

Wright State University

Implementation of a Wildfire 5282 Microcontroller for a
High Altitude Balloon Package

WSU High Altitude Balloon Program

Team Members:

Ryan Back

Jon Baumann

Michael Walters

June 15, 2009

1. Proposal

1.1 Purpose

The purpose of this document is to propose an idea that will fulfill the requirements for a Senior Design Project for CEG 498 Winter, 2009.

1.2 Background

We will be working closely with other members of the Wright State High Altitude Balloon team to enhance their existing balloon package by integrating a new hardware/software system that will be used to control various aspects of the balloon package before and during flight.

1.3 Scope

The proposed project will consist of developing a basic understanding of the existing balloon systems, particularly the basic stamp that is being used on the package and the various functions that it controls. We will then be replacing the basic stamp with a Wildfire 5282 microcontroller and implementing it to perform the necessary functions that are required by other systems on the balloon.

1.4 Personnel

Personnel Include:

Ryan Back	rulevoid@gmail.com
Jon Baumann	baumann.7@wright.edu
Michael Walters	walters.40@wright.edu

Also contributing will be other members of the 2008-2009 High Altitude Balloon Team. Names and contact information can be found here:

http://www.cs.wright.edu/balloon/index.php/2008-2009#Senior_Design_Students

1.5 Facilities and Equipment

The equipment that will be used to complete our project will be provided to us by the WSU High Altitude Balloon Team. We will also be utilizing necessary tools in the lab located in the basement of the Russ Engineering Center at Wright State University. Access to the lab is provided to us by the Mechanical and Materials Engineering Department at Wright State.

The attached “Hardware Analysis” describes, in more detail, the specific hardware that we will be implementing and provides a basic synopsis for the cost analysis of this hardware.

1.6 Costs

Costs include basic supplies, tools, and materials to construct, develop, and maintain the High Altitude Balloon Package. Additional costs include travel expenses and labor for man-hours spent working on the project.

More specifically, our individual team will be looking at the costs of the Wildfire 5282 Microcontroller board and development kit: \$399.00 ea. Two of these boards have already been purchased by the Balloon Team and will be implemented as the primary goal of our project.

1.7 Time

Our team will spend approximately 30-50 man-hours per week during the course of the next 14 weeks to complete the project. Time spent may vary depending on any changes that are made or if specific challenges or problems arise.

1.8 Summary

The primary goal of this Senior Design Project is to successfully implement a new hardware/software system that will control certain functionality of a High Altitude Balloon package. The system will consist of one or more Wildfire 5282 Microcontrollers and will carry out any tasks that are needed by other systems on the balloon.

To determine the overall success of our project, the balloon will be launched (at least once) and the system will be tested for proper functionality during flight.

2. Hardware Analysis

2.1 Background

BASIC Stamp 2p 24-Pin Module with Super Carrier Board



(pictures not to scale)

Cost:	Basic Stamp:	\$79.00
	Carrier Board:	\$19.00
	Total:	\$98.00

BASIC Stamp 2p 24-Pin Module Technical Specifications:

- Processor Speed = 20 MHz Turbo
- Program Execution Speed = ~12,000 instructions/sec.
- RAM Size = 38 Bytes (12 I/O, 26 Variable)
- EEPROM (Program) Size = 8 x 2K Bytes, ~4,000 instructions
- I/O Pins = 16 +2 Dedicated Serial
- Voltage Requirements = 5 - 12 vdc
- Current Draw at 5V = 40 mA Run / 350 μ A Sleep
- PBASIC Commands = 61
- Size = 1.2"x0.6"x0.4"

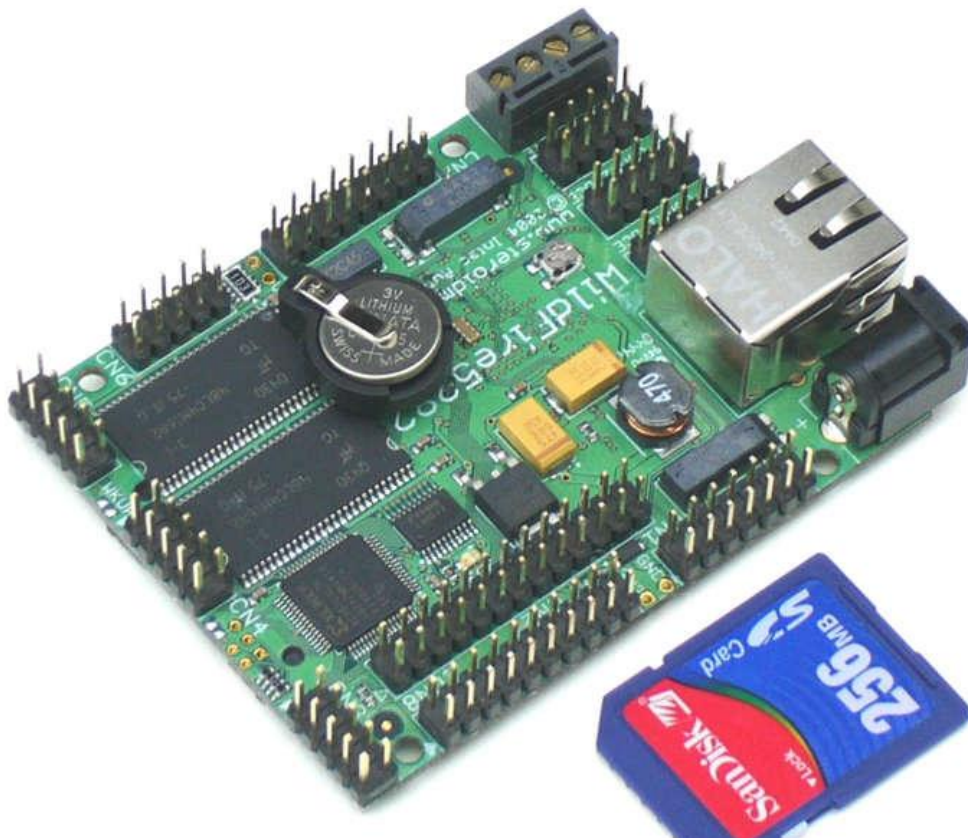
Super Carrier Board Features:

- 14-pin SIP socket and 24-pin DIP socket supports multiple microcontrollers (BS1-IC, BS2-IC, JS1-IC, etc.)
- 3-pin programming connection (for use with BASIC Stamp 1 Serial Adapter #27111)
- Serial DB9 connector (for use with 24-pin BASIC Stamps or Javelin Stamp)
- 9-volt battery clips and power jack for flexible power supply options
- Solder-hole area (1.5" x 2.0") arranged to easily support servos, 300 and 600 mil DIP ICs, DB9, DB25, RJ-11 connectors.
- Reset Button

Super Carrier Board Technical Specifications:

- Power Requirements: 6 - 30 vdc (6 to 15 vdc recommended)
- Communication: Serial
- Dimensions: 4.00L x 3.00W x 0.65H in (10.2L x 7.62W x 1.65H cm)
- Operating Temperature: -40° to +185° F (-40° to 85° C)

WildFire 5282 Microcontroller



Cost: \$199.00

Microprocessor

- 64MHz FreeScale Integrated ColdFire Version 2 Microcontroller

Memory

- 512K fast on-chip FLASH EPROM
- 64K fast on-chip SRAM
- 2 - 4 MB flash memory for program storage
- 6 MB fast SDRAM for program execution
..Ample memory forLinux, datalogging as well as debugging in SRAM

SD Card Socket

- 16 MB - 1GB removable SD memory card (opt.) or 802.11 adapter card
..for obscene amounts of non-volatile memory and/or WiFi and Bluetooth SDIO cards

Timer Port (2x10 header)

- 16 Input capture or output compare (also GP I/O)
..for stepper motor control, D/A conversion

Interrupt Port (2x5 header)

- 7 Edge or level interrupt pins (also GP I/O)
..for instant response to critical events

A/D Port (2x8 header)

- 8 highly programmable Analog Inputs
- 10 bit, 140KHz sampling rate
..for input from microphones, analog sensors, thermistors pressure transducers, etc.

3 Configurable I/O Ports (3-2x5 headers)

- 3 x 8 general purpose I/O pins, independently configurable as either an input or output
..to drive relays, sense switches

Debug Port

- Supports the P&E ICDPPC In-Circuit Emulator/ Debug pod.
..for the ultimate in debug control
- Can alternate as 9 gen purpose I/O

Ethernet Port (RJ45 Socket)

- 10/100 BaseT capability, half or full duplex
..for high speed internet connectivity, send emails, serve web pages, ftp files, etc.

3 Serial Ports (3-2x5 headers)

- 3 UARTs with RS232 level signals: 2 DCE + 1 DTE
..Connect to host PC, high speed modem, instruments, etc

CAN 2.0B Port (screw terminal)

- CAN Port with transceiver - to 1Mbps
- Full implementation of CAN 2.0A and 2.0B
..for robust networking in noisy environments

LCD/Keypad Port (2x7 header)

- Signals to drive character based LCDs (including contrast pot.) and signals to interrogate up to 16 contact keypads
..for user interactivity in the field

Additional Timers

- 4 Periodic Interrupt Timer
..to trigger periodic events
- SWT (Software Watchdog Timer) *..to recover from software lockup*

Clock/Calendar Option

- Battery backed clock/calendar and hibernation circuit
..to turn power off and restore power seconds to a week later

Power

- 6 - 24 vdc unregulated input - 2.1mm center
- Switching power supply on-board
..accepts a wide range of input voltages

Synopsis

Although the Wildfire microcontroller costs twice as much as the Basic Stamp system currently in use by the WSU High Altitude Balloon Team, we feel that it will provide a much more reliable and upgradeable system. Once the Wildfire board is programmed to perform the functionality of the current system, additional functionality can easily be implemented to meet the needs of the project. The Wildfire board has very large amounts of flash memory as well as ram compared to the Basic stamp, so running additional software will not slow down performance as it would on the existing system. This microcontroller provides a wide array of features for a reasonable price, and is a major advance for the future of the WSU HAB team.

2.2 Design

Our final configuration of the hardware for the final launch planned for June 18, 2009 will consist of two separate wildfire boards that will be sent up. There will be one board in the top package and one board in the bottom package. The board in the top package will be receiving data from two separate GPS units. Both of these GPS units will be powered from an independent power supply and not from the wildfire board itself. The wildfire will be collecting data from both GPS units and formatting this data into custom packet sentences. These packets will then be sent to a separate Terminal Node Controller which will convert the digital data into audio tones which will be sent to a transceiver module which will transmit them over HAM radio frequencies.

For power, the wildfire will be supplied with 12V DC from an independent battery. The wildfire is also connected to a common ground with the GPS units and the power source.

The second wildfire in the bottom package will be laid out in a similar fashion. However, there will only be a single GPS unit connected to this wildfire. Instead of transmitting the packets via a standalone transceiver, we will be using a Kenwood radio model: THD7A. In addition to the GPS unit and Kenwood radio, we will have two separate connections being made to the wildfire board. One connection will be made to a separate circuit which will control the release of the parachute for the package. This connection is an output from the wildfire board and will push a

high signal to trigger the parachute release at the proper time. The second connection is an input to the wildfire board which will be coming from another circuit attached to the Dual Tone Multi-Frequency board. This will be triggered high or low when a cutdown/separation signal is sent to the balloon package. This signal is further analyzed in the code and will be discussed in the software analysis.

