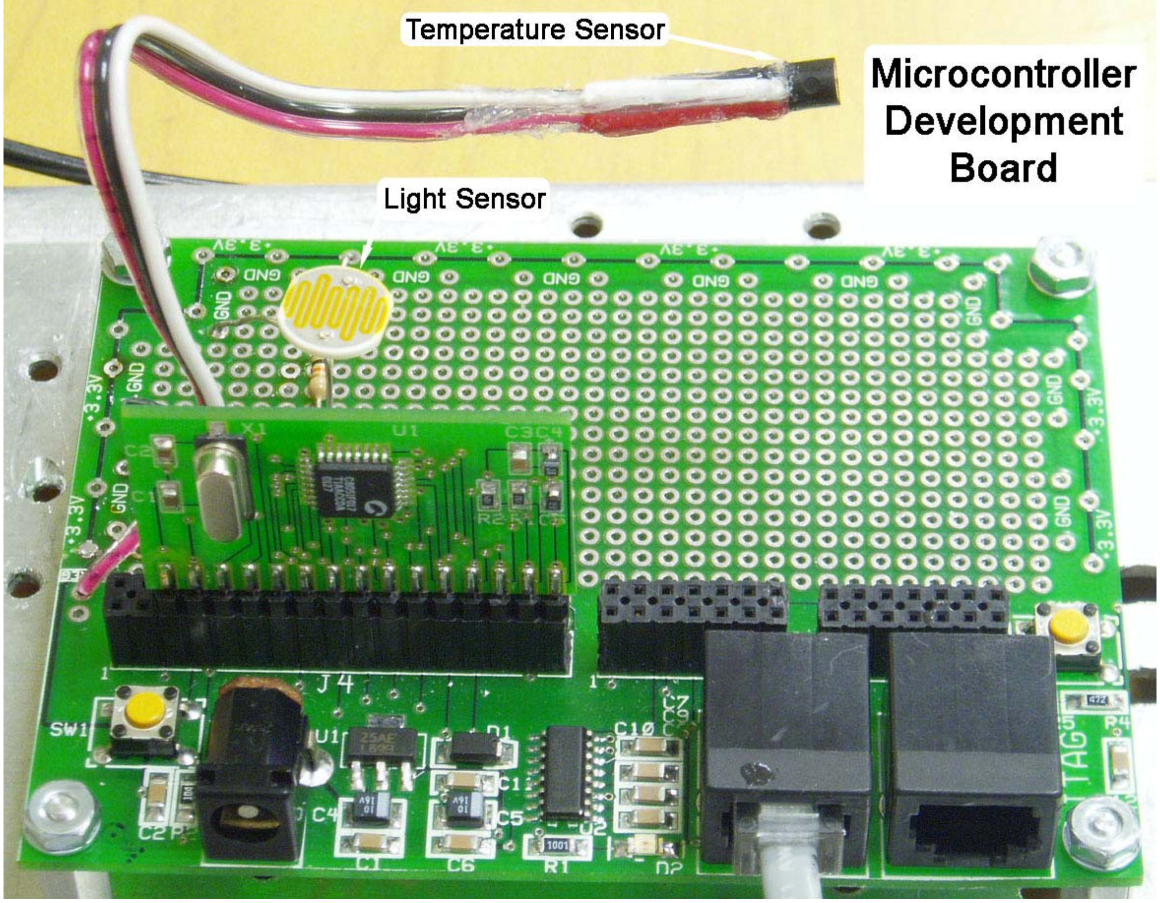
Output Message Name	Message ID	Input Message Name	Message ID
Geodetic Position Status Output (*)	1000	Geodetic Position and Velocity Initialization	1200
Channel Summary (*)	1002	User-Defined Datum Definition	1210
Visible Satellites (*)	1003	Map Datum Select	1211
Differential GPS Status	1005	Satellite Elevation Mask Control	1212
Channel Measurement	1007	Satellite Candidate Select	1213
ECEF Position Output	1009	Differential GPS Control	1214
Receiver ID (**)	1011	Cold Start Control	1216
User-Settings Output	1012	Solution Validity Criteria	1217
Built-In Test Results	1100	User-Entered Altitude Input	1219
UTC Time Mark Pulse Output (*)	1108	Application Platform Control	1220
Frequency Standard Parameters In Use	1110	Nav Configuration	1221
Power Management Duty Cycle In Use	1117	Perform Built-In Test Command	1300
Serial Port Communication Parameters In Use	1130	Restart Command	1303
EEPROM Update	1135	Frequency Standard Input Parameters	1310
EEPROM Status	1136	Power Management Control	1317
Frequency Standard Table Output Data	1160	Serial Port Communications Parameters	1330

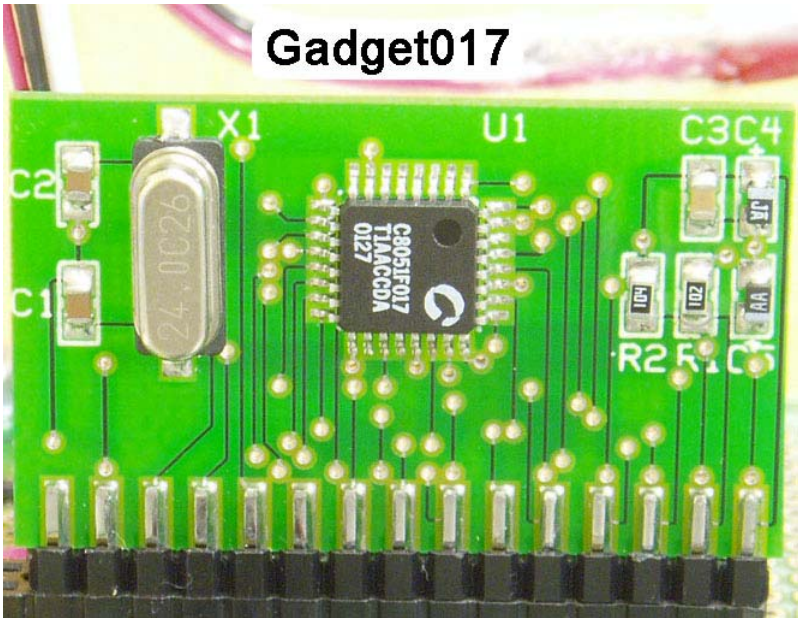
Boot Status	1180	Factory Calibration Input	1350
Error/Status	1190	Frequency Standard Table Input Data	1360
		Message Protocol Control	1331
		Raw DGPS RTCM SC-104 Data	1351
		Flash Reprogram Request	1380
(*) Enable by default at power-up (**) Once at power-up/reset			

Message ID:	1000				
Rate:	Variable; defaults to 1 Hz				
Message Length:	55 words				
Word No.:	**Name:**	**Type:**	**Units:**	**Range:**	**Resolution:**
1-4	Message Header				
5	Header Checksum				
6-7	Set Time (Note 1)	UDI	10 msec ticks	0 to 4294967295	
8	Sequence Number (Note 2)	I		0 to 32767	
9	Satellite Measurement Sequence Number (Note 3)	I		0 to 32767	
Navigation Solution Validity (10.0-10.15)					
10.0	Solution Invalid - Altitude Used (Note 4)	Bit		1 = true	
10.1	Solution Invalid - No Differential GPS (Note 4)	Bit		1 = true	
10.2	Solution Invalid - Not Enough Satellites in Track (Note 4)	Bit		1 = true	
10.3	Solution Invalid - Exceeded Maximum EHPE (Note 4)	Bit		1 = true	
10.4	Solution Invalid - Exceeded Maximum EVPE (Note 4)	Bit		1 = true	
10.5	Solution Invalid - No DR Measurements (Note 5)	Bit		1 = true	
10.6	Solution Invalid - No DR Calibration (Note 6)	Bit		1 = true	
10.7	Solution Invalid - No Concurrent DR Calibration by GPS (Note 7)	Bit		1 = true	
10.8-10.15	Reserved				
Navigation Solution Type (11.0-11.15)					
11.0	Solution Type - Propagated Solution (Note 8)	Bit		1 = propagated	
11.1	Solution Type - Altitude Used	Bit		1 = altitude used	
11.2	Solution Type -Differential	Bit		1 = differential	
11.3	Solution Type - PM	Bit		1 = RF off	
11.4	Solution Type - GPS (Note 9)	Bit		1 = true	
11.5	Solution Type - Concurrent GPS Calibrated DR (Note 10)	Bit		1 = true	
11.6	Solution Type - Stored Calibration DR (Note 11)	Bit		1 = true	
11.7-	Reserved				

