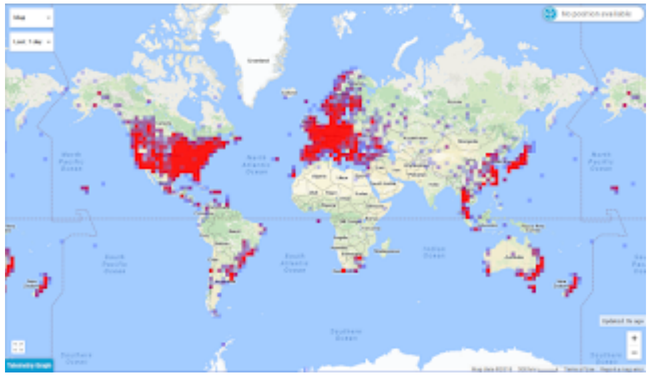
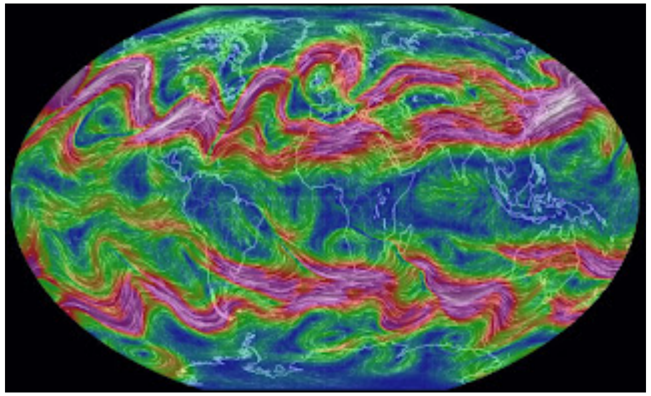
TT7 High Altitude Balloon

Wednesday 15 February 2017

APRS Automatic Packet Reporting System



APRS has been used on a number of balloons floating around the Earth. Its usefulness lies in an existing network of receivers that automatically repeat and upload the received telemetry packets to the Internet. Above is a map of digipeaters and IGates that were operating in the last 24 hours before the image was taken.



About Me

TT7

View my complete profile

Archive

The high altitude winds (earth.nullschool.net image 250hPa roughly 10000m) generally take the balloons from west to east holding them in either the Northern hemisphere or the Southern. These patterns very often take a balloon launched in Europe and floating at around 12km to periodically fly above these continental 'hot spots' that provide reception. In practice the balloon every now and then sends a message containing its position (and other information) on an APRS regional frequency. If it is nearby an APRS station, it either gets retransmitted and reaches further, in case of the receiving station being a digipeater, or it gets uploaded onto the Internet, in case of the station being an IGate. All this works without the need of actively tracking the balloon.
For these reasons I decided to implement APRS transmissions in my TT7F tracker as well. Since a fully functioning implementation requires a number of steps some of which took me a fairly long time to make work properly, I wanted to describe the final solution in a little more detail.

Protocol
Probably the basic source of information is the APRS Protocol Reference 101. The problem with the APRS documentation is that it is quite scattered all around www.aprs.org (and elsewhere) in not that user-friendly fashion. Some of it is outdated and to find out what the current way of doing things is may take a lot of digging around. It's been useful to me going to http://aprs.fi and simply watching what sort of packets people send these days and then checking it in the documentation.

Field	Bytes	Hex
Flag	1	0x7E
Destination Address	7	
Source Address	7	
Path	0-56	
Control Field	1	0x03
Protocol ID	1	0xF0
Information Field	1-256	
FCS	2	
Flag	1	0x7E

This is the basic packet structure. A packet starts and ends with a 0x7E **Flag** which in binary translates to 01111110.

Flickr

TT7 Flicker Account.

Links

Flickr TT7

GitHub TT7

UKHAS

HabHub

Habitat Tracker

IRC #highaltitude

CUSF Landing Predictor

Hourly Predictor - Roznov

The six '1' bits in a row is a pattern reserved to inform the receiver/decoder about the packet boundaries. In any other case a series of five or more '1' bits is stuffed with an extra '0' bit (more on that later) before the remaining bits follow. The packet is generally preceded by a number of these 0x7E Flags to help the receiver/decoder lock onto the signal.