The exact voltage has yet to be determined for the wildfire's power supply, but it will be connected to a power bus which will provide it with the proper amount of power. It will also be connected to a common ground along with the power supply the GPS unit and the Kenwood radio.

2.3 Simple Hardware Diagram for Bottom Package



2.4 Specific Pin Connections

Top Package:

COM1 pin5 wired to the transmit on GPS 1

COM2 pin5 wired to the transmit on GPS 2

COM3 pin5 wired to the receive on the TNC Board

Both GPS units and the TNC/transceiver must be connected to common ground with the wildfire.

Power: 6v~24v DC

Bottom Package:

COM2 pin5 - wired to the transmit on the GPS

COM3 pin5 - wired to the receive on the Kenwood Radio model THD7A

(Note: the Kenwood radio has a 2.5mm plug for the PC input. The other end of this cable will have to be specially wired to connect to the wildfire)

CN5 pin5 - This is the input pin which will be triggered when a signal is sent to the DTMF signaling a cut down/release

CN3 pin8 - This is the output pin which will output 3.3V to trigger the parachute deployment.

Power: 6v~24v DC

3. Software Analysis

3.1 Design

The code for the wildfire is written in C and is an entirely structural design. Many variables are used throughout the code and are initialized globally. There are a number of functions which perform individual routines which are called throughout the code to perform specific operations.

The program begins by initializing all of the ports that are using serial communication. UART is the type of serial communication being utilized and documentation on proper use of this feature can be found in the programmer reference manual for the wildfire.

After initializing serial communication, we call several functions to initialize digital IO.

Next, the code will enter into an infinite loop. A hard reset of the board is required to stop the code from executing. At the beginning of this loop, we check a flag called "gpsNum" which contains an integer value indicating which GPS unit we are reading data from. We then get a single character from the corresponding serial port for the particular GPS we are interested in (either GPS1 or GPS2). Data is gathered one character at a time and stored in a buffer called "buffer".

We keep gathering characters until we receive a '\$' character. This indicates the beginning of a GPS string and hence, the end of the previous string. Next, we call "parseGPS" function. This function takes two arguments: the buffer which contains our GPS sentence and a 2-dimensional character array which is used to store GPS data later on.

ParseGPS has several functions. It checks the GPS sentence stored in the buffer and looks for two types of sentences. The first sentence it looks for is a "GGA" sentence. If the sentence

contained in the buffer is a “GGA” sentence then we call “parseGGA”. We pass the buffer as an argument which still contains our GPS string.

ParseGGA will parse the GGA sentence and retrieve one specific piece of data: Altitude. It stores the altitude in an integer variable called “alt” and the length of the altitude is stored in an integer variable called “altLength”. If the altitude is invalid or cannot be read from the sentence, then a default altitude of 00000.00 is stored in “alt”.

Once the altitude is captured, parseGGA will return and set a flag. ParseGPS continues execution and now checks for a valid “RMC” sentence. If it finds this AND the parseGGA has already been called (by checking the flag) it will call parseRMC which also takes the buffer as an argument along with the 2-dimensional character array, “RMC” so that it can store the data from RMC.

The parseRMC function will collect the GPS status, coordinates, speed, and heading from the GPS RMC sentence and store this data in the 2-dimensional character array, “RMC”.

Upon returning to the “parseGPS” function execution, we will now call “arrange” function and pass “RMC” as an argument. Arrange will format all of the data into a custom sentence, including the altitude which is appended at the end. It will then store this custom sentence in a character array called “result”. A sentinel character, ‘@’ is used to mark the end of the sentence.

“Arrange” will now return and we will now copy the data in “result” into another buffer called “customPacket”. We stop once we reach the ‘@’ indicating the end of a sentence. Two additional values are appended to the end of “customPacket” which act as event flags. These can be set by the code later on, so that when we are receiving packets, we know if some specific event took place.

Next, we transmit the “customPacket” string via the COM3 port to the radio by calling the necessary UART function. We then change the value of GPSNum so that the GPS is alternated on the next loop of the code.

In addition to collecting, formatting, and transmitting GPS data, the parseGPS function also performs altitude checking many times per minute. It converts the value contained in “alt” which is in ASCII into an integer. If this value is nonzero (indicating that it is valid) it will store this in an integer array called “altitudeArray”. This array is designed to hold only three values, so once the index is incremented to “3” it is reset to “0”. This array will always contain the last three valid altitudes.

Next, the function will set the value for “alt_flag1” by calling the function “check_above_altitude” which takes a constant integer as an argument. This function will check all three values in “altitudeArray” and if all three values are above the “desired_altitude” (which was passed in as an argument) then it will return a one setting the flag, “alt_flag1”.

Continuing in the altitude checking section of code... An integer variable called “realeasePin” captures the input value of pin 5 on CN5 (either 0 or 1). If this value is set, this will indicate that a signal has been sent to the balloon package to trigger a cut down or separation of the bottom package.

Next the code will check to see if “alt_flag1” was set OR if the DTMF release was triggered. If either of these conditions is true, it will call “check_below_altitude” which returns an integer. This function uses the same technique as “check_above_altitude” to determine if the current altitude is less than the desired altitude. The result of this function is stored in “alt_flag2”.

Finally the code will check to see if “alt_flag2” is set and if so, it will set pin 8 of CN3 high. This currently triggers the parachute release.

The code then loops and the process is repeated for the next GPS packet.

4. Testing

4.1 Procedures

To setup / test Wildfire 5282:

- 1) Apply power to Wildfire. Must be between 5v and 24v. If using the power connector, **YOU MUST MAKE SURE GROUND IS ON THE OUTSIDE OF THE CONNECTOR** and not vice-versa. Having an incorrectly wired power source connected to the wildfire **WILL** damage the wildfire.
 - Green LED comes on labeled 3V3 when properly powered

- 2) If using two GPS's, make sure to use the correctly labeled code with dual GPS in the name.

