WEASEL TRACKER PROJECT

The design, building, and programming of a low cost Arduino tracker. Written by Chris Murkin 2011.

INTRODUCTION

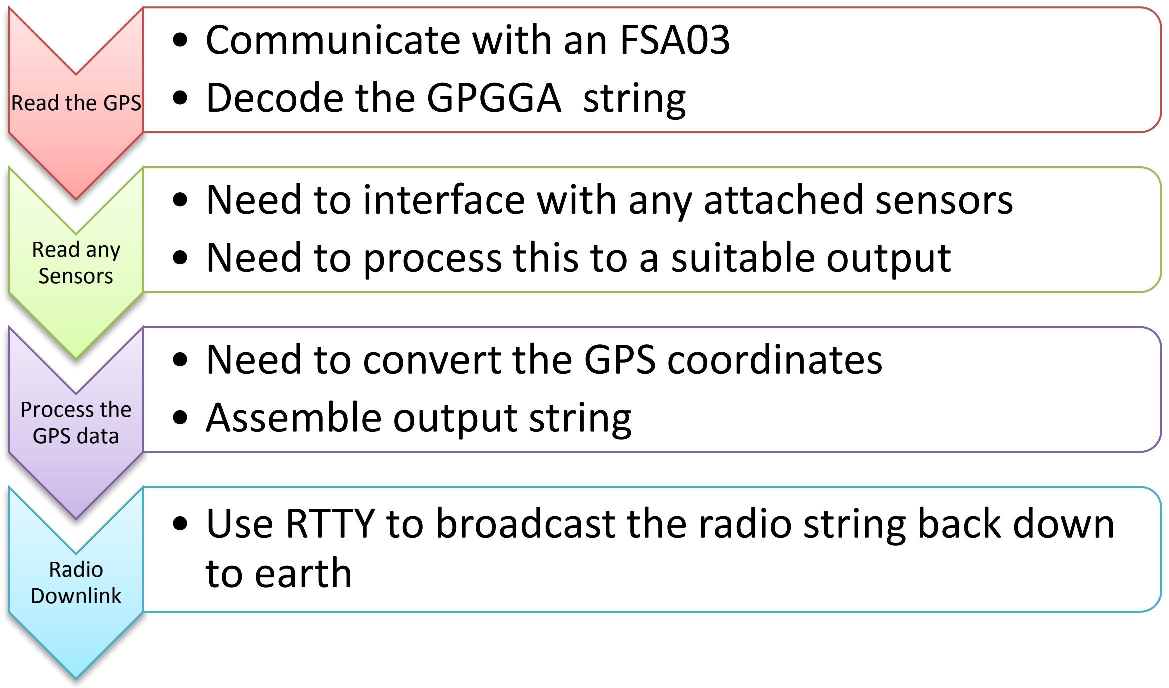
The aim for this project was to build a simple introductory High Altitude Balloon tracker. The idea of a tracker is to receive data from a GPS module, which works by receiving signals from a satellite and working out its exact position on earth. Using this information the tracker processes this data and converts it into a string of data which is then radioed back to ground based receivers. The radio receivers are connected to a computer, which decodes the radio message, and displays the output on the computer.

This was built as part of the Cambridge University Spaceflight (CUSF) project. This was intended to be a new members' project, and does not require prior experience.

The aim of this document is to assist anyone in trying to build a similar GPS tracker. There may be some hard to follow technical jargon in the document, although at some point in the guide most of the terms should be explained. If not a quick Google should explain what they mean.

The microprocessor used is an Arduino Uno. This is a low cost prototyping board, costing about £25. It is programmed by USB, and it is well supported with lots of example code online. It also supports the use of shields, which are circuit boards that fit on top of the Arduino where all the relevant components are attached. Other hardware includes a GPS module. We used an uBlox FSA03, which has an integrated aerial, and will work at high altitude. The radio used is a Radiometrix NTX2, which is legal to fly at altitude in the UK, and works well despite only a transmitting power of 10mW! We use high powered radios to receive the data. However with minor code modification different GPS modules or radios could be supported.

The overview process that needs to occur is shown in the graphic below.


Read the GPS
• Communicate with an FSA03
• Decode the GPGGA string
Read any Sensors
• Need to interface with any attached sensors
• Need to process this to a suitable output
Process the GPS data
• Need to convert the GPS coordinates
• Assemble output string
Radio Downlink
• Use RTTY to broadcast the radio string back down to earth

Contents

Getting Started

First you will probably need to buy some hardware; you need at least an Arduino to check the program works, and ideally a GPS too. For the flight you will also need a radio, battery (4 AA batteries), and aerial.

When connecting the GPS is important to check the operational voltage of the GPS and the Arduino, as if these are different a voltage regulator is required to power the GPS, along with a potential divider to ensure that any data signals from the higher voltage source do not damage the GPS receiver.

The GPS needs to be connected to a power supply, and the RX and TX pins connecting to pins 3 & 2, respectively, on the Arduino. The radio circuit diagram can be found on the UK High Altitude Society (UKHAS) website. http://ukhas.org.uk/guides:linkingarduinotontx2

You also need to install the Arduino programming software, this is a free download. You will also need to download the NewSoftSerial library.

Programming

The program for the Arduino was written in C++ using the Arduino programming environment. The code is shown in an appendix; along with a guide explain how the code operates. There is plenty of reference help available online, both relating to Arduino specific programming and C++ in general.

The Arduino has a USB socket so is easily programmed, and can also be used to debug the GPS.

TESTING

The tracker is very important to any High Altitude Balloon, and thus should work perfectly in flight. To ensure this is the case the tracker should be rigorously tested.

When testing Weasel, we started by unit testing the various functions to check they worked as expected. It was often easier to run this is a separate C++ compiler rather than on the Arduino. The initial conditions were manually set prior to the function being called, then the function was called, and verified the operation took place successfully. This took place on the GPS coordinate processing functions, the string creation, and testing the radio RTTY function (on the Arduino).