11.15					
12	Number of Measurements Used in Solution	UI		0 to 12	
13	Non-DR Link: Polar Navigation DR Navigation Link: Bit 0 = Polar Navigation Bit 15 to 1 = Hading Uncertainty Standard Deviation (Note 12)	Bit Bit UI	degrees	1 = true 1 = true 0 to 300	0.01
14	GPS Week Number	UI	weeks	0 to 32767	
15-16	GPS Seconds From Epoch	UDI	seconds	0 to 604799	
17-18	GPS Nanoseconds From Epoch	UDI	nanosec	0 to 999999999	
19	UTC Day	UI	days	1 to 31	
20	UTC Month	UI	months	1 to 12	
21	UTC Year	UI	year	1980 to 2079	
22	UTC Hours	UI	hours	0 to 23	
23	UTC Minutes	UI	minutes	0 to 59	
24	UTC Seconds	UI	seconds	0 to 59	
25-26	UTC Nanoseconds From Epoch	UDI	nanosec	0 to 999999999	
27-28	Latitude	DI	radians	±0 to p/2	10_{-8}
29-30	Longitude	DI	radians	±0 to p	10_{-8}
31-32	Height	DI	meters	±0 to 50000	10_{-2}
33	Geoidal Separation	I	meters	±0 to 200	10_{-2}
34-35	Ground Speed	UDI	meters/sec	0 to 1000	10_{-2}
36	True Course	UI	radians	0 to 2p	10_{-3}
37	Magnetic Variation	I	radians	±0 to p/4	10_{-4}
38	Climb Rate	I	meters/sec	±300	10_{-2}
39	Map Datum (Note 13)	UI		0 to 188 and 300 to 304	
40-41	Expected Horizontal Position Error (Note 14)	UDI	meters	0 to 320000000	10_{-2}
42-43	Expected Vertical Position Error (Note 14)	UDI	meters	0 to 250000	10_{-2}
44-45	Expected Time Error (Note 14)	UDI	meters	0 to 300000000	10_{-2}
46	Expected Horizontal Velocity Error (Note 14)	UI	meters/sec	0 to 10000	10_{-2}
47-48	Clock Bias (Note 14)	DI	meters	±0 to 9000000	10_{-2}
49-50	Clock Bias Standard Deviation (Note 14)	DI	meters	±0 to 9000000	10_{-2}
51-52	Clock Drift (Note 14)	DI	m/sec	±0 to 1000	10_{-2}
53-54	Clock Drift Standard Deviation (Note 14)	DI	m/sec	±0 to 1000	10_{-2}
55	Data Checksum				

Microcontroller





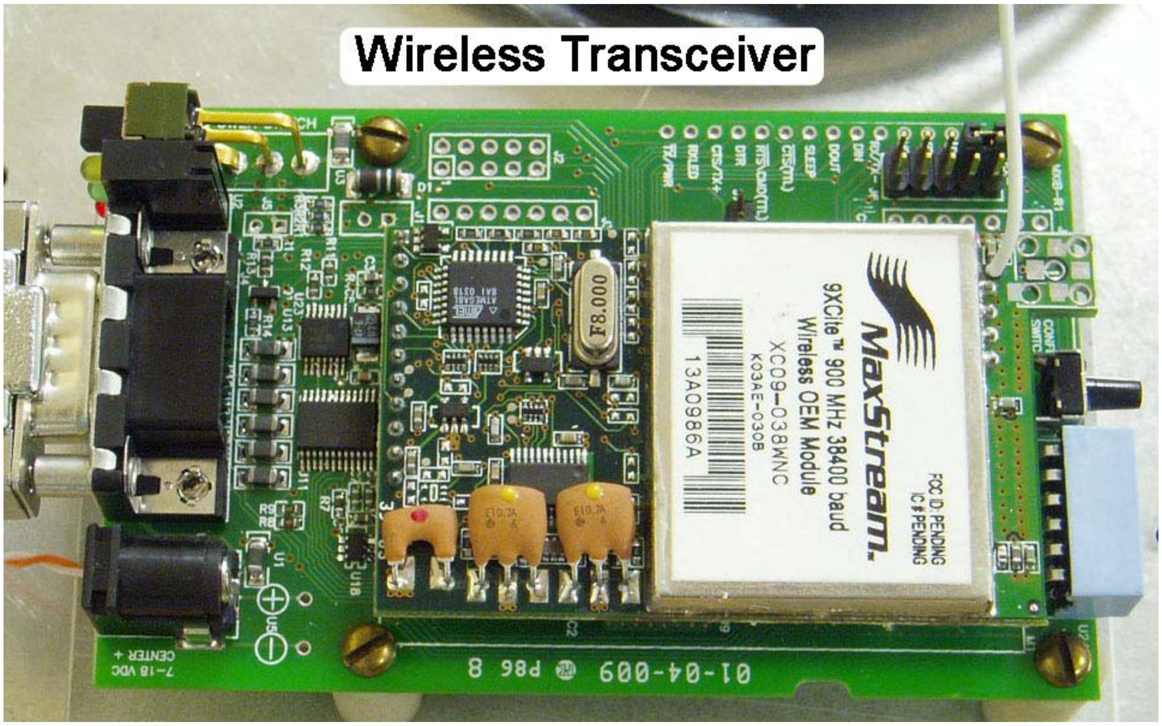
The microcontroller used for the wireless drifter is the Gadget017 from AMResearch. This microcontroller was chosen because of its many useful features, for a relatively inexpensive price. We also have another microcontroller we could use, the Gadget300, which is basically a slightly more powerful version of the Gadget017. If necessary in the future, it is possible to switch to the Gadget300 fairly easily.

The microcontroller contains a Cygnal C8051F017 8051 RISC chip at its heart. This chip allows most instructions to be performed in one clock cycle, making this a very fast chip running at 25 MHz. The chip is based off of the popular 8051 architecture. The Gadget017 has 2304 bytes of internal data RAM, and 32,000 bytes of external Flash RAM. This large amount of RAM is useful for storing and then processing the data that will be gathered by the wireless drifter. It also has 8 input/output pins, which can be used for the various sensors to be attached to the microcontroller. The I/O pins can also be read through a 10-bit analog to digital converter, so that the microcontroller can store data from the various pins.

Another advantage of the Gadget017 is the free development environment provided by AMResearch. The development environment includes an editor, compiler, and programmer, so all of the needed development tools were included free of charge, and included as an open source package, so the source code of the tools could easily be modified for any necessary changes. The chip and IDE (Integrated Development Environment) also allow for great flexibility in the programming, due to the support of three languages: Assembler, Forth, and Basic. The IDE also allows for interpretative execution, so that the code can be tested in real time on the microcontroller, without any emulators or other software. This normally allows for easy debugging, however when a program is compiled as a standalone program (compiled as a turnkey program) the interactivity is decreased.

For more information on the Gadget017 and its related software, read its datasheet at www.amresearch.com.

