Using SNAP to Drive and Monitor GPS Devices

SNAP Engines offer a nice mix of power and portability, and that means they are perfect candidates for pairing with GPS devices to plot locations or movements.

October 11, 2010
Doc Number 600041-01A

Barry Tice
Associate Design Engineer

500 Discovery Drive, Huntsville, Alabama 35806, 877-982-7888

Table of Contents

Revision History

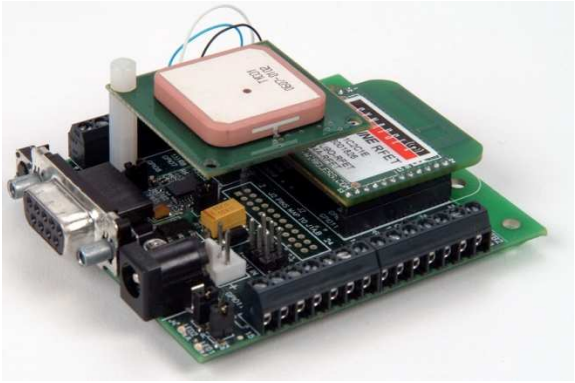
Previous Version	Change	Page

The Problem

GPS units are ubiquitous and fairly inexpensive. The challenge is finding (or building) a system with which the GPS device can interface for reporting.

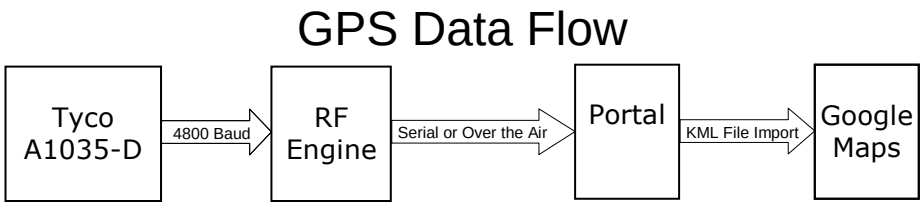
The Solution

Connecting a GPS unit to the UART of a SNAP device lets you wrap some intelligence around the incoming data AND allows you to accumulate and track that location data across a wide area.



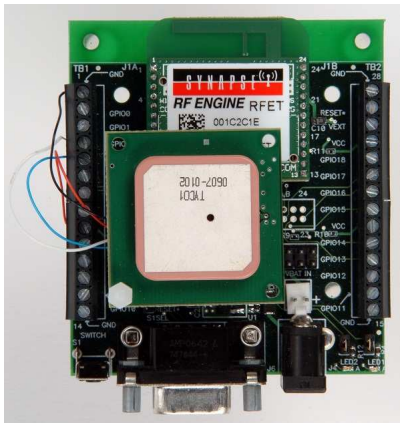
This example uses an RF100 SNAP Engine on a Synapse Wireless ProtoBoard, with a Tyco A1035-D GPS module. That GPS outputs location data in standard NMEA-0183 format as a serial stream at 4800 baud.

A SNAPpy script in the SNAP Engine connects to the GPS module through UART0. It reports that data back to Portal, which formats it into a KML file that can then be loaded into Google Earth or Google Maps. If you are in a position to tether the SNAP Engine to a PC (or laptop), you can connect it using a serial cable. Otherwise the unit can report back to Portal over the air. If reporting over the air, a mesh network of other SNAP devices can extend your reporting range over many miles.



Hardware setup

There isn't much hardware setup required.



You will need to determine the serial TX and RX pins on your device. Connect those pins to GPIO_3 and GPIO_4 of the SNAP Engine using the ProtoBoard's terminal blocks. On the Tyco A1035-D module, TX is Pin 3, RX is Pin 5, VCC is Pin 7, and Ground is Pin 9.

You can probably also power the GPS device from the ProtoBoard, depending on the GPS device's

requirements. The ProtoBoard provides 3.3 volts as VCC, when using an external power supply. If powering the board from batteries, VCC is the battery voltage. If that voltage is an appropriate level for the GPS module (it is for the A1035-D), connect the GPS module's power pins to the VCC and Ground connections on the ProtoBoard.

If you would like audible feedback from your GPS node, connect a piezo buzzer from GPIO_12 to ground. The buzzer will sound each time the board sends a GPS reading to Portal.

See Appendix 2 for one example of a way to modularize the GPS unit so it can be quickly connected to and removed from a ProtoBoard.

Software setup

Load the GPS_Portal.py Python script into Portal.

Load the GPS_Logger.py SNAPpy script into the RF Engine, and power the ProtoBoard.