After this we tested using a piece of emulator code (shown in the appendix). This allowed us to manually enter a GPS string, and let the Arduino process this as normal. We then checked the results output with google maps, and as we knew where it was recorded we were able to check the accuracy of the tracker. It allows you to simulate different events occurring such as the tracker crossing the prime meridian, going to high altitude (checking the code handles large numbers). At the moment this isn't automated, although this could be a useful feature.

Finally we tested with a real GPS. We would run the tracker with the GPS, and then checked the output with Google Maps. The accuracy was often very within a couple of metres. After checking the output string was correct, we would check it with a radio. The radio was reliable so did not need as close attention as the GPS decoder which was far more temperamental.

It was important to test by all three methods, as often just static testing with the GPS would not pick-up common bugs such as crossing the meridian, and high altitudes; therefore the emulator code is just as important. We did not do as much emulator testing as we should have, and thus missed a bug with the storage of GPS coordinates (fixed in this guide).

SUGGESTED FURTHER IMPROVEMENTS

- Create functions for sensors
    - Connect a digital temperature, pressure, light sensor. Read the sensor, and modify the output string last sprintf. You will need to add variables or even make a sensor structure.
- Allow more of the GPS data strings to be used
    - Create some form of logic that can tell the GPS strings apart in the separator code.
- Try different transmission encoding, and higher bit rates
    - This should be carefully ground tested first!

SUGGESTED EXERCISES IF YOU WANT TO LEARN MORE

If you want you could try writing bits of the code yourself, and see if you can get it to work better than my code (quite easy, this was my first tracker project!)

- Try creating a function to decode the time

    - This is nice an simple to start.
- Create a function to change GPS coordinate clause format (without looking at my solution).
    - This is tricker, but can be tested before general release.
- Interface with sensors.
- Try to improve the GPS string separator, this probably could be done a fair bit better.

APPENDIX 1: CODE WALKTHROUGH

DECLARING VARIABLES

In this section I will explain how all information we used is stored on the Arduino such that code snippets will make more sense.

```
//Declare Varibles ======================
char clock[8]; //The time
char lat[11], lon[11]; //Lat and Long, in separated parts
char no_sat[3], alt[8], a; //Various other readings
int alt_count= 0;
int gcount=0, comma_count=0, dot_count=0, array_count=0; //gcount is the
number of chars from GPS per read
int ticks;
char buffer[200];
char str_out[200];
char callsign[7] = "WEASEL";
```

First section is the loose variables. These are used mainly during the decoding of the GPS input string.

- Initially all the GPS data is stored as character arrays. This is due to the method the program stores information from the GPS, and will be explained later. The various counts are also used for decoding the output data.
- **Ticks** increases sequentially every time the radio broadcasts a radio string back down to earth. This is used to give each outputted data string a unique identifier so it is easier to process the data.
- **Buffer** is where temporarily some of the output string is stored, prior to a checksum being added.
- **Str_out** is the string that will be broadcast from the tracker.
- **Callsign** is a unique identifier for the tracker. We called ours Weasel.

However a lot of the GPS specific data is stored in a separate struct. This is a method of grouping variables together and simplifies access of the data throughout the program.

```
struct program_data {
  long clock_int; //This is the clock as an interger
  long hour; //Shouldn't need to be a long, but is multiplied a lot when
used, so programs fails if this is an int.
  long minute;
  long second;
  int sat;
  long alt;
  long lat_int; //This is the latitude as an interger, in DD.MMMMMM before
GPS_Coord, and DD.DDDDD after GPS_Coord NOTE always positive.
  long lon_int; //Longitude equvialant.
  char ew; //This is either E or W depending on hemisphere
  char ns; //This is N or S depending on hemisphere
  char blanks[8]; //This is used to put in the blanks for output string, do
not use for any data
  char lat[11]; //This will be a string, with the form of a signed float so
should have -52.00123 for example after GPS_Coord is called
  char lon[11];
}
gps; //gps is the name of the varible with a structure of program_data.
Refer to contents by gps.sat, gps.lat[5] etc...
```

A structure called **program_data** is formed. This contains fields for the GPS data to be stored in this structure. To read/write they are addressed as gps.clock_int, gps.lat, gps.ew for example. These can be accessed in functions without a pass by reference.

- **Hour, minute, second** store the relevant sections of the clock output
- **Sat** stores the number of satellites it has lock with
- **Alt** stores the altitude
- **Lat_int Lon_int** stores the coordinate as in integer (so removes the decimal point from the coordinate)
- **ew** and **ns** stores the hemisphere that the tracker is currently in.
- **Blanks** is a char array that can store a number of blanking zeros to correct for a later error arising in the GPS coordinate conversion.
- The char arrays **lat** and **lon** store the output GPS coordinates as a string with the decimal point added. Ready to be transmitted.

Don't worry if you can't see why certain variables are required, when you see where they are used it will make a lot more sense.

Interfacing with the GPS

The GPS outputs data in the form of a GPS string:

```
$$GPGGA,182856.00,5211.89849,N,00007.23320,E,1,05,3.17,4.7,M,45.7,M,,*5D
```

This may at first seem confusing but this line of text (a string) contains all the information we need for the tracker.

1. **GPGGA** refers to the name of this string. This is an NMEA standard output string from a GPS, most of the shelf GPS modules will output information in this form. There are several variants but our code requires the GPGGA version.
2. The time is shown next (**18:28:56**) in this example. Next along is the latitude in a DDMM.MMMMM format.
3. Followed by a **N** or **S** indicator. This tells us which hemisphere of the earth the coordinate refers to.
4. Longitude
5. **E** or **W** indicator
6. The next number we are interested in is **05**. This refers to the number of satellites the GPS module can 'see'. This is useful for telling if the GPS is working well.
7. Finally **4.7** is the altitude. This is really useful for high altitude ballooning!
8. The rest of the information we are not interested in. But if you want to find out more look at this link http://www.gpsinformation.org/dale/nmea.htm#GGA

Before we can do anything we must be able to read from the GPS. To do this we use a library called NewSoftSerial (Google to download). This enables us to use any input ports on the Arduino to create a virtual serial port to communicate with the GPS. Serial is a method that electronic devices use to communicate. It consists of two wires, RX and TX, each wire allows data to be transported from on device to another. A wire is only one way, therefore two wires are used. To use this feature three key sections are required:

```
 //Set New soft serial to GPS

NewSoftSerial nss(3, 2);



nss.begin(9600);

while (nss.available()){
    a = nss.read(); //Reads the gps
```

First we must tell the program which pins the GPS are connected to in the form (rx, tx). This is what the top statement does.