Wireless Transceiver



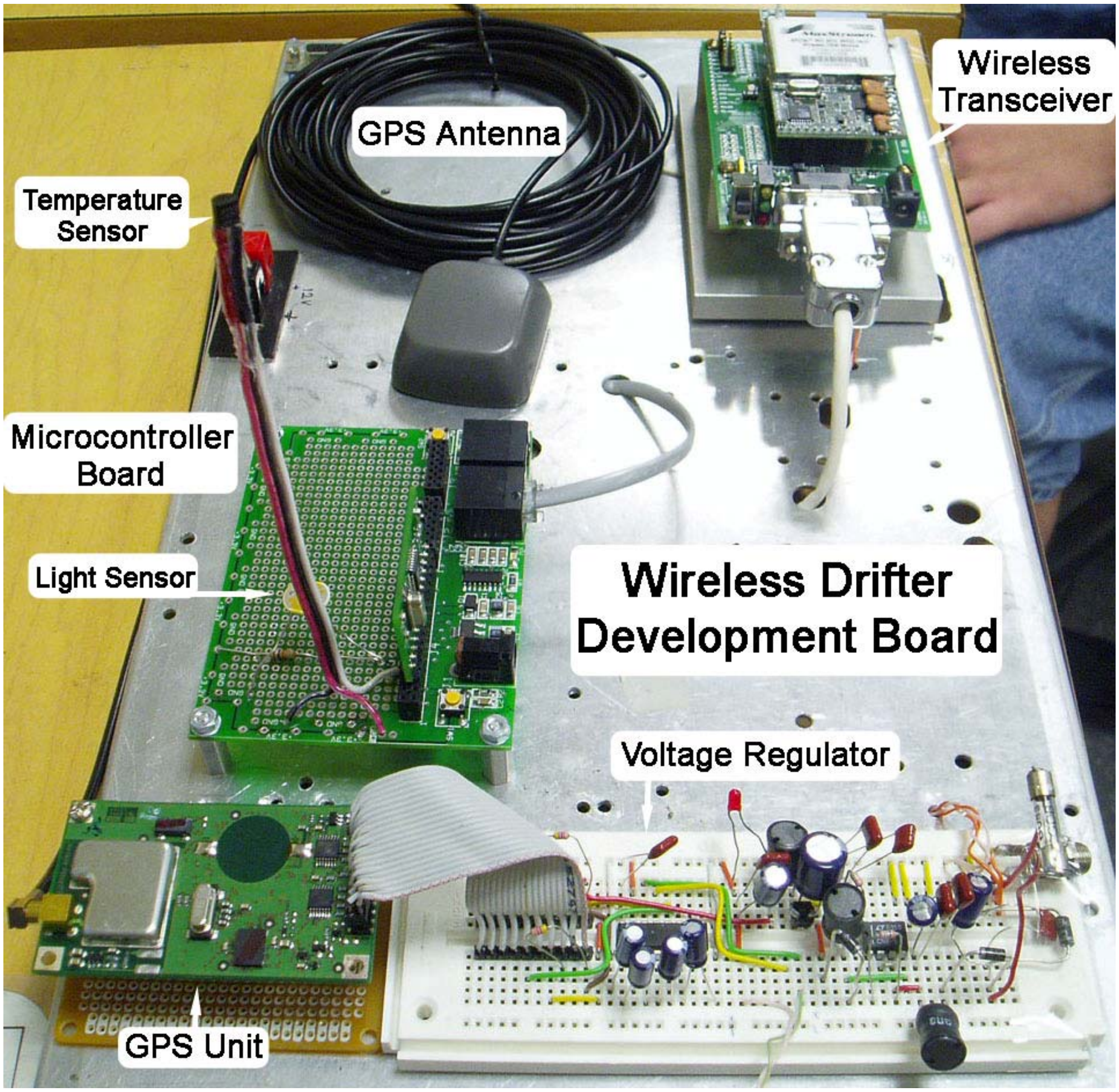
The wireless transceiver is another integral part of the wireless drifter. This is the component which allows the drifter to transmit data to a pc so that it can be interpreted. The particular chip used is the Maxstream 9XCite (www.maxstream.net) which supports the following features, as well as many others:

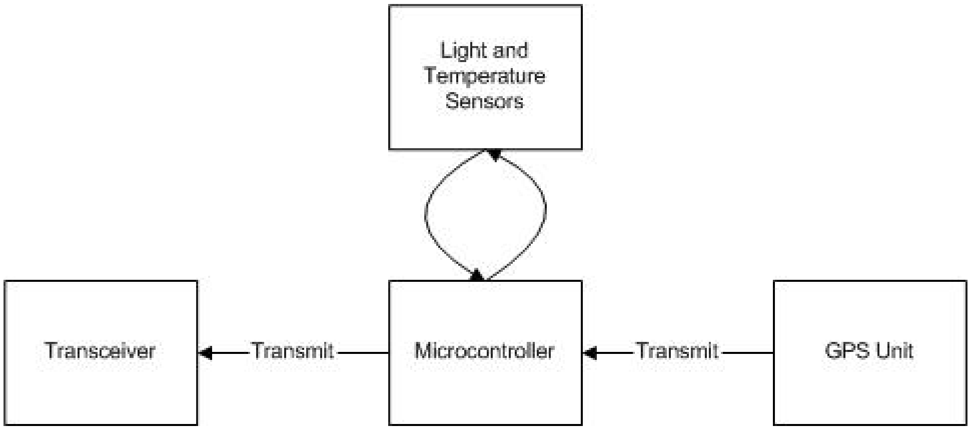
- Plug-and-communicate (default mode - no configuration required).
- True peer-to-peer network (no need to configure a "Master" radio).
- Transparent mode supports existing software applications and legacy systems.
- Addressing capabilities provide for point-to-point and point-to-multipoint networks.
- Uses Standard AT commands and/or fast binary commands for changing parameters.
- Native RS485/422 (multi-drop bus) protocol support.
- Retry and acknowledgements of packets provides guaranteed delivery of critical packets in difficult environments.
- Networking features allow up to 7 independent pairs (networks) to operate in close proximity.
- Multiple low power modes including shutdown pin, cyclic sleep and serial port sleep for current consumption as low as 20 µA.
- Host interface baud rates from 1200 to 57600 bps.

- Signal strength register for link quality monitoring and debugging.
- Parity support (None, Even, Odd, Mark, Space)
- 9-bit support

The transceiver has a very simple and straightforward purpose. It takes the data sent by the microcontroller and broadcasts it on a wireless frequency so that it can be received and decoded by specially designed software. Another possible way to transmit data would be to have the drifter send information to a satellite which would relay the information to a receiver/pc, however that would be much more costly and not very reliable when it cloudy or dark.

Integration



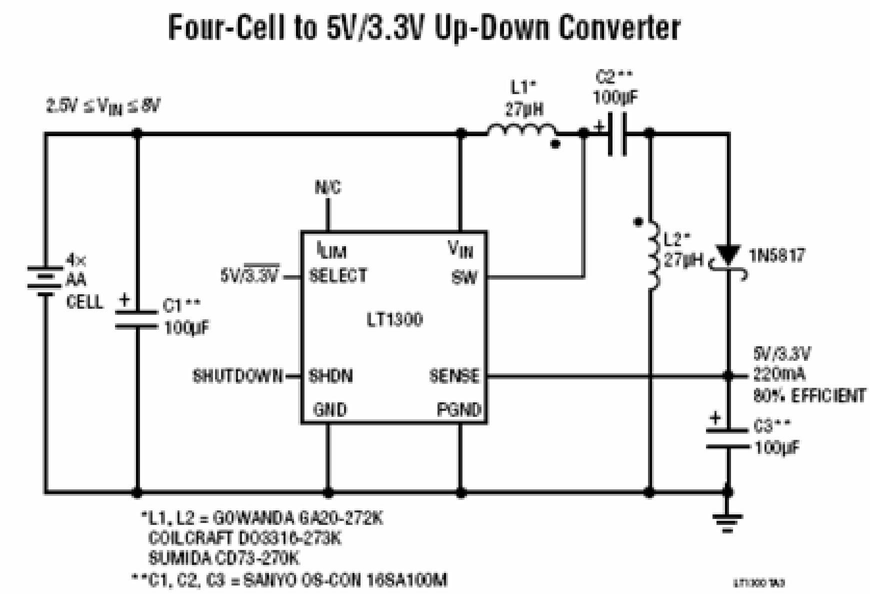


Once all of the components needed for the wireless drifter were gathered, we had to combine them all together. The components were linked together by the serial communications ports they each had. The transmit line on the GPS unit was connected to the receive line on the microcontroller, then the transmit line on the microcontroller was connected to the receive line on the transmitter. The microcontroller also had a temperature sensor and a light sensor attached to it. The temperature sensor is a LM34 Precision Fahrenheit Temperature Sensor from National Semiconductor, and the light sensor is simply a photo resistor from Radio-Shack.