Connect Portal to your network. You can use a serial cable directly to the ProtoBoard, or use another SNAP device as your bridge node and connect to the GPS node wirelessly.

If you have changed Portal's address from the default value of 00.00.01, use the specifyPortalAddress() function in the GPS node to tell the node what Portal's address is. It uses an NV parameter in the node to remember the address between reboots. (Alternately, you can edit the script to specify Portal's address.)

The GPS node will blink LED1 every second to indicate that it is powered and running. The node will blink LED2 (and will pulse GPIO_12) every time it sends location data to Portal.

Begin logging

Once you have the scripts loaded and the GPS node knows Portal's address, select the Portal node and click the startNewFile() function. Portal will create a new data file, named for the current hour and minute. By default, this file is created in c:\RangeData\' but you can change that by updating the FILE_PATH constant in the GPS_Portal.py script.

Portal will begin logging location data received from the GPS node. Every time the GPS module sends the SNAP Engine a new location, the SNAP

Engine will relay that data to Portal. Portal will then write the data into a KML file. (It also displays latitude, longitude, altitude, number of satellites seen, satellite time, and a data point number in the Portal log.)

KML defines the standard file format used by Google Earth and Google Maps. That means you can import your SNAP-generated data set into Google Maps to precisely map each location.

Remember that if you do not have your GPS node tethered by a serial cable, the data sent to Portal will wirelessly travel through a SNAP mesh network. Your GPS unit can travel anywhere within range of your SNAP network and will faithfully report its position.

When you have received all the position data you wish to receive, click the closeExistingFile() function in Portal. This creates closing tags within the KML file so it will be complete and valid.