1st	2nd	3rd	4th	5th	6th	7th
A	P	R	S			p

The **Destination Address** field is fixed to 7 bytes in length and is used to identify an APRS packet among other AX.25 frames. The first six bytes constitute the address (generally starting AP****) while the seventh byte represents an SSID. This last byte's structure in binary is as follows: 0b0CRRSSID where 'C' command/response bit '1', 'RR' reserved '11', 'SSID' value of 0-15. In the upper example the first six bytes correspond to 'APRS ' string of ASCII characters with an SSID byte '0b01110000' corresponding to the 'p' ASCII character. The list of possible addresses is in the tocalls.txt document. I use the generic 'APRS ' call while based on the list a couple of balloonists have apparently asked for their own destination addresses for their projects.

1st	2nd	3rd	4th	5th	6th	7th
N	0	C	A	L	L	;

The **Source Address** field represents the callsign of the sender and likewise constitutes of six ASCII characters for the callsign itself and one SSID byte. In contrast to the destination field I generally use '0b00111011' switching the command/response bit to '0' and 'SSID' to 11 (0b1011) which corresponds to a balloon symbol shown on map. The above example would be displayed as N0CALL-11 after reception.

1st	2nd	3rd	4th	5th	6th	7th
W	I	D	E	2		1

When building a transmitter, the **Path** field is used to define the number of times the packet should be received and retransmitted by the receiving stations. When retransmitting the packet, the station automatically alters the Path field to reflect that one retransmission was used up by it. The field also specifies which type of station should repeat this packet.

Path	Retransmitted by	Retransmissions

WIDE1 1	any station	one
WIDE2 1	only high-level digipeater	one
WIDE2 2	only high-level digipeater	two
WIDE1 1WIDE2 1	first any station, second only high-level digipeater	two

Since the purpose of a balloon's transmitter is to reach an IGate that uploads the packet onto the Internet, and due to its travelling altitude it is capable of reaching a number of stations down bellow, it is recommended to use either 'WIDE2 1' or no path at all. In case of 'WIDE2 1', if a high-level digipeater receives the packet, it changes the path to 'WIDE2*' marking the retransmission as used up.

The **Control Field** is set to 0x03 and identifies the packet as an Unnumbered Information frame within the AX.25 protocol.

The **Protocol ID** is set to 0xF0 and informs that there is no layer 3 protocol implemented.

APRS Data Type Identifier	Data Type
!	position without timestamp
/	position with timestamp
:	message
T	telemetry data (uncompressed)

The **Information Field** initialized by the APRS Data Type Identifier holds the data to be transmitted. The protocol offers a number of data types. I will focus only on those I use.

/210048h4916.54N/01814.58EOTT7Fhab

Identifier	Time	Latitude	Symbol Table Identifier	Longitude	Symbol	Comment
/	210048h	4916.54N	/	01814.58E	O	TT7F hab

This kind of a message initiated by the position with timestamp data type identifier displays time, latitude and longitude (degrees and decimal minutes) in human readable format. Symbol Table Identifier and Symbol then define under which symbol the information should be displayed on a map. In this case a balloon. Following the symbol the message leaves space for an optional comment.

```
/A=001319
```

If desired, altitude can be placed in the Comment section in the format above (402m expressed in feet).

```
T#005,1275,2533,1005,1492,9,11000000
```

Identifier	Sequence	Value1	Value2	Value3	Value4	Value5	Bitfield
T	#005,	1275,	2533,	1005,	1492,	9,	11000000

To send telemetry in human readable format use the 'T' identifier. The data type then offers space for 5 analog values and one 8-bit bitfield. The Sequence number takes care of ordering the received packets.

```
:N0CALL-11:PARM.Vsol,Vbatt,Tcpu,Ttx,Sats,Nav,Fix
```

Identifier	Addressee	Message	Label	Label	Label	Label	Label	Label	Label
:	N0CALL-11	:PARM.	Vsol,	Vbatt,	Tcpu,	Ttx,	Sats,	Nav,	Fix

```
:N0CALL-11:UNIT.V,V,C,C
```

Identifier	Addressee	Message	Unit	Unit	Unit	Unit
:	N0CALL-11	:UNIT.	V,	V,	C,	C

```
:N0CALL-11:EQNS.0,0.0008,0,0,0.0016,0,0,0.304,-263,0,0.222,-297,0,1,0
```

Id.	Addressee	Message	Coeff.	Coeff.	Coeff.	Coeff.	Coeff.