The way that the components had been connected had some advantages and some disadvantages. The main advantage was that this method did not require any multiplexing of the serial lines, since the GPS module only transmitted to the microcontroller, and the microcontroller only transmitted to the wireless transmitter. The main disadvantage is that since all the communication is one way, everything must be pre-programmed into the microcontroller, since there is no way to send it data other than from the GPS module.

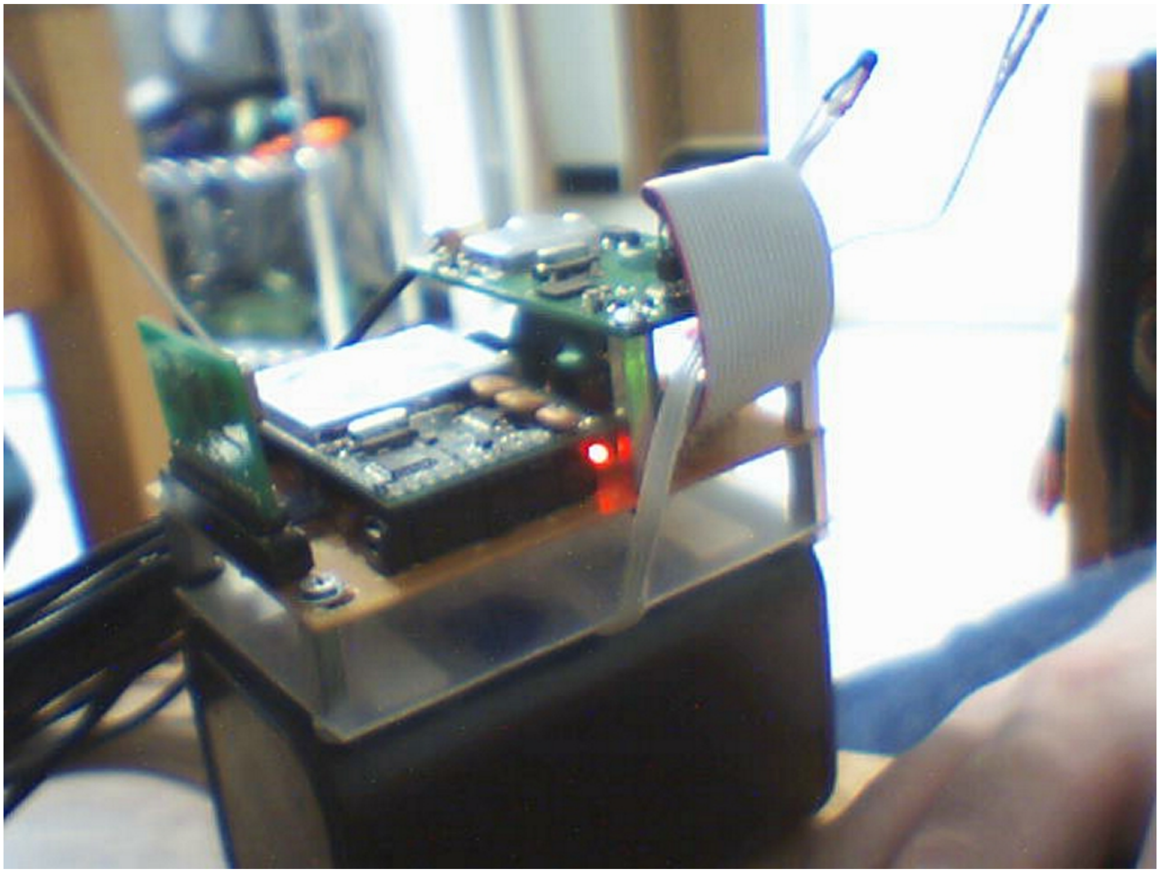
For development purposes, the drifter was assembled on a board that allowed us to switch from a direct serial connection to the microcontroller to the complete drifter unit. This allowed us to easily reprogram the microcontroller without rewiring the entire system.

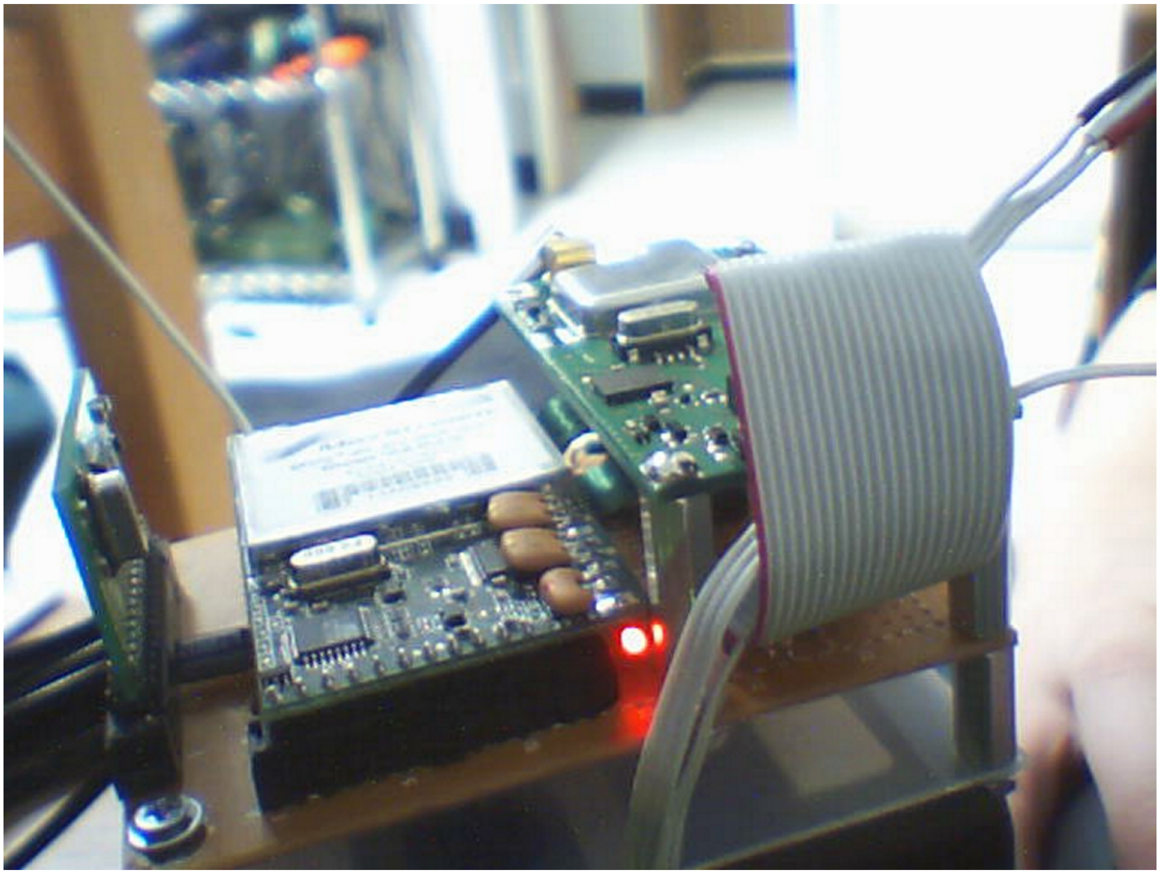
Once the development board was completed, and our program was finalized, a prototype was built. The three main components, the microcontroller, the GPS unit, and the transceiver, were integrated onto one circuit board. The microcontroller was programmed on the development board, and then it was just plugged into the socket made for it on the prototype. The prototype was powered by a six volt lantern battery using this circuit as a power regulator:

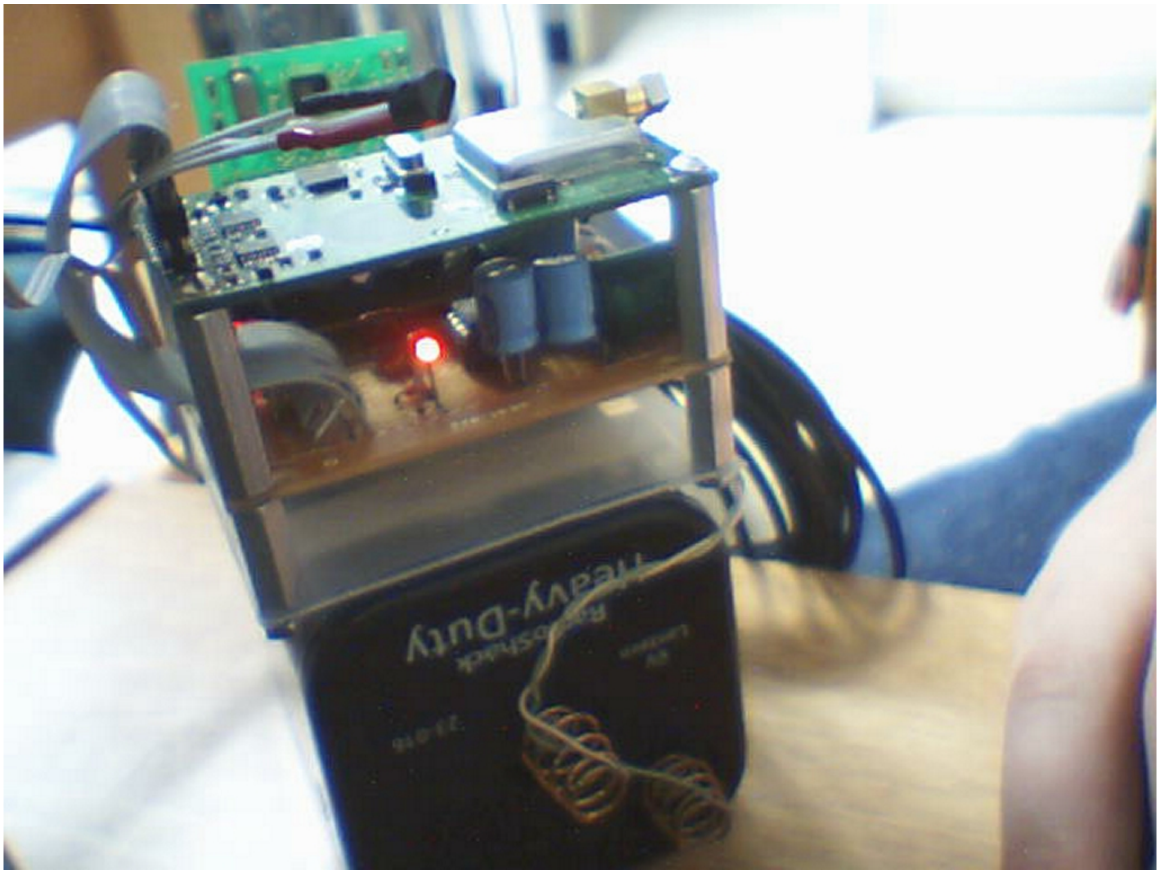


Four-Cell to 5V/3.3V Up-Down Converter

Here are a few pictures of the completed prototype:

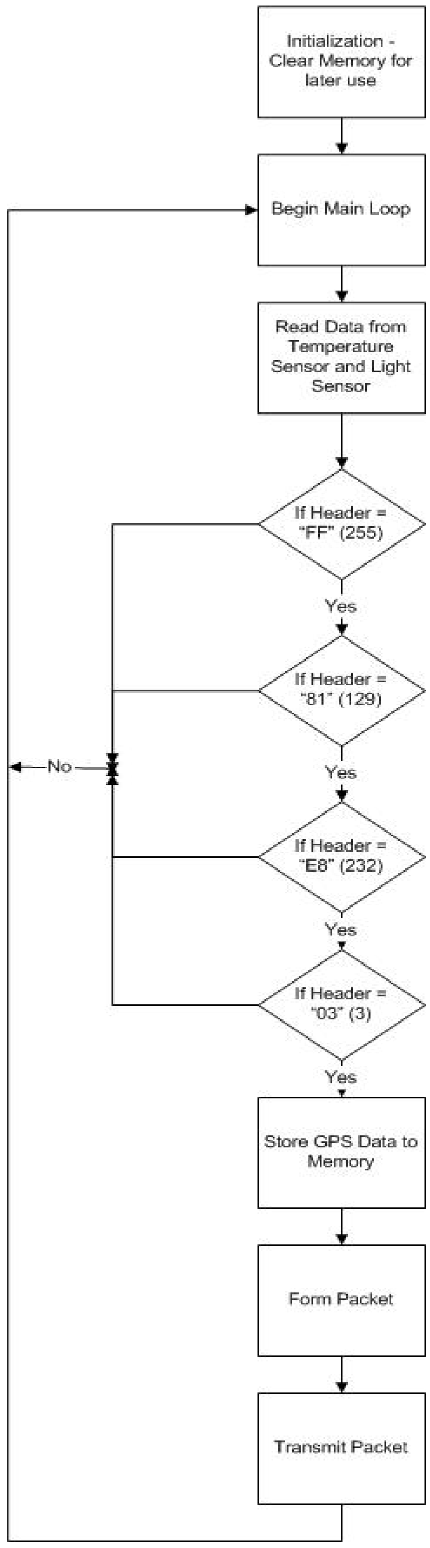





Heavy-Duty
RadioShack

Software

Microcontroller



When we started programming the microcontroller, we initially wanted to use Basic, since it was a high level language that would allow us to easily write our program, without worrying about the underlying hardware of the microcontroller. However, we quickly learned that the Basic implementation provided by AMResearch was not complete. The implementation did not allow us to access memory locations within the microcontroller, which was necessary to allow us to create a packet of all the gathered data. Another limitation was that all the variables that were to be used had to be declared in the basic.fs file (the forth implementation of the basic language given to us by AMResearch). Due to these limitations, we were forced to use Forth to program the microcontroller.

Forth is a very powerful language. It allows for high level programming, but with low level access. It even allows you to use in-line assembly code within your program, making the possibilities almost endless. The major drawback is that it is completely stack orientated, so the programmer has to be very careful of the hardware's current settings and stack operations. Therefore, it actually allows you to program at a high level, but you must monitor the low level results of your code. Because of the stack orientation, all operations in forth are in postfix notation (AKA reverse Polish notation). This was a very different concept for us, so we had to learn not only the forth language, but also how to do operations in postfix.

As stated in the preceding paragraph, the main program was written in Forth, using the AMRForth program provided by the manufacturer of the microcontroller. The program was written in its built-in editor, and compiled as a turnkey program. Compiling as a turnkey program allows the program to be executed, starting at the "go" statement, continually, without interaction from the computer. If the program is compiled normally, the microcontroller is treated as a tethered computer, meaning it will not function correctly without input from the computer.

Minimal modifications were made to the kernel and library provided by the software, but the few that were made were necessary. In the kernel, four lines were removed, since they were causing problems with the "key" function used to read data over the serial port on the microcontroller. In the file "kernel8051.fs" (the actual kernel of the microcontroller's operating system) lines 829 through 832 were commented out. Those lines were the redefinitions of "key" and "key?", which were removed as a suggestion by one of the programmers at AMResearch, after we contacted them when we were having problems with the key function.

The library, file "adc017.fs," was modified to allow use of the built in temperature sensor, and to allow for easy initialization of the other analog input pins contained on the microcontroller. The modified file's source code is included in the appendix of this paper. We changed the init files so that we could use two pins. One init function, "init-adc-temp" initialized the pin for the temperature sensor, while another, "init-adc-ain0" allowed us to use the light sensor. We then modified the "adc10@" function, which gave us the value received from whatever sensor the microcontroller was initialized to.

The microcontroller's program consisted of two main sections. Since it was compiled as a turnkey program, the "go" function was always started automatically. Right after the program started, it initialized itself, which is the first main section. Initialization took place in lines 4 to 22, as shown in the appendix for the file "drifter.fs" (the name given to our program file). After initialization, the program entered the main loop, which was an infinite loop by design. We made it an infinite loop, since we wanted the program to run continuously without any interaction when it was actually in use. The loop is the longest section of code, since it occurs in lines 23 to 121.