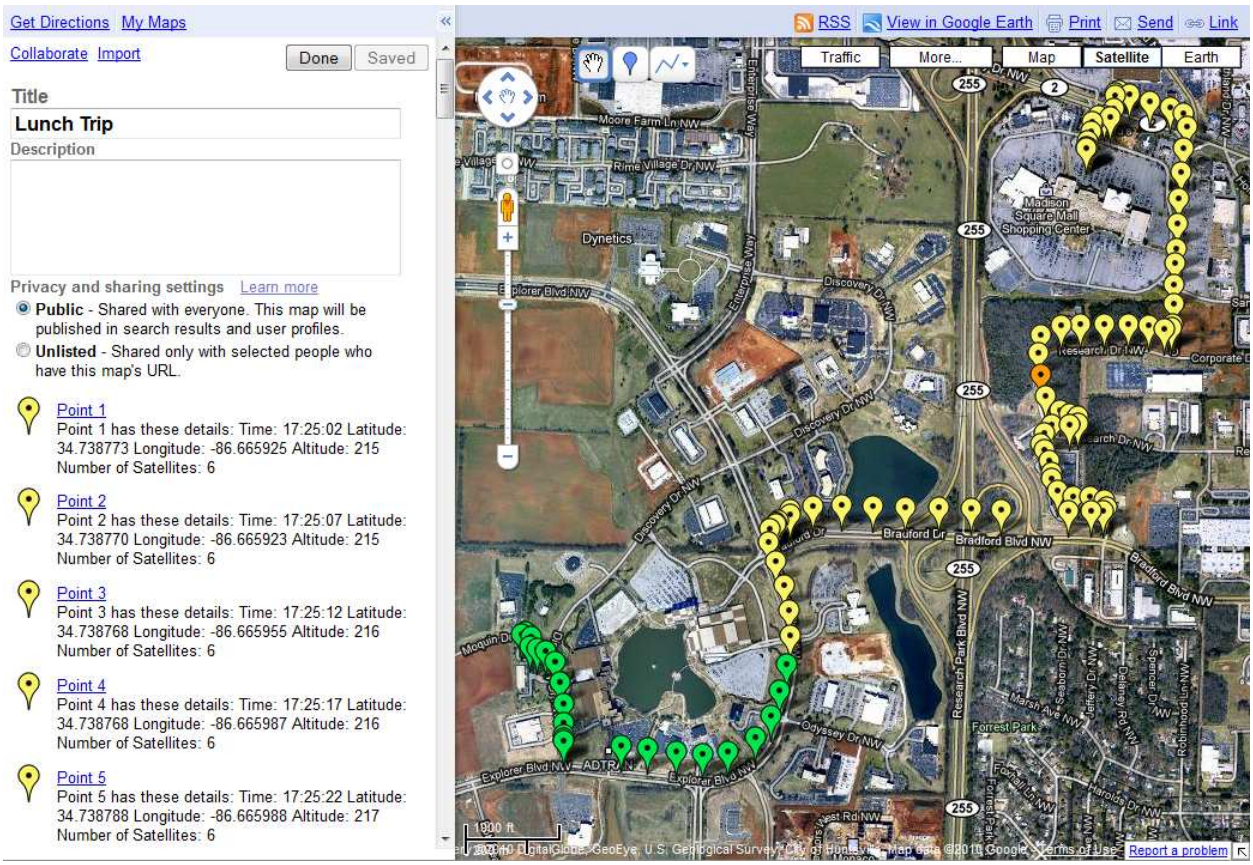
Viewing location information

Once you have all your data collected, log into Google Maps. (You can also import the data into Google Earth, if you have it installed.)

Click My Maps, and then click the Create new map link. Provide your new map with a name, and then click the Import link.

Browse to your data file and import it. Google Maps will plot each of your position points. Each point, if selected, will display its latitude, longitude, altitude, time, and number of satellites. The map "pins" will also be color-coded, indicating the number of satellites that were available for the reading. (This can be an indication of the reliability of the reading, as readings with fewer than four satellites responding can be erratic.)

Your resulting map will look something like the following:



What Makes it Work

The SNAP Engine Code

The code for this project (listed in Appendix 1) uses a few features that may not be immediately intuitive.

Private functions

For starters, you may notice that several of the functions in this script have an underscore at the beginning of their names (e.g., `_procToc()`, `_every100ms()`, and `_sendGpsInfo()`). In a SNAPpy script, starting a function name with an underscore makes the function "private." This means the function cannot be directly called by any external SNAP node, including Portal. Private functions can only be called by other functions within the same script. (You'll notice that the private functions, at the bottom of the list in Portal's Node Info pane, cannot be clicked in Portal to invoke them.)

You can use this to your advantage. SNAPpy limits you to 255 "public" functions in a script. In very long or very complex scripts, if you end up with too many functions you can privatize some of them that wouldn't be called

from outside the node. (This is rarely an issue.) You also might find that, because the private functions appear at the bottom of the list of function names when viewing your node in Python, you can use it to tidy up your function list, moving things that you wouldn't normally invoke directly out of the way and making it easier to find functions in the function list.

Character mode serial processing

The next unusual approach in the script is required because the GPS device outputs an NMEA-0183 standard message across its serial connection. The serial connection for this message is 4800 baud, 8N1, which is straightforward enough. However the NMEA signal coming in is too long to fit into a single string buffer, so SNAPpy is not able to receive the entire message in one piece. For example, the GPGGA format sent by the GPS device is:

$GPGGA,060055.000,0000.0000,N,00000.0000,E,0,00,99.0,0082.0,M,18.0,M,,*58

This message includes 15 bits of data, separated by commas and ending with a carriage return. But the complete message is 73 characters long, which the RF100 cannot accept.

Because of this, the `_startup()` function (hooked to the SNAP Engine's startup event) configures the standard input to be received in character mode, rather than line mode. This means the serial input handlers will be sending characters to SNAPpy as they receive them, rather than waiting for a complete buffer to be filled, or waiting for a carriage return or line feed.

Parsing the message

This method of receiving the characters colors the way the remainder of the SNAPpy script processes the data. It cannot simply take a long string and parse the pieces out of it. It must instead build its strings up one or two characters at a time. Each time it receives a comma or a carriage return (denoted in Python as '\r'), it knows it has another complete unit of some sort, so it also has to keep track of the last complete thing it received to know what the next thing it's expecting is.

This approach to remembering what the system is supposed to be doing in between function runs (triggered by the HOOK_STDIN event) is referred to as a "state machine." The system has a defined set of states it can end in, and moves from state to state based on the next information it receives.

Processing message tokens

In this script, the `state` global variable keeps track of the current state, and as characters are received they are appended to the `curTok` (current token) variable until the token delimiter (comma or carriage return) is received. Once a token is seen as complete (because the token delimiter is received), the token is sent off to the `_procTok()` function.

The `_procToc()` function receives the token and keeps track of what it has already heard from the current message string from the GPS unit. If it has already heard the time (state 2), it knows that the next thing it should expect would be the latitude (state 3). So it interprets the current token on the assumption it is a latitude.

Each token is processed by `_procTok()` (or, more precisely, is processed by another function called by `_procTok()` based on the current state), and the processed results are saved into a global variable until all the desired values (latitude [with pole], longitude [with meridian], number of satellites, altitude, and time) have been received. The `_procTok()` function then sets the `dataReadyToSend` variable to indicate it has one of everything and the data is ready to go.