:	N0CALL-11	:EQNS.	0,0.0008,0,	0,0.0016,0,	0,0.304,-263,	0,0.222,-297,	0,1,0

```
:N0CALL-11:BITS.11111111,TT7F HAB
```

Id.	Addressee	Message	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7	Bit8	Project
:	N0CALL-11	:BITS.	1	1	1	1	1	1	1	1,	TT7F HAB

These four messages are closely tied to telemetry. They allow configuring the receiving end to interpret the data as desired. This means the telemetry itself can be sent directly as the value output by an ADC with the conversion to the actual floating point value done on the receiver's side. The packets use the ':' Message data type identifier while the 'addressee' field contains the callsign that will be sending the telemetry. The 'PARM' message names the individual analog and digital channels. The 'UNIT' message informs about the units in which the values are expressed. The 'EQNS' message provides three coefficients (a, b, c) for each of the individual analog channels to use in the following equation to properly interpret the value:

$$a \times v^2 + b \times v + c$$

The 'BITS' message then defines the polarity of individual bits in the bitfield and provides the name of the project. In the case above I label all five analog values and two digital bits. I set units only to the first four analog values. I send coefficients to all 5 analog values (3 coeffs each). Lastly I name the project and set the active polarity of all 8 bits being '1'.

```
!/5LEGS*-/ON3W |!$1B<m,%1E!(!$|
```

Id.		Lat	Lon	Sym	Alt				Seq	Val1	Val2	Val3	Val4	Val5	Bits	
!	/	5LEG	S*-/	O	N3	W		|	!$	1B	<m	,%	1E	!(	!$	|

In many cases shorter packets may be preferred to human readability. For that APRS offers **Base91** compressed format. In this case the position without timestamp data type identifier '!' and the symbol table identifier '/' are followed by 4 bytes that hold encoded latitude according to the following calculation:

for 49.4913
(90 - 49.4913) x 380926

$15430817 / 91^3 = 20$ remainder 359397, $20 + 33 = 55$ -> **'5'**
$359397 / 91^2 = 43$ remainder 3314, $43 + 33 = 76$ -> **'L'**
$3314 / 91 = 36$ remainder 38, $36 + 33 = 69$ -> **'E'**
$38 + 33 = 71$ -> **'G'**

The next 4 bytes hold encoded longitude according to a similar calculation:

for 18.2232
$(180 + 18.2232) \times 190463$
$37754185 / 91^3 = 50$ remainder 75635, $50 + 33 = 83$ -> **'S'**
$75635 / 91^2 = 9$ remainder 1106, $9 + 33 = 42$ -> **'*'**
$1106 / 91 = 12$ remainder 14, $12 + 33 = 45$ -> **'-'**
$14 + 33 = 47$ -> **'/'**

To decode, simply revers the calculation. The next byte represents the symbol to be shown on a map (balloon) followed by 2 bytes representing altitude according to this calculation:

for 1131 meters
$1131 \times 3.2808 = 3710$
$\log(3710) / \log(1.002) = 4113$
$4113 / 91 = 45$ remainder 18, $45 + 33 = 78$ -> **'N'**
$18 + 33 = 51$ -> **'3'**

The reason for adding 33 to all the results is to push the values up to be represented by a human readable ASCII character in the range between '!' and '{' included. The last required byte in this case is 'W' representing a bitfield that serves to select altitude to be represented by the two previous bytes (more on that in the APRS documentation). That concludes the required part of the data type and whatever follows is placed inside the optional comment field. The compressed telemetry is delimited by the two '|' characters and should be at the end of the comment field. In between are Base91 encoded pairs of bytes representing Sequence, Value1, Value2, Value3, Value4, Value5 and the 8-bit bitfield. It is not necessary to use all 5 channels plus the bitfield. The minimum can be as short as just the sequence and value1 '|!$1B|'. The individual analog channels can range from 0 to 8280 and are encoded according to the following calculation:

for 1489
$1489 / 91 = 16$ remainder 33, $16 + 33 = 49$ -> **'1'**
$33 + 33 = 66$ -> **'B'**

In case of sending a comment within this packet as well, it should precede the compressed telemetry. In the example above the space ' ' character is essentially a comment. The data upon reception may be interpreted according to the four telemetry configuration messages just like the uncompressed telemetry. The advantage of this kind of a packet is that it includes both position and telemetry in one transmission.