- 3) GPS 1 and 2 must have 3.3v power to the red cable on their wiring harnesses.
 - The wildfire 5282 can only supply enough power to one GPS device from its 3.3V output port on the side of the board. Trying to power 2 GPS's off of the

wildfire will cause the wildfire to continually reset and sometimes not even get power on the 3.3v LED.

- 4) There must be a common ground shared between the Wildfire and any input compare or output compare or UART communication devices
 - You must use the area labeled GND located next to the 3.3V supply on the side of the Wildfire. Using a communication port's ground pin will cause the Wildfire to restart and output low voltage reset to COM1
- 5) The yellow wire from the GPS's wiring harnesses must be connected to pin5 on UART communication ports 1 and 2. If only using one GPS, connect to comm2.
- 6) TNC communication to the Tinytrack 4 or Kenwood must have the receive on tnc/kenwood side connected to pin 5 of UART comm3 as well as a common ground. Make sure to use the correct code version labeled TNC or kenwood as each has a different baud rate setting and set up procedure

To program Wildfire:

- 1) Defeat autorun if necessary, by connecting pins 1 and 3 on CN8. Port by the power LED
 - If autorun is turned on, the IDE program on the PC will not be able to detect the board.
- 2) Apply power

- 3) Connect serial cable to UART communication port 1 with red stripe at pin 1 on the UART port.
- 4) Launch SBC V3
- 5) Open needed code
- 6) Select sbc -> deploy -> then choose external flash.
- 7) Once finished, type “set autorun on” in the communication window at the bottom of the sbc program and press return.

4.2 Problems & Fixes Discovered During Testing

The following is a summary of the problems we ran into and what we did to meet our end goals.

A battery was wired incorrectly and fried two of the wildfire boards.

fix - One required a new voltage converter and the other needed a transistor removed to continue normal operation. Also the battery was rewired to the correct configuration.

When altitude started being received correctly instead of NULL, the wildfire code would freeze.

fix - we found out it was a problem with the formatting of the altitude and trying to add appending 0's to make it always XXXXX.XX format. So instead we removed the original implementation and let the altitude be dynamically sized. This fixed the buffer overflow problem.

We ran out of UART communication ports

fix - We decided to just always deploy the code to external flash then have it set to autorun so that we never had to use the UART comm1 port and thus freeing a needed port for the second

GPS set up. The downside to this is that without a terminal hooked up to the board (which we were doing through comm1) it is difficult to debug the code.

We had many problems with communicating with the Kenwood radio

fix - we figured out it defaults to 1200 baud then you can issue a command in 1200 baud to the kenwood like "HBAUD 9600" and then we switched our local baudrate on the comm port used (3) to 9600 and issued command "k" which switches the kenwood to converse mode. This makes it so that whatever you send to it then send a return character (ascii 13) it would broadcast it. This also enables you to operate at 9600 baud, which seemed to work best for us.

A bad GPS receiver was used on flight 1. In flight 1, we figured out the GPS was not transmitting any correct information to the wildfire.

fix - Tested all the GPS receivers we could get our hands on then marked them if they worked and marked them as broken if they didn't.

When testing the GPS packet transmission, we would get junk data or no data at all.

fix – We made sure that the GPS antenna and the antenna on the Kenwood radio or Transceiver are at least four or five feet apart. When these antennas are too close, they will interfere and data will not be transmitted correctly.

Transceiver would overheat from transmitting during long battery tests.

fix - The transceiver is not designed to transmit every few seconds over a long period of time. We changed the transmission delay so that it was transmitting once per minute instead of once every 10 seconds or so. This is the desired transmission occurrence during flight, anyhow.

5. Future Implementations

5.1 Ideas for the Future

UCLinux

This would be good to end up using because it makes everything easier. It makes storing information easier, provides multithreading possibilities, and delivers overall better performance.

Storing Custom Packets to SD card

This would have been implemented if we had enough time, but we ran out and it is really hard without using UCLinux. With UCLinux, you can just use the SD card as a mounted file system and perform normal file operations and output to a text file located on the SD card. This will be good so that flights can still be diagnosed if something goes wrong with the packet radio transmissions.

Threading

Right now, the code is a simple while loop with everything processed inline. This is bad because if one part of the code hangs up, nothing else happens. If threading is used with UCLinux, then it will be possible to set up all the separate GPS queries, radio transmission, and counters all separate.

Dual GPS failsafe

Right now, if one GPS completely loses power, the code will appear to be frozen and all packet transmissions will halt. In the future it would be a good idea to use threading to separate the GPS communication so that you can monitor each one and be able to fail it out if something goes wrong with it.

Combine all combinations of Wildire code into one

Use #defines and separate C files for each of the different sections of codes. This would combine the single gps kenwood, single gps transceiver, dual gps kenwood, and dual gps transceiver code into one easy to manage chunk of code. This would make life easier on everyone in the future.

Apply a change management software design

A big problem we had is that we could not simply revert to code we knew was working earlier. So having some kind of program keeping track of the different versions would make changing the code very easy to revert if something gets changed that breaks something.