The second line tells the Arduino what the rate of transmission the GPS will communicate at. This is measured in **baud**, which is the frequency of the communication in bits per second.

The final section simple says if a new piece of data is received by the Arduino, then save this as the character a. This character is then used in the later section to decode the string.

For the tracker to work we need to separate out the bits we want to know into separate variables for latter processing.

To do this we count the number of commas before the relevant section of the string we want, and then put all the characters of the string form section of the code into a char array. Then stop when the next comma comes along. At this point the data is stored as a group of alphanumerical characters and cannot be used in this form.

```
while (a != '*' && gcount <120 && nss.available()){
    a = nss.read(); //Reads the gps
    Serial.print(a); //Displays GPGGA (gps output string) on the USB debug

    if (a == ',') { //comma responce
      comma_count++;
      array_count=0;
    }
    if (a == '.') dot_count++; //dot count

//Store the data in the relevant place omitted here.

    gcount++;
  } //This is the end of the loop that runs until a * is reached.
```

The section of code above shows a while loop. A condition of this loop is that it only runs when a new character is read from the GPS; another is that it stops at the end of the string, before the checksum, to end the while loop; finally it has a time out clause which stops the loop running for ever so ends after 120 cycles.

Next shows an if clause that increments the comma count, resets the array count (as after a comma the data falls into a new array. A similar clause counts the full stops or decimal points.

```
switch (comma_count) { //Switch command

    case 1: //This is the time
      //Serial.print(" Time ");
      if (dot_count ==0){ //Only work before the dot!
        clock[array_count]=a;
        if (a!= ',') array_count++;
      }
      break;

    case 2: // This is latitude
      if (a != '.'& a != ',') {
        lat[array_count] = a;
        array_count++;
      }
      break;

    case 3: //N/S
      if (a !=',') gps.ns=a;
      break;

    case 4: // This is latitude
      if (a != '.'& a != ',') {
        lon[array_count] = a;
        array_count++;
      }
      break;

    case 5: //E/W
      if (a !=',') gps.ew=a;
      break;

    case 7: //Stats
      if (a != '.'& a != ',') {
        no_sat[array_count] = a;
        array_count++;
      }
      break;


    case 9: //Alt

      if (a != '.'& a != ',') {
        alt[array_count] = a;
        array_count++;
        alt_count++;
      }
      break;
    }
```

This section shows the use of the switch function. This acts as a look up table, and directs the program to the relevant section given by the comma count. Each section stores the current gps output character in the correct array.

```
//Insert stop bits on the string.
  clock[6]='\0';
  lat[10]='\0';
  lon[10]='\0';
  no_sat[2]='\0';
  alt[alt_count]='\0';
```

This code fragment above adds the stop bits onto a string. This ensures that the program knows where the end of the array is, and in doing so, becomes a string.

```
  //INT CONVERSION
  gps.lon_int = atol(lon);
  gps.lat_int = atol(lat);
  gps.alt = atol(alt)/10;
  gps.clock_int = atol(clock);
  gps.sat = atoi(no_sat);
```

The next section creates an integer or long integer; this is easier for the Arduino to process. It is now stored as part of a structure which allows the integers to be easily accessed in function in the code. This simplifies the code by reducing the need for a very long main program or by passing by reference lots of variables. Now all the numbers are integers, calculations can be easily made.

PROCESSING THE GPS DATA

The GPS data has now been read and stored as integers; however it is not in the correct format to transmit down to ground. Therefore we need to do some basic data processing. In the main body of the code this is represented as just three lines:

```
GPS_coord(); //Convert GPS Coordinates
  time_split();  //Get time format ready for the radio
  print_debug();//This will output the data from the GPS via USB serial
debugger. Can eliminate if required.
```

These call functions that are shown at the bottom of the main program. GPS_coord is a function that processes the GPS coordinates into the correct format. Time_split coverts integer time 123456 into 12:34:56. These are stored as strings ready to be transmitted.

GPS COORDINATE CONVERSION

```
void GPS_coord() {//This will convert the Coordinate to a form to transmit
  //Coordinate Conversion
  long lat_coord_dd, lon_coord_dd;
  long lat_coord_mm, lon_coord_mm;
  int a;
```

First of all this function is of a void type. This means that it does not report a number back the main program at the end of running the function.

Next we declare the variables. **Coord_dd** integer stores the degrees of a given longitude or latitude. **Coord_mm** stores the minutes of a given longitude or latitude. The tracker reports its position in a DD.DDDDD format, but the GPS records in a DDMM.MMMM format. Therefore I chose to split the output variable into two sections.

```
lat_coord_dd= (gps.lat_int/10000000);
  lat_coord_mm = ((gps.lat_int - (10000000*lat_coord_dd))/60);
  gps.lat_int  = lat_coord_dd*10000000  +  100*lat_coord_mm;  //This  should
return a coordinate as an interger using DD.DDDDD notation
```

This code changes **gps.lat_int** from DDMM[.]MMMM format to DD[.]DDDDDD format (the square brackets represent the missing decimal point in the integer). The first line stores just the first 2 digits which are in the degrees format.

The second line isolates the remaining digits, and divides by 60 (converting minutes to degrees).

Finally both parts are added together again, and stored as an integer.

```
lon_coord_dd= (gps.lon_int/10000000);
  lon_coord_mm = ((gps.lon_int - (10000000*lon_coord_dd))/60);
  gps.lon_int = lon_coord_dd*10000000 + 100*lon_coord_mm; //This should
return a coordinate as an interger using DD.DDDDD notation
```

A similar piece of code converts longitude.