Forwarding data to Portal

At this point, the global variables should all have GPS information in them. The `_every100ms()` function, hooked into the timer event that fires every tenth of a second, checks the `dataReadyToSend` flag (and the `timerReady` flag), and if all conditions have been met it invokes the `_sendGpsInfo()` function to pass the data to Portal's `getAll()` function, clear the variables, and start over.

The Portal Code

Once Portal receives the time, latitude, longitude, altitude and number of satellites in its `getAll()` function, it can process them in any way it likes. Remember that Portal runs a full Python implementation, rather than the SNAPpy subset, so it can write data files, save to databases, plot graphs, or do anything else you ask of it.

In this application, we want the Portal script to save the data into a KML file for upload into Google Maps or Google Earth.

KML files as XML data

A KML file is a data file in XML format that has specific data elements in it. As such, there is header data that must be established at the beginning of

each data file, and "close" tags must be added to the end of the file after all the "real" data has been written to it.

To implement this, the Portal script has a `startNewFile()` function and a `closeExistingFile()` function. The former creates a new data file (with a filename based on the PC's clock time when the file is created), and the latter closes out that file once all the data is received. Any call to `getAll()` by the GPS node between invoking `startNewFile()` and `closeExistingFile()` will result in one <Placemark> element being added to the data file. Each <Placemark> element represents one point on the Google Maps map.

Initializing the KML file

The `startNewFile()` function creates the new KML data file. This involves defining the <Document> and <Folder> tags, as well as creating the <Style> elements that will be used to color-code map data points based on the number of satellites that contributed to the GPS data. It calls the `writeStyle()` function once each for four colors to define those styles in the KML file. Once the file has been opened, the `fileIsOpen` variable is also set so Portal will know to start writing <Placemark> data points to the file.

Adding <Placemark> data

With the file open, the `getAll()` function (called by the node with the GPS unit) determines the correct color style based on the number of satellites and writes the data point to the KML file. It includes, as part of that data, an incrementing data point number. This number appears as the point's label if imported into Google Earth. In Google Maps, it is visible from the listing of data points to the left of the map. Each KML data point written by the `getAll()` function comprises one complete <Placemark> element in the XML.

Closing the KML file

When you have completed all the data collection you wish, the `closeExistingFile()` function closes the <Folder>, <Document>, and <kml> tags so the Google applications will recognize the file as valid. Loading the data into Google Maps or Google Earth, then, is pretty straightforward.