Before explaining the procedures the program goes through, it would be best to explain the various commands used in forth. As stated before, Forth is written in postfix notation, so keep that in mind when using the commands. Since one of the main uses of the program is to store and translate data, memory access is a widely used command in the program. The operators "!d" and "c!d" are used to store data to an external memory location. They store two bytes and one byte respectively. The operators "@d" and "c@d" read from an external memory location. As before, "@d" reads two bytes, while "c@d" reads only one. Another widely used command is "emit" which sends the top of the data stack (one byte) as an ASCII character out via the serial port. This is used to transmit the assembled packet. One last main function is the "key" function. It reads one character over the serial port and adds it to the top of the data stack. All the other functions we used are simple for-next loops and if statements.

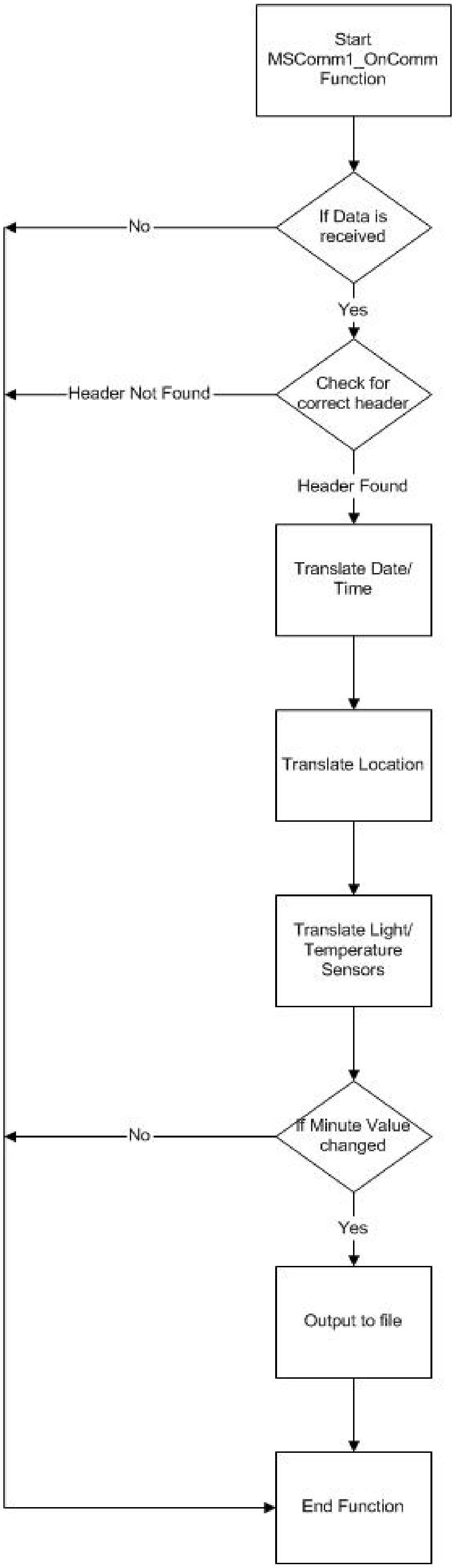
The initialization stage of the program consists of clearing the memory locations used for storing the raw data from the GPS unit, and the locations used for the assembled packet. It also clears out the memory location used as a counter in the for-next loops in the main loop.

The main loop starts off by taking the measurements from the temperature sensor and the light sensor, and placing the data directly into the packet, so that whenever the packet is fully assembled, it can be sent without waiting for that data. The program then goes through a nested set of if statements, while it looks for the Geodetic Position Status

Output header from the GPS unit. The microcontroller reads the headers one byte at a time using the "key" function, lines 29 to 36. The header for the Geodetic Position Status Output in decimal is "255,129,232,3" so once it finds that exact sequence, the program starts to take the raw position data from the GPS unit and stores it temporarily in memory, lines 40-44.

After completely storing the raw GPS data, the program starts to create the packet. The header code we picked to use was equal to "AB CD" in hexadecimal (171, 205 in decimal), lines 47 and 48. This value is arbitrary; we just picked something that would be easy to remember and not as likely to be repeated in the position statement. The next value inserted into the packet is the drifter number, lines 49 and 50. This has to be hard coded into each drifter for the time being. It is also in the low-byte/high-byte form, since that is how the Visual Basic program we wrote reads the information. Next comes the GPS time/date information, along with the location information, lines 52 to 78. Since the raw data was stored completely in the microcontroller's memory, we simply matched up the locations between the raw data, and where the packet was to be stored. For example, the value for the day information from the GPS unit was stored in memory locations 332 and 333. We then took the value stored in location 332 and transferred it to the location of the day information in the packet, which was 2004, and we did the same for 333, moving it to location 2005. We did this for the rest of the needed information. After the packet was created, we transmitted the packet using the "emit" command for each byte in the packet, lines 81 to 113. Finally, we inserted a pause, so that the program would save some battery life, by only transmitting at a certain period. To change the interval, simply change the numbers in the for-next statements in line 115. Each number can only be two bytes in length, so having two nested statements was necessary to give the correct delay. After the delay, the program loops again to continue gathering data.

Computer


Start MSComm1_OnComm Function
If Data is received
No
Yes
Check for correct header
Header Not Found
Header Found
Translate Date/ Time
Translate Location
Translate Light/ Temperature Sensors
If Minute Value changed
No
Yes
Output to file
End Function

The software written for a computer to read the data transmitted from the microcontroller was written using Microsoft visual basic 6. The program was based upon our initial GPS data reading program. After heavy modifications the program was able to work in conjunction with the microcontroller to a near perfect degree of functionality.

The first few tasks were simple. After creation of the form, the MSComm1 control was created and its values were initialized to 9600 kbps, 8 data bits, no parity, and 1 stop bit. The RThreshold is set to 30 bytes, the length of the packet that the microcontroller sends. It is designed so that once the entire packet is in the serial port buffer the OnComm event will be triggered. The buffer is cleared and then the port is opened to receive data as soon as the program is started. At this time, we have the program open a simple text file “log.txt” which will be used for saving a simply formatted version of the data it receives every minute. In future versions the user will most likely be given the option to choose the location and name of the file where the data will be saved. A true-false value is also set when the form is loaded. This value will help notify the user if there is an error in making a connection to the wireless drifter.

A timer is set which will change the true-false value if there is no data received within a certain amount of time. This value is dependent on how frequently the microcontroller sends data, and as such, must be modified according to how frequently the microcontroller sends data. For this program we set the timers length to the time it would take to receive two packets of data from the microcontroller.

The packet sent by the microcontroller, which has been designed by us, is designed to send us all of the information needed in as little space as possible. The packet consists of a header (ABCD), drifter ID number, and then it contains GPS data, temperature sensor data, and light sensor data. The data is received in high byte then low byte, which must be reversed before it is useful to us. The data is in hex, which is translated by a function called “nHexToDec” which was obtained online. All credit for writing this function goes to the author, Cristian Calugar. We decided to use this function instead of write our own due to time constraints. It would be much quicker to use an already created function than to learn how to do the translation in visual basic ourselves.

The most important function of the software, however, is one called “getData.” The getData function is used throughout the entire program to read information from the serial port buffer so that it can be translated into data that is useful to us. Because of the nature of VB, get data had to be specially modified so that even when the data was preceded by a zero (such as 01), the zero would not be cut off as VB normally does. To fix this issue, VB is told to check the length of the string. If it is a length of one, it adds a zero to the beginning of the string. If it is a length of two then the string is left alone. GetData also performs another key function; it reverses the order of the bytes so that they can be translated from hex to decimal.

The program has been designed to wait for data to be received from the serial port before it takes any action except for its error checking function which is always running. When data is found on the serial port buffer, the program goes into action and starts to translate the information which has been received. It first searches for the message header “AB CD”. Once the header has been found it will take the next two bytes as the drifter ID number, and then it knows that the next twelve bytes are the GPS data which correspond to date and time. The next twelve bytes are probably the most important bytes in the entire program. These bytes give the location of the drifter as provided by the GPS unit. They are given in the order of Latitude, Longitude, and then Height. The temperature sensor data is given next, and then finally the light sensor. Latitude, Longitude, and Height are all four bytes each, and must be read using 2’s complement. The temperature sensor needs a simple equation to calibrate it properly, and the light sensor is fine as is.