FCS corresponds to Frame-Check Sequence and the two bytes hold a 16-bit result of a CRC-16-CCITT calculation ran on all the previous bytes of the packet omitting the 0x7E flags. FCS is sent low byte first and the bits are flipped.

Software Implementation
ARM_APRS.h
ARM_APRS.c
Having some understanding of the protocol, let's look into constructing a whole packet and preparing it for transmission. The functions facilitating this in TT7F can be found in the library above. If all required data, such as GPS and ADC readings, have been prepared, **APRS_packet_construct()** can be called while passing it a desired output buffer.

0x7E
01111110

First it fills the buffer with a specified number of 0x7E flags. As mentioned before this bit pattern is a signal to the receiver/decoder and doesn't appear in any other place within the transmitted bitstream other than the beginning and the end of the packet.

```
// Left Shifting the Address Bytes
for(uint16_t i = APRSFLAGS; i < num; i++)
{
    buffer[i] <<= 1;
    if(i == (num - 1)) buffer[i] |= 0x01;
}
```

Proceeding with adding the destination address, source address and path bytes into the buffer comes the first necessary bit manipulation. All these bytes need to be **left shifted by one bit**.

A	P	R	S			p

01000001	01010000	01010010	01010011	00100000	00100000	01110000
10000010	**10100000**	**10100100**	**10100110**	**01000000**	**01000000**	**11100000**

N	0	C	A	L	L	;
01001110	00110000	01000011	01000001	01001100	01001100	00111011
10011100	**01100000**	**10000110**	**10000010**	**10011000**	**10011000**	**01110110**

W	I	D	E	2		1
01010111	01001001	01000100	01000101	00110010	00100000	00110001
10101110	**10010010**	**10001000**	**10001010**	**01100100**	**01000000**	**01100011**

On top of that **the lowest bit of the last byte has to be set to '1'** to signal the end of the address/path fields. That corresponds to the last byte in path (WIDE2 **1**) in the above example. In case of not using any path, the last byte of the source address becomes the byte of interest.

0x03	0xF0
00000011	11110000

The next two bytes to add inside the buffer are the Control field and the Protocol ID.

```
if(APRS_packet_mode == 1) // !/5LEJS*-/ON3W|!$1B<m,%1E!(!$|
{
    buffer[num++] = '!';
    APRS_position_base91(buffer, APRSlatitude, APRSlongitude, (float)APRSaltitude, 1);
    APRS_telemetry_base91(buffer, APRSsequence, APRSvalue1, APRSvalue2, APRSvalue3, APRSval
}
```

Now comes the time for the Information field and the decision on what content I want to send. I've written this bit of the function in a way that allows me to easily switch the desired content by changing the **APRS_packet_mode** global variable. The option above constructs a Base91 encoded position and telemetry without a timestamp data and inserts it into the buffer.

```
else if(APRS_packet_mode == 2) // :PARM.Vsol,Vbatt,Tcpu,Ttx,Sats,Nav,Fix
{
    APRS_telemetry_PARM(buffer, "Vsol,Vbatt,Tcpu,Ttx,Sats,Nav,Fix", "N0CALL-11");
}
```

This option on the other hand constructs the PARM telemetry configuration message.

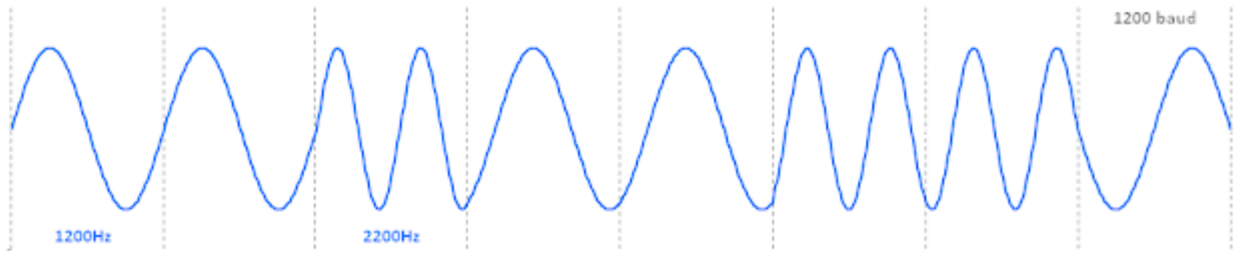
```
// Frame Check Sequence - CRC-16-CCITT (0xFFFF)
uint16_t crc = 0xFFFF;
for(uint16_t i = 0; i < (num - APRSFLAGS); i++) crc = crc_ccitt_update(crc, buffer[APRSFLAG
crc = ~crc;                             // flip the bits
buffer[num++] = crc & 0xFF;             // FCS is sent low-byte first
buffer[num++] = (crc >> 8) & 0xFF;
```