References

SNAP Reference Manual
EK2100 User's Guide
Portal Reference Manual

Appendix 1

GPS_Logger.py

Load this script into the SNAP Engine connected to the GPS device.

```
"""
GPS Interface: NMEA-0183 device connected to a ProtoBoard.
In this case, it is a Tyco A1035, which sends a serial NMEA signal at
4800 baud, 8N1.

Connect the GPS serial lines to GPIO_3 and GPIO_4 of your SNAP Engine.
(If using an RF300, you will need to use GPIO_7 and GPIO_8, as that Engine
has only one UART.) Connect the GPS power lines to VCC and Ground.

The serial NMEA signal is processed in character mode because the whole message
is too long to fit in a string buffer. Pieces are parsed out into their components
and sent to Portal, which logs them into a KML file. You can then load that file
into Google Maps to see where you ended up.

This code assumes it is running on a ProtoBoard. Connect a piezo buzzer between
GPIO_12 and ground to hear a beep each time the unit sends a GPS reading to Portal.
LED2 also pulses each time a GPS reading goes to Portal. LED1 pulses every second.
(Use the specifyPortalAddress() function if your Portal address is not the default.)
"""

from synapse.platforms import *
from synapse.switchboard import *

NV_PORTAL_ADDRESS = 213
WRITE_POINT_EVERY_SECS = 5

# Declare some globals
# Time
utc_hr = 0
utc_min = 0
utc_sec = 0

# Latittude
lat_deg = 0
lat_min = 0
lat_frac_min = 0
lat_pole = ' '

# Longitude
lng_deg = 0
lng_min = 0
lng_frac_min = 0
lng_meridian = ' '

fixStat = 0          # nonzero is valid fix
numSatellites = 0    # 0-12; at least 3 needed for lat/long, 4 for altitude
altitude = 0

state = 0
curTok = ''

dataReadyToSend = False
timerCount = 0
timerReady = False

@setHook(HOOK_100MS)
```



```
def _every100ms(tick):
    """
    Polls to see if the GPS has provided a complete set of data to forward
    to Portal. If the data is available, it sends it, blinks LED2, and
    pulses GPIO_12, expecting
    """
    global dataReadyToSend
    global timerReady

    if dataReadyToSend and timerReady:
        dataReadyToSend = False
        timerReady = False
        _sendGpsInfo()
        pulsePin(GPIO_12, 20, True)
        pulsePin(GPIO_2, 100, True)

@setHook(HOOK_1S)
def _everySecond(tick):
    """
    Invoked every second, this blinks LED1 as a visual "I'm alive!" indicator.
    Also, increments counter to determine if a point should be saved.
    """
    global timerReady
    global timerCount

    pulsePin(GPIO_1, 100, True)

    timerCount += 1
    timeCount %= WRITE_POINT_EVERY_SECS
    if timerCount == 0:
        timerReady = True

def specifyPortalAddress(portalAddress):
    """
    Call this function to specify Portal's address in your SNAP network.
    By default, Portal's address is '\x00\x00\x01' and you will only need
    to invoke this if you have changed that default value in Portal.
    """
    if portalAddress == None:
        portalAddress = '\x00\x00\x01'

    if len(portalAddress) != 3:
        portalAddress = '\x00\x00\x01'

    if ord(portalAddress[0]) != 0 or ord(portalAddress[1]) != 0:
        portalAddress = '\x00\x00\x01'

    saveNvParam(NV_PORTAL_ADDRESS, portalAddress)

@setHook(HOOK_STARTUP)
def _startup():
    """
    Configures the serial connection to the GPS device.
    Configures LEDs and the piezo buzzer for feedback.
    """
    initUart(0, 4800)
    stdinMode(1, False)   # Char mode, no echo

    # Connect GPS serial output to STDIN, where our event handler will parse the
messages
    crossConnect(DS_UART0, DS_STDIO)

    specifyPortalAddress(loadNvParam(NV_PORTAL_ADDRESS))
```



```
    setPinDir(GPIO_1, True)
    writePin(GPIO_1, False)
    setPinDir(GPIO_2, True)
    writePin(GPIO_2, False)
    setPinDir(GPIO_12, True)
    writePin(GPIO_12, False)

@setHook(HOOK_STDIN)
def _stdinEvent(buf):
    """
    Receive handler for character input on UART0.
    The parameter 'buf' will contain one or more received characters.