A problem with storing a number with a decimal point as two integers occurs when a number like 11.01111 is stored. This becomes 11.1111 when converted into a string in the method used here. (Here we have coord_dd before decimal point, coord_mm after the decimal point.)

To get around this we need to add the zeros back in where needed, therefore I use another function **zero_lookup** to do this job.

```
void zero_lookup(long small_part){ //This will fill in zeros after the
decimal point
  int n = 5;
  if (small_part == 0){
    n=4;
  }
  else{
    while (small_part != 0) {
      small_part = small_part/10;
      n--;
    }
  }
  if (n>0){
    for (int i = 0; i<n; i++){
      gps.blanks[i] = '0';
    }
  }
  gps.blanks[n] = '\0';

}//End of zero_lookup
```

The idea of this function is that it is feed the number representing the part after the decimal point, and it sets **gps.blanks** to be a string of the correct number of zeros required before reporting the value of the small part. We first set n to be 5, this ensure that the maximum number of zeros entered is 5 as we work to 5 decimal places. The code next checks to see if **small_part** is zero, if so it reports n=4 (as the program will add the final zero later on).

If **small_part** is not zero, then the program keeps dividing **small_part** by ten until it becomes less than 1. In which case it records the number of zeros required.

Finally this writes it into a character array of zeros, and adds a stop digit.

This function is called twice, once for latitude and again for longitude. This can be seen in the last part of **GPS_coord**.

```
//Create a char array in the form of a signed float
  //latitude
  zero_lookup(lat_coord_mm);
  if (gps.ns == 'N'){
    a = sprintf(gps.lat, "%.2ld.%s%ld\0", lat_coord_dd, gps.blanks,
lat_coord_mm);
  }
  else{
    a = sprintf(gps.lat, "-%.2ld.%s%ld\0", lat_coord_dd, gps.blanks,
lat_coord_mm); //This is negative when in the northern hemisphere
  }
```

The above section only shows the code for latitude, with a repeat for longitude.

1. The program calls the zero look up function with the small part of latitude
2. Next the Arduino looks at the N/S indicator; if it is south it needs to add a minus sign when reporting southern hemisphere coordinates.
3. The program chooses the correct version of sprintf which is a function that assembles variables of the coordinate into a string with decimal point and correct number of zeros inserted. This is further explained in the transmission section.

TIME PROCESSING

```
void time_split(){ //This will split the clock interger to 3 components
  gps.hour = gps.clock_int/10000;
  gps.minute = (gps.clock_int - 10000*gps.hour)/100;
  gps.second = (gps.clock_int - 10000*gps.hour  -100*gps.minute);
}//End of time_split
```

This function separates out the time from an integer i.e. 123456, into three separate variables hour, minute, and second. These are found by dividing the integer by a factor, and storing it as an integer thus extracting the correct values. As these are stored in a structure, they do not need any pass by reference variables thus making the function quite simple.

USB DEBUG OUTPUT

```
void print_debug(){ //This will output the data from the GPS via USB serial
debugger.
  Serial.println(" ");
  Serial.print("Time: ");
  Serial.print(gps.hour);
  Serial.print(":");
  Serial.print(gps.minute);
  Serial.print(":");
  Serial.println(gps.second);
  Serial.print("Latitude: ");
  Serial.println(gps.lat);
  Serial.print("Longitude: ");
  Serial.println(gps.lon);
  Serial.print("Number of sats: ");
  Serial.println(gps.sat);
  Serial.print("Altitude: ");
  Serial.println(gps.alt);
}//end of Debug
```

This uses the **Serial.print** command to output the data collected by the tracker to a laptop connected by USB. This is used when testing if the GPS is connected correctly, and that the decoding procedures work correctly.

Transmission

The Arduino has now decoded the GPS data, and now needs to transmit the data. To do this it needs to create a string of data in the desired format ready to transmit back to ground.

The main function used is called **sprintf**. This is a really useful command in C++ that merges different variables of different types, and creates a string. For full information see http://www.cplusplus.com/reference/clibrary/cstdio/sprintf/. It gives the benefit of allowing you to add commas, colons, and decimal points in the string. Stop digits and also specify the number of significant figures.

We also use a checksum which is added to the end of a string, so that the ground radio can check that the string is received and decoded correctly. We use a **CRC16Sum**, and the function used to call this is shown in the code. This originates from a previous CUSF tracker project Ferret, and should not be changed.

Output string creation

```
//Prepare the string for transmission
  int  a  =  sprintf(buffer,  "$$%s,%u,%.2ld:%.2ld:%.2ld,%s,%s,%ld,%.2u\0",
callsign,  ticks,  gps.hour,  gps.minute,  gps.second,  gps.lat,  gps.lon,
gps.alt, gps.sat);
  //int a = sprintf(buffer, "$$%s,%l
  unsigned int checksum = CRC16Sum(buffer);
  char checksum_str[6];
  sprintf(checksum_str, "*%04X\n", checksum);
  int b = sprintf(str_out, "%s%s\n\0", buffer, checksum_str);
  Serial.println(str_out);
```

First we use sprintf to create the string in the character array called buffer. Note how the colons are added, and decimal points are added. \0 adds the stop digit. ".2" represents two significant figures (minimum) therefore shows number of satellites as 09 rather than 9.

Next we create an integer called checksum, which is created using the function CRC16Sum. The function is shown below.

```
unsigned int CRC16Sum(char *string) { //This will find the check sum
  unsigned int i;
  unsigned int crc;

  crc = 0xFFFF;

  // Calculate the sum, ignore $ sign's
  for (i = 0; i < strlen(string); i++) {
    if (string[i] != '$') crc = _crc_xmodem_update(crc,(uint8_t)string[i]);
  }

  return crc;
} //END OF CHECKSUM LOOP
```

Finally we use sprintf again to create string **str_out** with the checksum added. The \n allows the computer to skip to a new line after displaying the string, and helps aid clarity when reading the string. The last line sends the output string to the USB debug serial link, and is not used to transmit the data.