Appendix A.1

Source Code for the Wildfire for the Top Package

```
/*  
-----  
FILE:   Main.c                               by - WSU High Altitude  
Balloon Team  
Original Code Developed by - CEG  
Members: Jon Baumann, Ryan Back, and Michael Walters  
With Contributions from - Dan Rahn  
date - Winter-Spring 2009  
  
VERSION: DUAL GPS WITH TRANSCEIVER  
  
PURPOSE: This code is designed to run on the Wildfire 5282 (non Linux) SBC.  
It is set up to read information from one or more GPS units,  
format  
the information into custom packet "sentences" transmit this  
information  
to a Kenwood radio which will transmit it via TNC packet radio.  
  
NOTES:  
-----*/  
  
#include <wf5282.h>           // uart_xxx, ...  
#include <stdio.h>           // for printf  
#include <string.h>  
#include <stdlib.h>  
  
//-----Function Prototypes-----  
void parseRMC(char str[], char RMC[8][15]);  
void parseGGA(char str1[]);  
void parseGPS(char str[], char RMC[8][15]);  
void arrange(char str1[8][15]);  
void TNC_init();  
int check_above_altitude(double desired_altitude);  
int check_below_altitude(double desired_altitude);  
  
//----- Variables --- (Should not be changed) -----  
-----  
char reinitStr[] = { "TNC Re-initialized for safety" };  
unsigned char buffer[100];  
char alt[8]; //="00000.00"; // altitude will be formatted to take on 00000.0  
format with leading zeros if less than 5 leading digits  
char RMC[8][15];  
int altitudeArray[3]; //array for storing five consecutive altitudes  
int altIndex = 0; //index for above array  
char result[1000];  
int i, j, k; // used to iterate through various loops  
char tempalt[100];
```

```

int altFlag = 0;
int transmit = 0;
int gotGGA = 0;
int reinit = 0;
int GGACounter = 0;
int alt_flag1 = 0; // Test trigger @ 60,000ft
int alt_flag2 = 0;
int trigger1 = 0;
int trigger2 = 0;
int releasePin;
unsigned char DTMFstate;
unsigned char prev_state;
int gpsNum = 1;

unsigned int sd_offset = 0;
int altLength = 0;

//----- Variables --- (May need to be modified) -----
//-----

const int ALTITUDE_CONST1 = 70000; //Integer value must be in feet
const int ALTITUDE_CONST2 = 62000; //Integer value must be in feet

COMPort gpsComm1, gpsComm2, TNC; //declar COM ports

//----- Implementation -----

// TYPE: Main
// return error code (not used)
int main( void )
{
    //-----Local Variables-----
    unsigned char c = ' '; //temporary variable used to store the GPS data
    from the serial port

    gpsComm1 = UART_COM1; //initialize GPS1 to the COM2 port
    gpsComm2 = UART_COM2;
    TNC = UART_COM3; //initialize TNC to the COM3 port

    uart_init( gpsComm1, // Initialize this UART
               UART_NO_PARITY, // No parity
               UART_DATA_BITS_8, // 8 data bits
               UART_STOP_BITS_1, // 1 stop bits
               4800 ); // MUST BE 4800 baud rate

    uart_init( gpsComm2, // Initialize this UART
               UART_NO_PARITY, // No parity
               UART_DATA_BITS_8, // 8 data bits
               UART_STOP_BITS_1, // 1 stop bits
               4800 ); // MUST BE 4800 baud rate

```

```

        //check the Wildfire Manual to see where I got WF_DIO_PORT_CF and
WF_DIO_PORT_CA
        dio_init( WF_DIO_PORT_CF, MSK_PINALL, OUTPUT); //initialize output
header
        dio_init( WF_DIO_PORT_CA, MSK_PINALL, INPUT); //initialize
input header

        TNC_init(); //call this function to initialize the TNC
i=0;
while( 1 ) //infinite loop (hard reset to quit)
{
    if (gpsNum == 1)
    {
        if( uart_chkch( UART_COM1 ) ) // if a char is ready to be
read
        {
            c = uart_getch( UART_COM1 ); // read the next char
            buffer[i++] = c; //store the char in the buffer &
increment buffer index
        }
        else
        {
            if( uart_chkch( UART_COM2 ) ) // if a char is ready to be
read
            {
                c = uart_getch( UART_COM2 ); // read the next char
                buffer[i++] = c; //store the char in the buffer &
increment buffer index
            }
            if(c == '$') // a "$" character indicates the beginning of a
GPS string and hence, the end
            {
                // of the previous string
                if (i > 1)
                {
                    parseGPS(buffer,RMC); // parsing data
                }
                i = 0; //reset buffer index
            }
        }
    }
}

return 0;
}

```

```

void parseGPS(char str[], char RMC[][15])
{
    if(str[2]=='G'&& str[3]=='G'&& str[4]=='A' && gotGGA == 0 ) // if the
first string read is the $GPGGA sentence
    {

```

```

        gotGGA = 1;
        parseGGA(str); // go to routine to handle $GPGGA sentence
    }

    if(str[2]=='R' && str[3]=='M' && str[4]=='C' && gotGGA == 1) // if the
first string read is the $GPRMC sentence
    {

        if (transmit == 10)                //add delay when transmitting
packets
        {
            if (reinit++ == 30)
            {
                TNC_init();
                uart_putstr( TNC, reinitStr );
                uart_putchar( TNC, 13);
                reinit = 0;
            }
            parseRMC(str,RMC); // go to routine to handle $GPRMC
sentence

            puts(result);
            putchar('\n');
            arrange(RMC);

            char customPacket[75];
            for(i = 0; result[i] != '@'; i++)
            {
                //putchar(result[i]);
                customPacket[i] = result[i];
            }

            customPacket[i++] = ' ';
            customPacket[i++] = (trigger1 + 48);
            customPacket[i++] = (trigger2 + 48);
            customPacket[i] = 0x00;

            uart_putstr( TNC, customPacket);
            uart_putchar( TNC, 13);
            if (gpsNum == 1)
                gpsNum = 2;
            else
                gpsNum = 1;

            putchar('\n');
            transmit = 0;
            gotGGA = 0;
                //new place
        }
    }
    else
    {
        //altitude checking
        //altitude is checked ten times more often than the
sentence is transmitted