    The UART input is handled one (or a few) characters at a time because
    the complete NMEA sentence exceeds the size of some SNAPpy string buffers.
    """
    global state
    global curTok

    n = len(buf)
    i = 0
    while(i < n):
        c = buf[i]
        i += 1

        if len(curTok) > 20:
            state = 0

        if state == 0:
            # Look for 'start' delimiter
            if c == '$':
                state = 1
        else:
            # Look for 'token' delimiter
            if c == ',' or c == '\r':
                _procTok(curTok)
                curTok = ''
            else:
                # Accumulate characters to build next token
                curTok += c

def _sendGpsInfo():
    """
    Sends the collected GPS information to Portal
    """
    global lat_deg
    global lat_min
    global lat_frac_min
    global lat_pole
    global lng_deg
    global lng_min
    global lng_frac_min
    global lng_meridian
    global altitude
    global numSatellites

    portalAddress = loadNvParam(NV_PORTAL_ADDRESS)

    timeString = str(utc_hr) + ':'
    if utc_min < 10:
        timeString = timeString + '0'
    timeString = timeString + str(utc_min) + ':'
    if utc_sec < 10:
```



```
        timeString = timeString + '0'
    timeString = timeString + str(utc_sec)

    latString = str(lat_deg) + ' ' + str(lat_min) + '.' + str(lat_frac_min) + lat_pole
    lngString = str(lng_deg) + ' ' + str(lng_min) + '.' + str(lng_frac_min) +
lng_meridian
    altString = str(altitude)
    satString = str(numSatellites)

    rpc(portalAddress, 'getAll', timeString, latString, lngString, altString,
satString)

    lat_deg = lat_min = lat_frac_min = lng_deg = lng_min = lng_frac_min = altitude =
numSatellites = 0
    lat_pole = lng_meridian = ' '

def _parseTime():
    """
    Parse the time results
    """
    global utc_hr
    global utc_min
    global utc_sec
    utc_hr = int(curTok[:2])
    utc_min = int(curTok[2:4])
    utc_sec = int(curTok[4:6])

def _parseLat():
    """
    Parse the latitude details
    """
    global lat_deg, lat_min, lat_frac_min
    lat_deg = int(curTok[:2])
    lat_min = int(curTok[2:4])
    lat_frac_min = int(curTok[5:])

def _parseLon():
    """
    Parse the logitude details
    """
    global lng_deg, lng_min, lng_frac_min
    lng_deg = int(curTok[:3])
    lng_min = int(curTok[3:5])
    lng_frac_min = int(curTok[6:])

def _procTok(curToken):
    """
    Process NMEA tokens
    """
    global state
    global lat_pole
    global lng_meridian
    global fixStat
    global numSatellites
    global altitude
    global dataReadyToSend

    # States:
    #   0 - Idle
    #   1 - Sentence ID
    #   2 - GPGGA: time
    #   3 - GPGGA: latitude
    #   4 - GPGGA: N/S
```



```
#   5 - GPGGA: longitude
#   6 - GPGGA: E/W
#   7 - GPGGA: Fix
#   8 - GPGGA: # satellites
#   9 - GPGGA: HDOP
#   10 - GPGGA: altitude

if state == 1:
    if curToken == 'GPGGA':
        state = 2
    elif curToken == 'GPRMC':
        state = 20
    elif curToken == 'GPGSA':
        state = 40
    else:
        state = 0
elif state == 2:
    if len(curToken) == 10:
        _parseTime()
        state = 3
    else:
        state = 0
elif state == 3:
    if len(curToken) == 9:
        _parseLat()
        state = 4
    else:
        state = 0
elif state == 4:
    if len(curToken) == 1:
        lat_pole = 'N' if curToken == 'N' else 'S'
        state = 5
    else:
        state = 0
elif state == 5:
    if len(curToken) == 10:
        _parseLon()
        state = 6
    else:
        state = 0
elif state == 6:
    if len(curToken) == 1:
        lng_meridian = 'E' if curToken == 'E' else 'W'
        state = 7
    else:
        state = 0
elif state == 7:
    fixStat = int(curToken)
    state = 8
elif state == 8:
    numSatellites = int(curToken)
    state = 9
elif state == 9:
    # Discard HDOP
    state = 10
elif state == 10:
    altitude = int(curToken)
    dataReadyToSend = True
    state = 0
else:
    state = 0
```