Radio downlink

The radio down link uses RTTY at 50 baud, with 8 bits in a byte. This is quite an old radio technique, and thus takes about 10s to send a single string. Thankfully we do not require much faster data transfer for a tracker, but it does prevent photos being sent using this method. However it does increase the accuracy of the radio transmission, and due to the lack of an error correction in the output string without it we could sometimes not be able to receive a string.

```
// Iterate through the string transmitting byte-by-byte.
  int j=0;
  while(str_out[j] != 0) {
    rtty_50(str_out[j]); //Use 50 baud RTTY, to downlink using pin 5 as
data pin
    j++;
```

This function reads each character in the output string **str_out** sequentially, and sends it to the function **rtty_50**. The function is the part that controls the radio, with this section in the main code only stepping through the characters until a stop digit '0' (not normal zero) is found.

```
void rtty_50(char abit) { //This is the RTTY on pin 5 with 50 baud
transmition string

  digitalWrite(5, LOW);     // START BIT set the radio to off
  delay(20);               // wait for 20ms

  for(int k = 0; k<8; k++){ //This is a loop as there are 8 bits in a byte,
which is one charactor
    if (abit & 1){ // This will return 1 if the last bit is 1, or 0 if the
last bit is 0, as 00000001 AND abcdefgh = 0000000h
      digitalWrite(5, HIGH);     //HIGH BIT set the radio to on
      delay(20);                // wait for 20ms
    }
    else{
      digitalWrite(5, LOW);     // Low BIT set the radio to off
      delay(20);                // wait for 20ms
    }
    abit = abit >>1; //Shift bits to the right by 1 step
  }

  digitalWrite(5, HIGH);     // END BIT set the radio to on
  delay(20);               // wait for 20ms

}//End of RTTY LOOP
```

The function will output to the radio each character in the string. Each signal is 20ms long, and thus we can have 50 bits per second rate of transmission.

1. To start a character need a start bit. This is a low bit for 1 baud.
2. Next we work through each bit of a character, which is 8 bits long. The Arduino stores the characters using an 8 bit byte, and we use this to create the up and down bits used in the transmission.
    a. First we create the character abit which is equal to the character being transmitted.
    b. Next we bitwise and abit with 1 (0000 0001) so this creates the output (0000 000x) where x is the last bit of abit.
    c. If this output 1 then we output high for 20ms, otherwise we output low for 20ms
    d. Next we use abit = abit >>1; this shifts bits to the right, so 1111 1111 becomes 0111 1111, so now the original second bit from the right, is not the final bit.
    e. Repeat 8 time for all eight bits
3. Finally create the stop bit, which is also 1 baud length long, which is high.

4. Transmit next character.

This might seem a little mysterious, but feel free to just copy this function into your code. To change the baud rate, change every instance of 20 to another number, this must be in milliseconds.

Reset the Arduino for next loop

This second covers the routine resetting of the microprocessor to ensure that all the variables are reset back their original states.

```
ticks++;// increments the no. of cycles counter

delay(500); //This just adds in a delay before starting again with the next line of the code

  if  (1.0*ticks/20  == ticks/20)  set_GPS(); //This should every 20th cycle reset the GPS, ensuring that it remains in Nav Mode with all of the correct strings in place

  //RESET string decoder

  while(a != '$' && nss.available()) { //This runs through until a dollar sign is reached the start of a new part of the code.
    a = nss.read();
  }

  comma_count = 0; //This initalise the varibles used in decoding a GPS string.
  dot_count = 0;
  gcount = 0;
  alt_count = 0;
```

- **Ticks**, this is incremented every loop, and reflects the number of transmissions the tracker has made. This is a unique number every string, and often used for data processing after.
- The delay ensures that the transmissions do not merge together.
- The if statement resets the GPS every 20 transmissions, this ensure that the GPS stays in the correct mode throughout the flight, in our case the FSA03 must in Navigation mode, so that it works above 24km, if the power momentarily fails it could leave this mode, and compromise data. Thus this check is made.
- The while loop ensure that the GPS output is in the correct place for the separator code to work
- The final section resets the separator code variables.

SET-UP LOOP

This only runs once on power up of the Tracker. I left this section till last, as it is quite specific to the FSA03 GPS. A fair chunck of the code was used from the UKHAS website section on the FSA03.

```
#include <NewSoftSerial.h>
#include <stdio.h>
#include <string.h>
#include <util/crc16.h>

```

This shows what libraries we used in the code.

- **NewSoftSerial** is used to interface with the GPS, this can be downloaded online
- **Stdio** is built in, and is required for sprintf, and itoa.
- **String.h** is used when processing the strings. This is also built into the Arduino environment.
- **Util/crc16** is used in the checksum, and was built into the Arduino complier, but is probably online.

```
void                    setup()                   {//Startup                  loop
=====================================================

  // Start up serial ports
  nss.begin(38400);
  Serial.begin(115200); // used for debug ouput

  delay(2000); // Give the GPS time to come boot

  // Lower the baud rate to 9600 from 38.4k
  Serial.print("Setting uBlox port mode: ");
  uint8_t setPort[] = {
    0xB5, 0x62, 0x06, 0x00, 0x14, 0x00, 0x01, 0x00, 0x00, 0x00, 0xD0, 0x08,
0x00,  0x00,  0x80,  0x25,  0x00,  0x00,  0x03,  0x00,  0x03,  0x00,  0x00,  0x00,
0x00, 0x00, 0x9E, 0x95                };
  sendUBX(setPort, sizeof(setPort)/sizeof(uint8_t));

  // Switch baud rates on the software serial
  Serial.println("Switching to 9600b GPS serial");
  nss.begin(9600);
  delay(1000);

  set_GPS();

  ticks = 1;


} //END OF SET UP LOOP
```

First we set the baud rate of the nss (new soft serial) which communicates with the GPS. We also set Serial.begin, which is the USB debugger baud rate 115200 for fast data transfer.

We have a 2 second delay to allow the GPS to boot.