```

```

        if(atoi(alt)!=0) //ignore altitudes of 0.0000
            altitudeArray[altIndex++] = atoi(alt);

        //reset altitude index to 0 after collecting 3 altitudes
        if(altIndex == 3) altIndex = 0;

        if(alt_flag1 == 0)
            alt_flag1 = check_above_altitude(ALTITUDE_CONST1);
        //set the first altitude flag if > ALTITUDE_CONST1

        //get the current state of the input pin
        DTMFstate = dio_get( WF_DIO_PORT_CA );

        //Input ports initialize with a value of 1
        printf( "Pin 2 state = %d\n", !(DTMFstate & MSK_PIN2) );
        releasePin = !(DTMFstate & MSK_PIN2); //get the
state of pin 5 on CN5: int of either 0 or 1

        //dio_set( parachutePort, MSK_PIN7, HIGH);

        if((alt_flag1 || releasePin) && (alt_flag2==0)) //if the
package is descending or DTMF release was triggered
        {
            alt_flag2 = check_below_altitude(ALTITUDE_CONST2);
        }
        if(alt_flag2) //final flag set: do action (deploy
parachute)
        {
            dio_set( WF_DIO_PORT_CF, MSK_PIN1, HIGH); //set pin 8
of CN3 high
            trigger1 = 1;
        }
    }
    transmit++;
}

}

void parseGGA(char str1[])
{
    char realgga[100];
    int comma = 0;
    j = 0;
    // loop until the 9th comma is reached - the altitude data lies between
the 9th & 10th comma
    for(i = 0; comma < 9; i++)
    {
        realgga[i] = str1[i];
        if(str1[i]==',') // if a comma is detected
        {
            comma++; // increment comma counter
            if(comma == 9) // the 9th comma is reached
            {
                if(str1[i+1] == ',') // if the next character is a
comma (null altitude)

```

```

        {
            alt[0] = '0';
            alt[1] = '0';
            alt[2] = '0';
            alt[3] = '0';
            alt[4] = '0';
            alt[5] = '.';
            alt[6] = '0';
            alt[7] = '0';
            altLength = 8;
        }

    else // otherwise altitude is valid
    {
        altLength = 0;
        for(i++; str1[i] != ','; i++)
        {
            alt[j++] = str1[i];
            altLength++;
        }
    }
}

return;
}

```

// this function reads the GPS status, coordinates, speed, and heading from the \$GPRMC sentence

```

void parseRMC(char str[], char RMC[][15])
{
    int comma = i = j = 0;

    for(i = 0; comma < 8 ; i++) // read in the data between commas
    {
        if(str[i] == ',') // if a comma is detected
        {
            comma++; // increment the comma counter

            if((comma == 7 || comma == 8) && str[i+1] == ',') // check to
            see if the speed & heading are null values
            {
                RMC[comma-1][0] = '0';
                RMC[comma-1][1] = '0';
                RMC[comma-1][2] = '0';
                RMC[comma-1][3] = '.';
                RMC[comma-1][4] = '0';
                RMC[comma-1][5] = ' ';
            }

            else
            {

```

```

        // read in the data from $GPRMC and copy to RMC array for
later processing
        for(j=0; str[j+i+1] != ',' && j<strlen(str) && str[j+i+1]
!= '*'; j++) // read in the data between commas
        RMC[comma-1][j]=str[j+i+1]; // copy to the RMC matrix for
future processing
        RMC[comma-1][j]=' '; // set the last character in each row
to space
    }
}
}

return;
}

void arrange(char str1[8][15])
{
    int x,y,z;
    z = x = y = 0;

    // set the index
    z=0;//strlen("GPS1 ");
    result[0] = 'G';
    result[1] = 'P';
    result[2] = 'S';
    result[3] = gpsNum+48; // add 48 to convert from an integer value to
an ASCII value
    result[4] = ' ';
    z = 5;
    // copy the data from the RMC matrix to the final string
    for(x=0;x<8;x++)
    {
        for(y=0;(str1[x][y] != ' ');y++)
        {
            result[z]=str1[x][y];
            z++;
        }
        result[z++] = ' '; // add a space between each piece
of data in the sentence
    }

    // append the altitude data to the end of the final string
    for (x = 0; x < altLength; x++)
        result[z++] = alt[x];

    result[z] = '@'; //add sentinel character
    y = z = x = 0;

}

void TNC_init()
{

```

```

    /*
    uart_init( TNC,          // Initialize this UART
              UART_NO_PARITY, // No parity
              UART_DATA_BITS_8, // 8 data bits
              UART_STOP_BITS_1, // 1 stop bits
              1200 );        // MUST BE 9600 baud rate
    */

    delay( 5000 );
    uart_putstr(TNC, "HBAUD 4800"); //Must be 9600 for Kenwood / 4800 for
transceiver
    uart_putch( TNC, 13 );

    uart_init( TNC,          // Initialize this UART
              UART_NO_PARITY, // No parity
              UART_DATA_BITS_8, // 8 data bits
              UART_STOP_BITS_1, // 1 stop bits
              4800 );        // MUST BE 4800 baud rate for transciever
/ 9600 for kenwood

    //delay( 5000 );
    //uart_putstr(TNC, "MY W1WSU-11");
    // uart_putch( TNC, 13 );
    //delay( 5000 );
    //uart_putstr(TNC, "k");
    // uart_putch( TNC, 13 );

}

int check_above_altitude(double desired_altitude) //returns 1 if
altitudes are above desired, 0 if it is not
{
    desired_altitude = desired_altitude * 0.3048; //convert feet to
meters
    for(i=0; i < 3; i++)
    {
        if(altitudeArray[i] < desired_altitude)
            return 0;
    }
    return 1;
}

int check_below_altitude(double desired_altitude) //returns 1 if
altitudes are above desired, 0 if it is not
{
    desired_altitude = desired_altitude * 0.3048; //convert feet to
meters
    for(i=0; i < 3; i++)
    {
        if(altitudeArray[i] > desired_altitude)
            return 0;
    }
    return 1;
}