Once all of the data has been obtained from the microcontroller and translated, it is then stored in the log file (currently log.txt) in the following format:

Drifter ID Number
Date
Time
Latitude
Longitude
Height
Temperature
Light Intensity

The programs GUI was designed to be small and compact, but at the same time, full of information. All of the information obtained from the drifter unit is displayed on screen in an easily readable fashion. The small screen area allows for many more additions to be added in the future, and as most computers run only moderately high resolutions, it will not look too small on your average pc. The error message is a sort of popup box which will be displayed in the center of the main program screen if there is no connection with the drifter.

Lastly, the exit button has been coded so that it will make sure the serial port is closed (so that no problems will occur with the pc after the software is shut down). Clicking on the x in the top left corner will do the same job as clicking exit, as both have been coded to do the same thing.

Packaging

Enclosure

The enclosure for the drifter unit will be designed to be waterproof. We intend to make it clear because in the future, solar power will most likely be used to power the drifter and extend its lifetime. The overall size will be small to save costs and allow the drifter to move with the current more freely. A "tail" will be attached to the drifter which will hang underwater allowing the drifter to flow with the underwater currents instead of the surface current.

Power

The drifter will be powered by an alkaline lantern battery, because we will be including a voltage regulation circuit so that each component will get the voltage it needs. In future models, a solar panel will be used to charge a battery, so that the drifter will have a longer lifetime.

Test Results

Using the assembled prototype, the system was given a test run. We walked around campus with the prototype and a laptop with the receiver attached to it. The laptop then logged all the data from the prototype drifter. This test had mixed results. The temperature sensor and light sensor worked perfectly, as cloud cover and shade changed the light sensor's readings, and sun and breezes changed the temperature sensors values. The GPS location was almost perfect. The latitude and longitude changed correctly as we moved, however the height values were incorrect. The height was reported as a negative value for the majority of the readings. This is most likely caused by a small bug in the translation software; however there is also a chance that the GPS unit had just not calibrated correctly.

As this was the first actual test of the unit, it performed better than expected. The prototype functioned correctly, with no apparent problems. Only some minor bug fixes are needed in the software, which should not take very long.

Competitors' Drifter Designs

The purpose of this research was to develop a wireless drifter that could be marketed successfully against other competing designs. Our goal was to produce a design that had all of the features of the more expensive drifters at a fraction of the cost. Of course it’s not possible to have every feature of the more expensive models, but it still has similar functionality.

The highest end drifters can cost approximately $4500. These drifters are much larger than our design, and also more powerful. They use the ARGOS satellite system to transmit and receive data; therefore they can be used on a more global scale than our current design. However, due to their larger size they are more difficult to disperse and collect.

Another type of drifter, which is closest to our design, was developed at Texas A&M University-Corpus Christi. It was assembled for approximately $500. It has the same functions as our design, with the exception of the light and temperature sensors we have included. They used a more expensive microcontroller than us, causing the increase in price. The expensive microcontroller allowed programming in C, which made programming slightly easier, but all the functions can still be made in forth.

The drifter we created has most of the functions of all the more expensive drifters, but for only a fraction of a price. We should be able to produce and sell the drifters for about $300. This price advantage almost makes the drifters expendable, unlike the $4500 ones. This also makes the technology accessible to more consumers, and allows even more research on ocean currents to be performed. The drifter also can be upgraded in the future, since it is made of off-the-shelf components. Therefore, our goals for the creation of a marketable wireless drifter have been met.

Recommendations

The following is a list of possible changes that could be made to enhance the usefulness of the wireless drifter.

- Allow multiplexing in serial communications so that on the fly reprogramming can be done.
- Allow more than one drifter to communicate with each other so that only one would have to transmit to the base, allowing for the “transmitter” to be equipped with a more powerful antenna.
- Add a checksum to the packet to ensure accuracy of the data received.
- Encrypt data to ensure privacy.
- Use alternate power source, such as solar power.
- Make modular code so that new functions can be added easily.
- Include a larger size memory unit such as a mini-hard drive or flash drive to store data for longer periods of time.
- Use a higher power antenna so that longer transmit ranges are available.
- Add other sensors such as pollution and sound sensors.
- Develop another application (or an add-on to the current one) which takes the GPS data and plots it on a map, for a visual representation of the drifter's movements.

Source Code

Microcontroller Program (Drifter.fs)

```
\ drifter.fs

: go
    \ Clear memory locations that will be used
    0 260 !d
    106 for
        0 300 260 @d + c!d

        260 @d 1 + 260 !d
    next

    \ init-adc-ain1

    0 260 !d
    28 for
        0 2000 260 @d + c!d

        260 @d 1 + 260 !d
    next

    0 260 !d \ reset counter

    begin
        init-adc-ain1
        adc10@ 2028 !d
        init-adc-ain0
        adc10@ 2030 !d

        key
        255 = if
            key
            129 = if
                key
                232 = if
                    key
                    3 = if

                    0 260 !d

            \ This for loop reads data from the gps and stores it to memory
                    106 for
                        key 300 260 @d + c!d
        260 @d 1 + 260 !d
next

\ This is the packet header
171 2000 c!d \ AB
205 2001 c!d \ CD
1 2002 c!d   \ These two lines are the drifter ID #
0 2003 c!d

\ This is where the GPS time/date data is stored
332 c@d 2004 c!d \ day
333 c@d 2005 c!d
334 c@d 2006 c!d \ month
335 c@d 2007 c!d
336 c@d 2008 c!d \ year
337 c@d 2009 c!d
338 c@d 2010 c!d \ hour
339 c@d 2011 c!d
340 c@d 2012 c!d \ minute
341 c@d 2013 c!d
342 c@d 2014 c!d \ second
343 c@d 2015 c!d

\ This is where the actual location data is stored
348 c@d 2016 c!d \ latitude
349 c@d 2017 c!d
350 c@d 2018 c!d
351 c@d 2019 c!d
352 c@d 2020 c!d \ longitude
353 c@d 2021 c!d
354 c@d 2022 c!d
355 c@d 2023 c!d
356 c@d 2024 c!d \ height
357 c@d 2025 c!d
358 c@d 2026 c!d
359 c@d 2027 c!d


\ this is the actual transmitting of the packet
2000 c@d emit
2001 c@d emit
2002 c@d emit
2003 c@d emit
2004 c@d emit
2005 c@d emit
2006 c@d emit
```

```
                    2007 c@d emit
                    2008 c@d emit
                    2009 c@d emit
                    2010 c@d emit
                    2011 c@d emit
                    2012 c@d emit
                    2013 c@d emit
                    2014 c@d emit
                    2015 c@d emit
                    2016 c@d emit
                    2017 c@d emit
                    2018 c@d emit
                    2019 c@d emit
                    2020 c@d emit
                    2021 c@d emit
                    2022 c@d emit
                    2023 c@d emit
                    2024 c@d emit
                    2025 c@d emit
                    2026 c@d emit
                    2027 c@d emit
                    2029 c@d emit \ temp data in low/high form
                    2028 c@d emit
                    2031 c@d emit \ light data in low/high form
                    2030 c@d emit

                    500 for 1000 for noop next next
                    then \ end of if 3
                then \ end of if 232
            then \ end of if 129
        then \ end of if 255

    again
-;
```