Next we tell the GPS to lower the Baud rate, using the hex string provided. This is in accordance the Ublox (GPS communication specification) method. We next change the baud rate on the Arduino to match the new GPS baud rate.

Set_GPS is another function that put the GPS in the correct (navigation) mode, and turns of the unwanted strings. See full code for this function, you can remove a paragraph if you want to retain a certain string, however this will not work with the gps string separator code as present.

The function sendUBX is from the UKHAS, and is used to send the correct data string to the GPS.

Emulator Code

This allows the user to manually enter a GPGGA string, and the tracker will process this rather than using a GPS. I recommend doing this in a copy of the code, as quite a few changes are required.

```
char test_str[] =
"$$GPGGA,135506.00,5212.25882,N,00007.52029,E,2,09,0.99,17.2,M,45*8A";
//Note this is a real string from cambridge
```

This line of code needs to be added in the declaration of variables section.

```
void
loop()//=================================================================
{

  //READ THE GPS AND EXTRACT THE USEFUL DATA
  while (a != '*' && gcount <120){
    a = test_str[gcount]; //Reads the gps
    Serial.print(a); //Displays GPGGA (gps output string) on the USB debug

    if (a == ',') { //comma response
```

This is the modified start of the void loop() function. The rest of the code stays as is, with all references to the function set_GPS should be deleted or commented out.

When using this changing the E/W or N/S indicators in the GPGGA string, change different number to ensure that the correct section is read, and finally check the coordinates conversion works at different points.