```

Appendix A.2

Source Code for the Wildfire for the Bottom Package

```
/*
-----
FILE:   Main.c                               by - WSU High Altitude
Balloon Team

                                Original Code Developed by - CEG
Members: Jon Baumann, Ryan Back, and Michael Walters
                                With Contributions from - Dan Rahn
                                date - Winter-Spring 2009

VERSION: SINGLE GPS (COM2) WITH KENWOOD RADIO

PURPOSE: This code is designed to run on the Wildfire 5282 (non Linux) SBC.
          It is set up to read information from one or more GPS units,
format
          the information into custom packet "sentences" transmit this
information
          to a Kenwood radio which will transmit it via TNC packet radio.

NOTES:
----- */

#include <wf5282.h>           // uart_xxx, ...
#include <stdio.h>           // for printf
#include <string.h>
#include <stdlib.h>

//-----Function Prototypes-----
void parseRMC(char str[], char RMC[8][15]);
void parseGGA(char str1[]);
void parseGPS(char str[], char RMC[8][15]);
void arrange(char str1[8][15]);
void TNC_init();
int check_above_altitude(double desired_altitude);
int check_below_altitude(double desired_altitude);

//----- Variables --- (Should not be changed) -----
-----

char reinitStr[] = { "TNC Re-initialized for safety" };
unsigned char buffer[100];
char alt[8]; //="00000.00"; // altitude will be formatted to take on 00000.0
format with leading zeros if less than 5 leading digits
char RMC[8][15];
int altitudeArray[3]; //array for storing five consecutive altitudes
int altIndex = 0; //index for above array
char result[1000];
int i, j, k; // used to iterate through various loops
char tempalt[100];
int altFlag = 0;
```

```

int transmit = 0;
int gotGGA = 0;
int reinit = 0;
int GGACounter = 0;
int alt_flag1 = 0; // Test trigger @ 60,000ft
int alt_flag2 = 0;
int trigger1 = 0;
int trigger2 = 0;
int releasePin;
unsigned char DTMFstate;
unsigned char prev_state;
int gpsNum = 1;

unsigned int sd_offset = 0;
int altLength = 0;

//----- Variables --- (May need to be modified) -----
//-----

const int ALTITUDE_CONST1 = 70000; //Integer value must be in feet
const int ALTITUDE_CONST2 = 62000; //Integer value must be in feet

COMPort gpsComm1, gpsComm2, TNC; //declar COM ports

//----- Implementation -----

// TYPE: Main
// return error code (not used)
int main( void )
{
    //-----Local Variables-----
    unsigned char c = ' '; //temporary variable used to store the GPS data
    from the serial port

    gpsComm2 = UART_COM2;
    TNC = UART_COM3; //initialize TNC to the COM3 port

    uart_init( gpsComm2, // Initialize this UART
               UART_NO_PARITY, // No parity
               UART_DATA_BITS_8, // 8 data bits
               UART_STOP_BITS_1, // 1 stop bits
               4800 ); // MUST BE 4800 baud rate

    //check the Wildfire Manual to see where I got WF_DIO_PORT_CF and
    WF_DIO_PORT_CA
    dio_init( WF_DIO_PORT_CF, MSK_PINALL, OUTPUT); //initialize output
    header
    dio_init( WF_DIO_PORT_CA, MSK_PINALL, INPUT); //initialize
    input header

    TNC_init(); //call this function to initialize the TNC
    i=0;

```

```

while( 1 ) //infinite loop (hard reset to quit)
{
    if( uart_chkch( UART_COM2 ) ) // if a char is ready to be read
    {
        c = uart_getch( UART_COM2 ); // read the next char
        buffer[i++] = c; //store the char in the buffer &
increment buffer index
    }

    if(c == '$') // a "$" character indicates the beginning of a
GPS string and hence, the end
    {
        // of the previous string
        if (i > 1)
        {
            parseGPS(buffer,RMC); // parsing data

        }
        i = 0; //reset buffer index
    }
}

return 0;
}

```

```

void parseGPS(char str[], char RMC[][15])
{
    if(str[2]=='G'&& str[3]=='G'&& str[4]=='A' && gotGGA == 0 ) // if the
first string read is the $GPGGA sentence
    {
        gotGGA = 1;
        parseGGA(str); // go to routine to handle $GPGGA sentence
    }

    if(str[2]=='R'&& str[3]=='M'&& str[4]=='C' && gotGGA == 1) // if the
first string read is the $GPRMC sentence
    {

        if (transmit == 10) //add delay when transmitting
packets
        {
            if (reinit++ == 30)
            {
                TNC_init();
                uart_putstr( TNC, reinitStr );
                uart_putch( TNC, 13);
                reinit = 0;
            }
            parseRMC(str,RMC); // go to routine to handle $GPRMC
sentence

            puts(result);
            putchar('\n');
            arrange(RMC);