GPS-MC Translator Program

```
Option Explicit

Dim blnError As Boolean
Dim intCounter As Integer

Private Sub cmdExit_Click()
    Unload Me
End Sub

Private Sub Form_Unload(Cancel As Integer)
    MSComm1.PortOpen = False                  ' Closes the port
    Close #1
    Unload Error
End Sub

Private Sub Form_Load()                         ' MSComm component initialization
    MSComm1.RThreshold = 30
    MSComm1.CommPort = 1                        ' Sets the port to be used to COM1
    MSComm1.InputLen = 1                        ' Sets the input length to 1 character
    MSComm1.InputMode = comInputModeText        ' Sets the input type to text
    MSComm1.Settings = "9600,n,8,1"             ' Sets the baud to 9600, no parity, 8
                                                ' data bits, 1 stop bit
    MSComm1.PortOpen = True                     ' Opens the port
    MSComm1.InBufferCount = 0                   ' Makes sure the buffer is empty
                                                ' before data is received

    blnError = True
    Error.Visible = False
    Open "log.txt" For Append As #1
    intCounter = -1
End Sub

Private Function getData() As String        ' This function reads from the GPS unit
                                            ' and translates the data stream into hex
    Dim str1 As String, str2 As String      ' it takes in 2 characters (4 hex symbols) and
                                            ' reverses them, so that the data can be
                                            ' translated

    str1 = Hex$(Asc(MSComm1.Input))
    If Len(str1) = 1 Then
        str1 = "0" + str1
    End If
    str2 = Hex$(Asc(MSComm1.Input))
    If Len(str2) = 1 Then
        str2 = "0" + str2
    End If
    getData = (str2 + str1)
End Function

Private Sub Timer1_Timer()
    If blnError Then
        Error.Visible = True
    End If
End Sub

Private Sub MSComm1_OnComm()

    If MSComm1.CommEvent = comEvReceive Then
        blnError = False
        Error.Visible = False

        Dim strHex As String
        Dim strHex2 As String
        Dim strTest As String
        strTest = MSComm1.Input

        If strTest = "" Then
            Error.Visible = True
        Else
            Error.Visible = False
            strHex = Hex$(Asc(strTest))             ' takes the received data, turns it into its
                                                    ' ascii value, then into its hex value

            If Len(strHex) = 1 Then
                strHex = "0" + strHex
            End If
            If strHex = "AB" Then
                strHex2 = Hex$(Asc(MSComm1.Input))
                If Len(strHex2) = 1 Then
                   strHex2 = "0" + strHex2
                End If

                If strHex2 = "CD" Then
                    Dim intMinute As InputModeConstants

                    lblDrifter.Caption = nHexToDec(getData())
                    lblDay.Caption = Format(nHexToDec(getData()), "#00")
                    lblMonth.Caption = Format(nHexToDec(getData()), "#00")
                    lblYear.Caption = Format(nHexToDec(getData()), "#00")
                    lblHour.Caption = Format(nHexToDec(getData()), "#00")

        intMinute = nHexToDec(getData())

        lblMinute.Caption = Format(intMinute, "#00")
        lblSecond.Caption = Format(nHexToDec(getData()), "#00")

        Dim dblLat As Double
        Dim dblLon As Double
        Dim dblHeight As Double

        Dim s1 As String
        Dim s2 As String
        Dim s As String

        s1 = getData()
        s2 = getData()
        s = s2 + s1

        If nHexToDec(s) < 2147483647 Then      ' words 27 and 28
           dblLat = (nHexToDec(s) / 100) * (180 / 3.141592654)
        Else
           dblLat = -1 * ((42949672.95 - (nHexToDec(s) / 100) + 0.01) * (180 /
3.141592654)) 'words 27-28
        End If

        dblLat = dblLat / 1000000

        s1 = getData()
        s2 = getData()
        s = s2 + s1

        If nHexToDec(s) < 2147483647 Then       'words 29 and 30
           dblLon = (nHexToDec(s) / 100) * (180 / 3.141592654)
        Else
           dblLon = -1 * ((42949672.95 - (nHexToDec(s) / 100) + 0.01) * (180 /
3.141592654)) 'words 29-30
        End If

        dblLon = dblLon / 1000000

        s1 = getData()
        s2 = getData()

        s = s2 + s1

        If nHexToDec(s) < 2147483647 Then       'words 31 and 32
           dblHeight = nHexToDec(s) / 100
```

```
                    Else
                        dblHeight = -1 * (42949672.95 - (nHexToDec(s) / 100) + 0.01)
                    End If

                    lblLat.Caption = Format(dblLat, "#.0000")
                    lblLon.Caption = Format(dblLon, "#.0000")
                    lblHeight.Caption = Format(dblHeight, "#.00")


                    Dim strTemp As String
                    Dim intTemp As Integer
                    Dim strLux As String
                    Dim intLux As Integer

                    strTemp = getData()
                    intTemp = nHexToDec(strTemp)
                    lblTemp = Format((intTemp + 87) / 5.2, "#.00")

                    strLux = getData()
                    intLux = nHexToDec(strLux)
                    lblLux = Format(1023 - intLux, "#.00")

                    If intCounter <> intMinute Then
                        intCounter = intMinute
                        Print #1, lblDrifter.Caption + vbCrLf + lblMonth.Caption + "/" +
         lblDay.Caption + "/" + lblYear.Caption + vbCrLf + lblHour.Caption + ":" +
         lblMinute.Caption + ":" + lblSecond.Caption + vbCrLf + lblLat.Caption + vbCrLf +
         lblLon.Caption + vbCrLf + lblHeight.Caption + vbCrLf + lblTemp.Caption +
         vbCrLf + lblLux.Caption + vbCrLf
                    End If 'intCounter <> intMinute
                End If 'CD
            End If 'AB
        End If 'strTest
        Timer1.Enabled = False
        Timer1.Enabled = True
    End If 'MSComm1.CommEvent
    blnError = True
End Sub

'***********************************************************************
' Name: Convert hex to decimal (big values)
'
' Description:Convert big hex values to decimal numbers.
'
' By: Cristian Calugar
'
```

```
' Inputs:Hex value (up to 12 bytes).
'
' Returns:Decimal number (up to 29 digits)
'
'
'This code is copyrighted and has' limited warranties. Please see http://w
'     ww.Planet-Source-Code.com/vb/scripts/Sho
'     wCode.asp?txtCodeId=49991&lngWId=1'for details.
'**********************************************************************


Public Function nHexToDec(sSource As String) As Variant
    '*****************************************************************
    ' Name: nHexToDec
    ' Author : Cristian Calugar
    ' Comments:
    ' Max. accurate is nHexToDec(string(24,"F"))
    '
    ' Result: 79,228,162,514,264,325,024,342,547,895
    '*****************************************************************
    Dim i As Integer, iVal As Byte

    If (Trim$(sSource) = "") Then
        nHexToDec = 0
        Exit Function
    End If

    nHexToDec = 0


    For i = 1 To Len(sSource)
        iVal = CByte("&h" & Mid$(sSource, i, 1))
        nHexToDec = CDec(nHexToDec + iVal * 16 ^ (Len(sSource) - i))
    Next i

End Function
```

Glossary

Forth – The programming language used to program the Microcontroller. Forth uses postfix for carrying out operations because it works around a central “stack” which contains the data. As such, the addition of “2 + 1” would be written “2 1 +”

Gadget017 – The name of the microcontroller used in the wireless drifter described in the paper. It contains a Cygnal C8051F017 microprocessor based on the 8051 architecture.

GPS Unit – A receiver that takes signals from the Global Positioning System satellites that orbit the Earth and translates them into the location of the unit. This system can pinpoint the latitude, longitude, and height of the unit with an accuracy of a few meters.

Microcontroller – An embedded processor that is used to collect and analyze data. The one used in the wireless drifter described in this paper is the Gadget017. A microcontroller contains the software to control the unit, along with the processor to make decisions based on the information gathered.

Transceiver – A piece of equipment that sends and receives signals over a wireless frequency.

Visual Basic – A powerful and robust programming language developed by Microsoft. Called VB for short, it allows programmers to quickly and easily create in-depth graphical user interfaces so that more time can be spent to develop functions and features in the code.

Wireless Drifter – The focus of this paper. The one described in the paper consists of a GPS unit, a microcontroller, and a transceiver. The microcontroller gathers data from the various onboard sensors and the GPS unit, and transmits the collected data via the transceiver.

References

http://www.maxstream.net

http://www.amresearch.com

http://www.conexant.com

http://msdn.microsoft.com/vbasic

http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=49991&lngWId=1

http://www.silabs.com/products/microcontroller/mixsig_matrix.asp

http://www.national.com/pf/LM/LM34.html#Datasheet

http://vathena.arc.nasa.gov/curric/oceans/drifters/drifters.html

http://www.cbi.tamucc.edu/Publications/Proceedings/Perez_etal%20IEEE_CMTC.pdf