GPS_Portal

Load this script into Portal to perform the logging.

```
"""
This is part of the suite of position scripts. This script is to be loaded into
Portal.

Load the GPS_Logger script into a node on a ProtoBoard with a GPS device attached.
Make
a serial connection from Portal to that node.

Every time the GPS unit receives a GPS position it sends that information to Portal,
which logs the data to a text file in the C:\RangeData directory.

You can then load that file into Google Maps to see where you ended up.

Data logging does not begin until you invoke the startNewFile() function. When you
are done, invoke the closeExistingFile() function to close out the KML file.
"""

from datetime import datetime
import os

def getAll(timeString, latString, lngString, altString, satString):
    global pointNumber
    stringToWrite = latString + ', ' + lngString + ', ' + altString + ', ' + satString
+ ', ' + timeString + ', ' + str(pointNumber) + '\n'
    print stringToWrite

    if fileIsOpen and latString != '0 0.0 ' and lngString != '0 0.0 ' and altString !=
'0' and satString != '0':
        # The reported information is in sync. Save it!
        pointNumber += 1

        # Save the KML data.
        theLat = latString.split(' ')
        theLng = lngString.split(' ')

        theLat.append(theLat[1][-1])
        theLat[0] = float(theLat[0])
        theLat[1] = float(theLat[1][:-1])

        theLng.append(theLng[1][-1])
        theLng[0] = float(theLng[0])
        theLng[1] = float(theLng[1][:-1])

        latitude = theLat[0] + theLat[1] / 60
        longitude = theLng[0] + theLng[1] / 60

        numSats = int(satString)

        if theLng[2] == 'W':
            longitude = 0 - longitude
        if theLat[2] == 'S':
            latitude = 0 - latitude

        if numSats <= RED_THRESHHOLD:
            pinColor = 'red'
        elif numSats <= ORANGE_THRESHHOLD:
            pinColor = 'orange'
        elif numSats <= YELLOW_THRESHHOLD:
            pinColor = 'yellow'
```



```
        else:
            pinColor = 'green'

        f = open(FILE_PATH + 'data' + startTimeString + '.kml', 'a')
        f.write('                <Placemark>\n')
        f.write('                    <name>Point %i</name>\n' % (pointNumber, ))
        f.write('                    <styleUrl>#%sStyle</styleUrl>\n' % (pinColor, ))
        f.write('                    <description>Point %i has these details:\n'
%(pointNumber, ))
        f.write('                                    Time: %s\n' % (timeString, ))
        f.write('                                    Latitude: %f\n' % (latitude, ))
        f.write('                                    Longitude: %f\n' % (longitude, ))
        f.write('                                    Altitude: %s\n' % (altString, ))
        f.write('                                    Number of Satellites: %s\n' %
(satString, ))
        f.write('                    </description>\n')
        f.write('                    <Point>\n')
        f.write('                        <coordinates>%f,%f,%s</coordinates>\n' %
(longitude, latitude, altString))
        f.write('                    </Point>\n')
        f.write('                </Placemark>\n')
        f.close()


def startNewFile():
    global fileIsOpen
    global pointNumber

    setTimeString()
    # Create kml file for Google Maps upload
    f = open(FILE_PATH + 'data' + startTimeString + '.kml', 'w')
    f.write('<?xml version="1.0" encoding="UTF-8"?>\n')
    f.write('<kml xmlns="http://www.opengis.net/kml/2.2">\n')
    f.write('    <Document>\n')
    writeStyle('red', 'ff0000ff', f)
    writeStyle('orange', 'ff007fff', f)
    writeStyle('yellow', 'ff00ffff', f)
    writeStyle('green', 'ff00ff00', f)
    f.write('        <Folder>\n')
    f.write('            <name>GPS position file started at %s</name>\n' %
(startTimeString, ))
    f.write('            <description>This GPS tracking record was run starting at
%s.</description>\n' % (startTimeString, ))
    f.close()
    fileIsOpen = True
    pointNumber = 0

def writeStyle(color, hex, f):
    f.write('        <Style id="%sStyle">\n' % (color, ))
    f.write('            <IconStyle id="pin%s">\n' % (color, ))
    f.write('                <color>%s</color>\n' % (hex, ))
    f.write('                <colorMode>normal</colorMode>\n')
    f.write('                <scale>1</scale>\n')
    f.write('                <Icon>\n')
    f.write('                    <href>http://maps.google.com/mapfiles/ms/micons/%s-
dot.png</href>\n' % (color, ))
    f.write('                </Icon>\n')
    f.write('                <heading>0</heading>\n')
    f.write('            </IconStyle>\n')
    f.write('        </Style>\n')

def closeExistingFile():
    global fileIsOpen
```



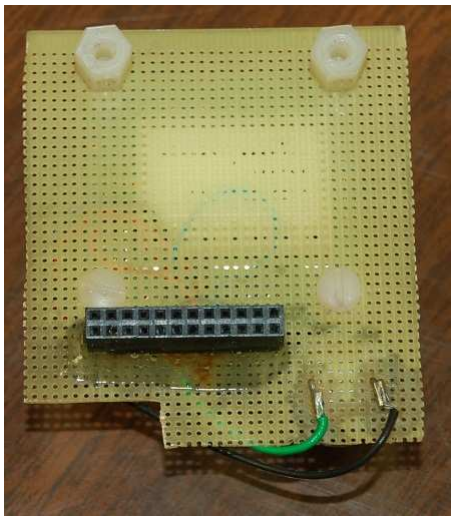
```
    global pointNumber

    pointNumber = 0

    if fileIsOpen:
        f = open(FILE_PATH + 'data' + startTimeString + '.kml', 'a')
        f.write('        </Folder>\n')
        f.write('    </Document>\n')
        f.write('</kml>')
        f.close()
        fileIsOpen = False

def setTimeString():
    global startTimeString
    startDateTime = datetime.now()
    startDateTuple = startDateTime.timetuple()
    startHour = str(startDateTuple[3])
    startMinute = str(startDateTuple[4])
    if len(startMinute) == 1:
        startMinute = '0' + startMinute
    startTimeString = startHour + '_' + startMinute


# Run at startup
setTimeString()
pointNumber = 0
fileIsOpen = False

RED_THRESHHOLD = 3
ORANGE_THRESHHOLD = 5
YELLOW_THRESHHOLD = 7

FILE_PATH = 'c:\\RangeData\\'
if not os.access(FILE_PATH, os.F_OK):
    os.makedirs(FILE_PATH)
```