```

```

char customPacket[75];
for(i = 0; result[i] != '@'; i++)
{
    //putchar(result[i]);
    customPacket[i] = result[i];
}

customPacket[i++] = ' ';
customPacket[i++] = (trigger1 + 48);
customPacket[i++] = (trigger2 + 48);
customPacket[i] = 0x00;

uart_putstr( TNC, customPacket);
uart_putch( TNC, 13);

putchar('\n');
transmit = 0;
gotGGA = 0;
    //new place
}
else
{
    //altitude checking
    //altitude is checked ten times more often than the
sentence is transmitted
    if(atoi(alt)!=0) //ignore altitudes of 0.0000
        altitudeArray[altIndex++] = atoi(alt);

    //reset altitude index to 0 after collecting 3 altitudes
    if(altIndex == 3) altIndex = 0;

    if(alt_flag1 == 0)
        alt_flag1 = check_above_altitude(ALTITUDE_CONST1);
    //set the first altitude flag if > ALTITUDE_CONST1

    //get the current state of the input pin
    DTMFstate = dio_get( WF_DIO_PORT_CA );

    //Input ports initialize with a value of 1
    printf( "Pin 2 state = %d\n", !!(DTMFstate & MSK_PIN2) );
    releasePin = !!(DTMFstate & MSK_PIN2); //get the
state of pin 5 on CN5: int of eitehr 0 or 1

    //dio_set( parachutePort, MSK_PIN7, HIGH);

    if((alt_flag1 || releasePin) && (alt_flag2==0)) //if the
package is descending or DTMF release was triggered
    {
        alt_flag2 = check_below_altitude(ALTITUDE_CONST2);
    }
    if(alt_flag2) //final flag set: do action (deploy
parachute)
    {

```

```

        dio_set( WF_DIO_PORT_CF, MSK_PIN1, HIGH); //set pin 8
of CN3 high
    }
    }
    transmit++;
}

}
void parseGGA(char str1[])
{
    char realgga[100];
    int comma = 0;
    j = 0;
    // loop until the 9th comma is reached - the altitude data lies between
the 9th & 10th comma
    for(i = 0; comma < 9; i++)
    {
        realgga[i] = str1[i];
        if(str1[i]=='(',') // if a comma is detected
        {
            comma++; // increment comma counter
            if(comma == 9) // the 9th comma is reached
            {
                if(str1[i+1] == ',') // if the next character is a
comma (null altitude)
                {
                    alt[0] = '0';
                    alt[1] = '0';
                    alt[2] = '0';
                    alt[3] = '0';
                    alt[4] = '0';
                    alt[5] = '.';
                    alt[6] = '0';
                    alt[7] = '0';
                    altLength = 8;
                }
                else // otherwise altitude is valid
                {
                    altLength = 0;
                    for(i++; str1[i] != ','; i++)
                    {
                        alt[j++] = str1[i];
                        altLength++;
                    }
                }
            }
        }
    }
    return;
}
}

```

```

// this function reads the GPS status, coordinates, speed, and heading from
the $GPRMC sentence
void parseRMC(char str[], char RMC[][15])
{
    int comma = i = j = 0;

    for(i = 0; comma<8 ; i++) // read in the data between commas
    {
        if(str[i]==',') // if a comma is detected
        {
            comma++; // increment the comma counter

            if((comma == 7 || comma == 8)&& str[i+1]==',') // check to
see if the speed & heading are null values
            {
                RMC[comma-1][0] = '0';
                RMC[comma-1][1] = '0';
                RMC[comma-1][2] = '0';
                RMC[comma-1][3] = '.';
                RMC[comma-1][4] = '0';
                RMC[comma-1][5] = ' ';
            }

            else
            {
                // read in the data from $GPRMC and copy to RMC array for
later processing
                for(j=0; str[j+i+1] != ',' && j<strlen(str) && str[j+i+1]
!= '*'; j++) // read in the data between commas
                RMC[comma-1][j]=str[j+i+1]; // copy to the RMC matrix for
future processing
                RMC[comma-1][j]=' '; // set the last character in each row
to space
            }

        }

    }

    return;
}

void arrange(char str1[8][15])
{
    int x,y,z;
    z = x = y = 0;

    // set the index
    z=0;//strlen("GPS1 ");
    result[0] = 'G';
    result[1] = 'P';
    result[2] = 'S';
}

```

```

    result[3] = gpsNum+48; // add 48 to convert from an integer value to
an ASCII value
    result[4] = ' ';
    z = 5;
    // copy the data from the RMC matrix to the final string
    for(x=0;x<8;x++)
    {
        for(y=0;(str1[x][y] != ' ');y++)
        {
            result[z]=str1[x][y];
            z++;
        }
        result[z++] = ' '; // add a space between each piece
of data in the sentence
    }

    // append the altitude data to the end of the final string
    for (x = 0; x < altLength; x++)
        result[z++] = alt[x];

    result[z] = '@'; //add sentinel character
    y = z = x = 0;
}

void TNC_init()
{

    uart_init( TNC, // Initialize this UART
                UART_NO_PARITY, // No parity
                UART_DATA_BITS_8, // 8 data bits
                UART_STOP_BITS_1, // 1 stop bits
                1200 ); // MUST BE 1200 baud rate

    delay( 5000 );
    uart_putstr(TNC, "HBAUD 9600"); //Must be 9600 for Kenwood / 4800 for
transceiver
    uart_putchar( TNC, 13 );

    uart_init( TNC, // Initialize this UART
                UART_NO_PARITY, // No parity
                UART_DATA_BITS_8, // 8 data bits
                UART_STOP_BITS_1, // 1 stop bits
                9600 ); // MUST BE 4800 baud rate for transceiver
/ 9600 for kenwood

    delay( 5000 );
    uart_putstr(TNC, "MY W1WSU-11");
    uart_putchar( TNC, 13 );
    delay( 5000 );
    uart_putstr(TNC, "k");
    uart_putchar( TNC, 13 );

```

```
}

int check_above_altitude(double desired_altitude) //returns 1 if
altitudes are above desired, 0 if it is not
{
    desired_altitude = desired_altitude * 0.3048; //convert feet to
meters
    for(i=0; i < 3; i++)
    {
        if(altitudeArray[i] < desired_altitude)
            return 0;
    }
    return 1;
}

int check_below_altitude(double desired_altitude) //returns 1 if
altitudes are above desired, 0 if it is not
{
    desired_altitude = desired_altitude * 0.3048; //convert feet to
meters
    for(i=0; i < 3; i++)
    {
        if(altitudeArray[i] > desired_altitude)
            return 0;
    }
    return 1;
}
```