Appendix 2: Full code

```
// This code sets a FSA03 GPS module to give gps data to a radio module to
downlink to earth.

//Code written by Chris Murkin, Leo, Hannah, and CU Spaceflight with FSA03
set up code from the UKHAS website
//This code is written for the Cambridge University Space Flight project.
//It was used for the Weasel tracker an introductory proect
//The check sum code was borrowed from CUSF Ferret Tracker, written by Jon
Sowman avalible from Git hub.

//This should work in all parts of the world, but be careful as
gps.lat_int, and gps.lon_int are always positive

#include <NewSoftSerial.h>
#include <stdio.h>
#include <string.h>
#include <util/crc16.h>


//Set New soft serial to GPS
NewSoftSerial nss(3, 2);

//Declare Varibles ======================
char clock[8]; //The time
char lat[11], lon[11]; //Lat and Long, in separated parts
char no_sat[3], alt[8], a; //Various other readings
int alt_count= 0;
int gcount=0, comma_count=0, dot_count=0, array_count=0; //gcount is the
number of chars from GPS per read
int ticks;
char buffer[200];
char str_out[200];
char callsign[7] = "WEASEL";

struct program_data {
  long clock_int; //This is the clock as an interger
  long hour; //Shouldn't need to be a long, but is multiplied a lot when
used, so programs fails if this is an int.
  long minute;
  long second;
  int sat;
  long alt;
  long lat_int; //This is the latitude as an interger, in DD.MMMMMM before
GPS_Coord, and DD.DDDDD after GPS_Coord NOTE always positive.
  long lon_int; //Longitude equvialant.
  char ew; //This is either E or W depending on hemisphere
  char ns; //This is N or S depending on hemisphere
  char blanks[8]; //This is used to put in the blanks for output string, do
not use for any data
  char lat[11]; //This will be a string, with the form of a signed float so
should have -52.00123 for example after GPS_Coord is called
  char lon[11];
}
gps; //gps is the name of the varible with a structure of program_data.
Refer to contents by gps.sat, gps.lat[5] etc...

void                    setup()                 {//Startup            loop
==================================================

  // Start up serial ports
  nss.begin(38400);
  Serial.begin(115200); // used for debug ouput

  delay(2000); // Give the GPS time to come boot

  // Lower the baud rate to 9600 from 38.4k
  Serial.print("Setting uBlox port mode: ");
  uint8_t setPort[] = {
    0xB5, 0x62, 0x06, 0x00, 0x14, 0x00, 0x01, 0x00, 0x00, 0x00, 0xD0, 0x08,
0x00,  0x00,  0x80,  0x25,  0x00,  0x00,  0x03,  0x00,  0x03,  0x00,  0x00,  0x00,
0x00, 0x00, 0x9E, 0x95                  };
  sendUBX(setPort, sizeof(setPort)/sizeof(uint8_t));

  // Switch baud rates on the software serial
  Serial.println("Switching to 9600b GPS serial");
  nss.begin(9600);
  delay(1000);

  set_GPS();

  ticks = 1;


} //END OF SET UP LOOP


void
loop()//============================================================================
{

  //READ THE GPS AND EXTRACT THE USEFUL DATA
  while (a != '*' && gcount <120 && nss.available()){
    a = nss.read(); //Reads the gps
    Serial.print(a); //Displays GPGGA (gps output string) on the USB debug

    if (a == ',') { //comma responce
      comma_count++;
      array_count=0;
    }
    if (a == '.') dot_count++; //dot count

    switch (comma_count) { //Switch command

    case 1: //This is the time
      //Serial.print(" Time ");
      if (dot_count ==0){ //Only work before the dot!
        clock[array_count]=a;
        if (a!= ',') array_count++;
      }
      break;

    case 2: // This is latitude
      if (a != '.'& a != ',') {
        lat[array_count] = a;
        array_count++;
      }
      break;

    case 3: //N/S
```

```
      if (a !=',') gps.ns=a;
      break;

    case 4: // This is latitude
      if (a != '.'& a != ',') {
        lon[array_count] = a;
        array_count++;
      }
      break;

    case 5: //E/W
      if (a !=',') gps.ew=a;
      break;

    case 7: //Stats
      if (a != '.'& a != ',') {
        no_sat[array_count] = a;
        array_count++;
      }
      break;


    case 9: //Alt

      if (a != '.'& a != ',') {
        alt[array_count] = a;
        array_count++;
        alt_count++;
      }
      break;
    }

    gcount++;
  } //This is the end of the loop that runs until a * is reached.

  //Insert stop bits on the string.
  clock[6]='\0';
  lat[10]='\0';
  lon[10]='\0';
  no_sat[2]='\0';
  alt[alt_count]='\0';


  //INT CONVERSION
  gps.lon_int = atol(lon);
  gps.lat_int = atol(lat);
  gps.alt = atol(alt)/10;
  gps.clock_int = atol(clock);
  gps.sat = atoi(no_sat);


  //TEST CODE = MAKESURE OFF FOR LAUNCH
  //gps.ew='W'; //This will simulate crossing the prime meridian
  //gps.ns='S'; //This will simulate crossing the equator.

  GPS_coord(); //Convert GPS Coordinates
  time_split();  //Get time format ready for the radio
  print_debug();//This will output the data from the GPS via USB serial
debugger. Can eliminate if required.

  //Prepare the string for transmission
  int  a  =  sprintf(buffer,  "$$%s,%u,%.2ld:%.2ld:%.2ld,%s,%s,%ld,%.2u\0",
callsign,  ticks,  gps.hour,  gps.minute,  gps.second,  gps.lat,  gps.lon,
gps.alt, gps.sat);
  //int a = sprintf(buffer, "$$%s,%l
  unsigned int checksum = CRC16Sum(buffer);
  char checksum_str[6];
  sprintf(checksum_str, "*%04X\n", checksum);
  int b = sprintf(str_out, "%s%s\n\0", buffer, checksum_str);
  Serial.println(str_out);

  ticks++;

  // Iterate through the string transmitting byte-by-byte.
  int j=0;
  while(str_out[j] != 0) {
    rtty_50(str_out[j]); //Use  50  baud  RTTY,  to  downlink  using  pin  5  as
data pin
    j++;
  }

  delay(500); //This  just  adds  in  a  delay  before  starting  again  with  the
next line of the code

  if  (1.0*ticks/20  ==  ticks/20)  set_GPS(); //This  should  every  20th  cycle
reset the GPS, ensuring that it remains in Nav Mode with all of the correct
strings in place

  //RESET string decoder

  while(a  !=  '$'  &&  nss.available())  {  //This  runs  through  until  a  dollar
sign is reached the start of a new part of the code.
    a = nss.read();
  }

  comma_count  =  0;  //This  initalise  the  varibles  used  in  decoding  a  GPS
string.
  dot_count = 0;
  gcount = 0;
  alt_count = 0;


} //END OF MAIN LOOP


//FUNCTIONS
===========================================================================
=====================

void  rtty_50(char  abit)  {  //This  is  the  RTTY  on  pin  5  with  50  baud
transmition string

  digitalWrite(5, LOW);     // START BIT set the radio to off
  delay(20);              // wait for 20ms

  for(int k = 0; k<8; k++){ //This is a loop as there are 8 bits in a byte,
which is one charactor
    if (abit & 1){ // This will return 1 if the last bit is 1, or 0 if the