Appendix 2

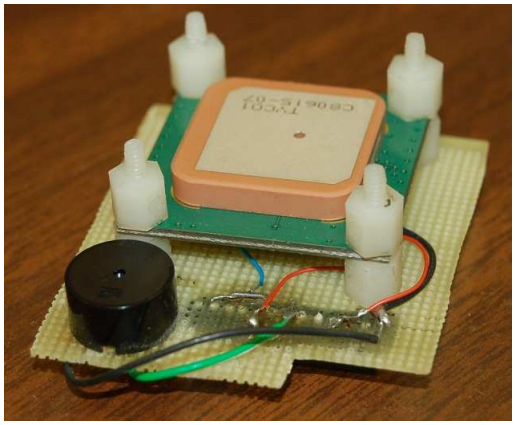
A SNAP-friendly GPS device

You can connect your GPS device to a Synapse ProtoBoard by connecting the wires through the terminal block, as shown at the beginning of this document. If you want a more robust and modular package, you might want to construct a more permanent fixture to allow you to easily attach the GPS device.

One approach to solving this takes advantage of jumper J2 on the ProtoBoard, which has 24 pins mapping to the 24 pins on the RF Engine. A 24-pin connector, mounted to a small piece of perf board, provides a stable platform for mounting the GPS device so the entire board can be quickly and correctly connected to the ProtoBoard.

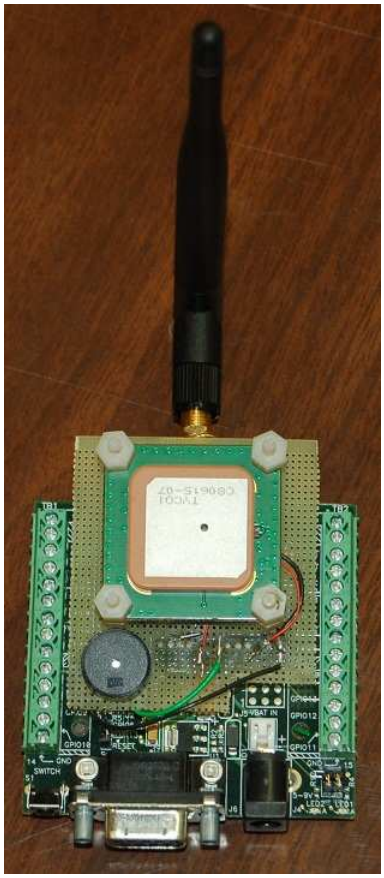


The image at left shows a 24-pin connector mounted to a piece of perfboard. The green and black wires, as you will see, connect to a piezo buzzer on the other side of the board. A little bit of epoxy helps hold the connector in place.



The top of the perfboard shows how the piezo buzzer and GPS device connect to the pins from the 24-pin connector. (Unused pins were removed from the connector before it was attached to the board.) This device requires only pins 21 and 24 for power, pin 14 for the buzzer (GPIO_12), and pins 5 and 6 for the serial connection to UART0 on the SNAP Engine.

With the GPS device mounted to the perfboard, it's an easy matter to affix it to a ProtoBoard with a SNAP Engine. The GPS module is easily removed when you need to retask your SNAP hardware, and easily reattached at a later date without having to go looking for pinout diagrams to remember which wire went where.



It may be useful to use a rubber band or zip strip to hold the GPS module to the ProtoBoard, if you expect to be moving the unit around.

License governing any code samples presented in this Application Note

Redistribution of code and use in source and binary forms, with or without modification, are permitted provided that it retains the copyright notice, operates only on SNAP® networks, and the paragraphs below in the documentation and/or other materials are provided with the distribution:

Neither the name of Synapse nor the names of contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SYNAPSE AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SYNAPSE OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF SYNAPSE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Disclaimers

Information contained in this Release Note is provided in connection with Synapse products and services and is intended solely to assist its customers. Synapse reserves the right to make changes at any time and without notice. Synapse assumes no liability whatsoever for the contents of this Release Note or the redistribution as permitted by the foregoing Limited License. The terms and conditions governing the sale or use of Synapse products is expressly contained in the Synapse's Terms and Condition for the sale of those respective products.

Synapse retains the right to make changes to any product specification at any time without notice or liability to prior users, contributors, or recipients of redistributed versions of this Release Note. Errata should be checked on any product referenced.

Synapse and the Synapse logo are registered trademarks of Synapse. All other trademarks are the property of their owners.

For further information on any Synapse product or service, contact us at:

Synapse Wireless, Inc.
500 Discovery Drive
Huntsville, Alabama 35806

256-852-7888
877-982-7888
256-852-7862 (fax)

www.synapse-wireless.com