last bit is 0, as 00000001 AND abcdefgh = 0000000h
      digitalWrite(5, HIGH);     //HIGH BIT set the radio to on
      delay(20);              // wait for 20ms
    }
    else{
      digitalWrite(5, LOW);     // Low BIT set the radio to off
      delay(20);               // wait for 20ms
    }
    abit = abit >>1; //Shift bits to the right by 1 step
  }

  digitalWrite(5, HIGH);     // END BIT set the radio to on
  delay(20);               // wait for 20ms

}//End of RTTY LOOP

void set_GPS(){
  // Set the navigation mode (Airborne, 1G)
  Serial.print("Setting uBlox nav mode: ");
  uint8_t setNav[] = {
    0xB5, 0x62, 0x06, 0x24, 0x24, 0x00, 0xFF, 0xFF, 0x06, 0x03, 0x00, 0x00,
0x00, 0x00, 0x10, 0x27, 0x00, 0x00, 0x05, 0x00, 0xFA, 0x00, 0xFA, 0x00,
0x64, 0x00, 0x2C, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x16, 0xDC};
  sendUBX(setNav, sizeof(setNav)/sizeof(uint8_t));
  getUBX_ACK(setNav);

  //Switch off GLL
  Serial.print("Switching off NMEA GLL: ");
  uint8_t setGLL[] = {
    0xB5, 0x62, 0x06, 0x01, 0x08, 0x00, 0xF0, 0x01, 0x00, 0x00, 0x00, 0x00,
0x00, 0x01, 0x01, 0x2B};
  sendUBX(setGLL, sizeof(setGLL)/sizeof(uint8_t));
  getUBX_ACK(setGLL);

  //Switch off GSA
  Serial.print("Switching off NMEA GSA: ");
  uint8_t setGSA[] = {
    0xB5, 0x62, 0x06, 0x01, 0x08, 0x00, 0xF0, 0x02, 0x00, 0x00, 0x00, 0x00,
0x00, 0x01, 0x02, 0x32};
  sendUBX(setGSA, sizeof(setGSA)/sizeof(uint8_t));
  getUBX_ACK(setGSA);

  // Switch off GSV
  Serial.print("Switching off NMEA GSV: ");
  uint8_t setGSV[] = {
    0xB5, 0x62, 0x06, 0x01, 0x08, 0x00, 0xF0, 0x03, 0x00, 0x00, 0x00, 0x00,
0x00, 0x01, 0x03, 0x39};
  sendUBX(setGSV, sizeof(setGSV)/sizeof(uint8_t));
  getUBX_ACK(setGSV);

  // Switch off RMC
  Serial.print("Switching off NMEA RMC: ");
  uint8_t setRMC[] = {
    0xB5, 0x62, 0x06, 0x01, 0x08, 0x00, 0xF0, 0x04, 0x00, 0x00, 0x00, 0x00,
0x00, 0x01, 0x04, 0x40};
  sendUBX(setRMC, sizeof(setRMC)/sizeof(uint8_t));
  getUBX_ACK(setRMC);

  //Switch off VTG
  Serial.print("Switching off NMEA VTG (Test): ");
  uint8_t setVTG[] = {
    0xB5, 0x62, 0x06, 0x01, 0x08, 0x00, 0xF0, 0x05, 0x00, 0x00, 0x00, 0x00,
0x00, 0x01, 0x05, 0x47};
  sendUBX(setVTG, sizeof(setVTG)/sizeof(uint8_t));
  getUBX_ACK(setVTG);
} //END OF SET GPS



void time_split(){ //This will split the clock interger to 3 components
  gps.hour = gps.clock_int/10000;
  gps.minute = (gps.clock_int - 10000*gps.hour)/100;
  gps.second = (gps.clock_int - 10000*gps.hour  -100*gps.minute);
}//End of time_split



void GPS_coord() {//This will convert the Coordinate to a form to transmit
  //Coordinate Conversion
  long lat_coord_dd, lon_coord_dd;
  long lat_coord_mm, lon_coord_mm;
  int a;

  lat_coord_dd= (gps.lat_int/10000000);
  lat_coord_mm = ((gps.lat_int - (10000000*lat_coord_dd))/60);
  gps.lat_int  =  lat_coord_dd*10000000  +  100*lat_coord_mm;  //This  should
return a coordinate as an interger using DD.DDDDD notation

  lon_coord_dd= (gps.lon_int/10000000);
  lon_coord_mm = ((gps.lon_int - (10000000*lon_coord_dd))/60);
  gps.lon_int  =  lon_coord_dd*10000000  +  100*lon_coord_mm;  //This  should
return a coordinate as an interger using DD.DDDDD notation

  //Create a char array in the form of a signed float
  //latitude
  zero_lookup(lat_coord_mm);
  if (gps.ns == 'N'){
    a   =   sprintf(gps.lat,   "%.2ld.%s%ld\0",   lat_coord_dd,   gps.blanks,
lat_coord_mm);
  }
  else{
    a   =   sprintf(gps.lat,   "-%.2ld.%s%ld\0",   lat_coord_dd,   gps.blanks,
lat_coord_mm); //This is negative when in the northern hemisphere
  }
  //longitude
  zero_lookup(lon_coord_mm);
  if (gps.ew == 'E'){
    a   =   sprintf(gps.lon,   "%.2ld.%s%ld\0",   lon_coord_dd,   gps.blanks,
lon_coord_mm);
  }
  else{
    a   =   sprintf(gps.lon,   "-%.2ld.%s%ld\0",   lon_coord_dd,   gps.blanks,
lon_coord_mm); //This is negative when in the southern hemisphere
  }
}//End of GPS shift


void  zero_lookup(long  small_part){  //This  will  fill  in  zeros  after  the
decimal point
  int n = 5;
  if (small_part == 0){
    n=4;
  }
  else{
    while (small_part != 0) {
      small_part = small_part/10;
      n--;
    }
  }
  if (n>0){
    for (int i = 0; i<n; i++){
      gps.blanks[i] = '0';
    }
  }
  gps.blanks[n] = '\0';

}//End of zero_lookup


void print_debug(){ //This will output the data from the GPS via USB serial
debugger.
  Serial.println(" ");
  Serial.print("Time: ");
  Serial.print(gps.hour);
  Serial.print(":");
  Serial.print(gps.minute);
  Serial.print(":");
  Serial.println(gps.second);
  Serial.print("Latitude: ");
  Serial.println(gps.lat);
  Serial.print("Longitude: ");
  Serial.println(gps.lon);
  Serial.print("Number of sats: ");
  Serial.println(gps.sat);
  Serial.print("Altitude: ");
  Serial.println(gps.alt);
}//end of Debug


unsigned int CRC16Sum(char *string) { //This will find the check sum
  unsigned int i;
  unsigned int crc;

  crc = 0xFFFF;

  // Calculate the sum, ignore $ sign's
  for (i = 0; i < strlen(string); i++) {
    if (string[i] != '$') crc = _crc_xmodem_update(crc,(uint8_t)string[i]);
  }

  return crc;
} //END OF CHECKSUM LOOP

// Send a byte array of UBX protocol to the GPS
void sendUBX(uint8_t *MSG, uint8_t len) {
  for(int i=0; i<len; i++) {
    nss.print(MSG[i], BYTE);
    Serial.print(MSG[i], HEX);
  }
  Serial.println();
}//END sendUBX


// Calculate expected UBX ACK packet and parse UBX response from GPS
boolean getUBX_ACK(uint8_t *MSG) {
  uint8_t b;
  uint8_t ackByteID = 0;
  uint8_t ackPacket[10];
  unsigned long startTime = millis();
  Serial.print(" * Reading ACK response: ");

  // Construct the expected ACK packet
  ackPacket[0] = 0xB5; // header
  ackPacket[1] = 0x62; // header
  ackPacket[2] = 0x05; // class
  ackPacket[3] = 0x01; // id
  ackPacket[4] = 0x02; // length
  ackPacket[5] = 0x00;
  ackPacket[6] = MSG[2];        // ACK class
  ackPacket[7] = MSG[3];        // ACK id
  ackPacket[8] = 0;             // CK_A
  ackPacket[9] = 0;             // CK_B

  // Calculate the checksums
  for (uint8_t i=2; i<8; i++) {
    ackPacket[8] = ackPacket[8] + ackPacket[i];
    ackPacket[9] = ackPacket[9] + ackPacket[8];
  }

  while (1) {

    // Test for success
    if (ackByteID > 9) {
      // All packets in order!
      Serial.println(" (SUCCESS!)");
      return true;
    }

    // Timeout if no valid response in 3 seconds
    if (millis() - startTime > 3000) {
      Serial.println(" (FAILED!)");
      return false;
    }

    // Make sure data is available to read
    if (nss.available()) {
      b = nss.read();

      // Check that bytes arrive in sequence as per expected ACK packet
      if (b == ackPacket[ackByteID]) {
        ackByteID++;
        Serial.print(b, HEX);
      }
      else {
        ackByteID = 0; // Reset and look again, invalid order
      }

    }
  }
}//End of getUBX_ACK
```