To verify that the data arrived to the receiver uncorrupted, the function runs a Frame Check Sequence on all the previous bytes except the initial 0x7E flags and appends the result in the form of two bytes to the end of the data inside the buffer. The 16-bit result has its **bits flipped** and is appended **low-byte first**.

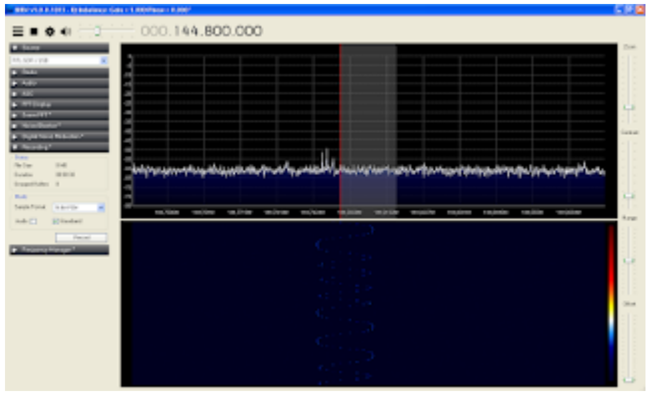
0x7E	0x7E
01111110	01111110

To finalize the bitstream a couple of 0x7E flags is inserted. At the end of this process there is a complete APRS packet in the buffer.

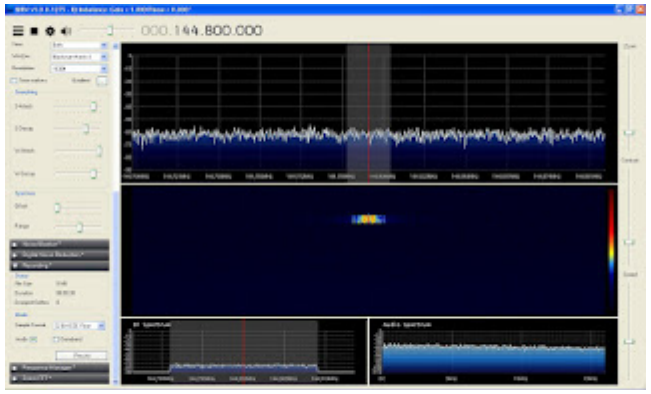
Modulation
ARM_SI4060.h
ARM_SI4060.c
In the 2-meter band APRS is transmitted using 1200 baud AFSK modulation that makes use of 1200Hz and 2200Hz tones. TT7F is equipped with Si4060 transmitter that doesn't support AFSK, however, it offers CW and GFSK modulations which are utilized.



The lower 1200Hz tone fits exactly into one baud, however, the faster 2200Hz tone isn't an integer multiple of the baud rate thus phase continuity should be handled during transitions between the tones. The illustration above shows the individual bauds initially starting at the same phase and as the transitions to the 2200Hz tone and back come about the following bauds begin at different phases.



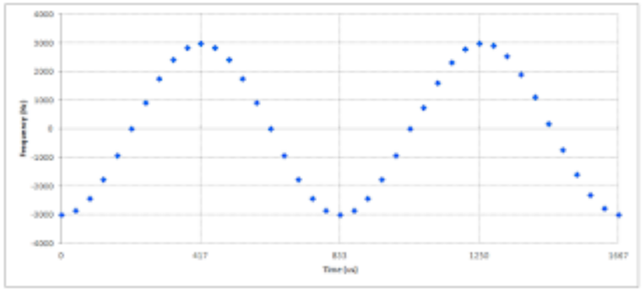
Just for a better idea, this is a significantly slowed-down shape of the modulation. Both the 1200Hz and 2200Hz sine waves are visible.



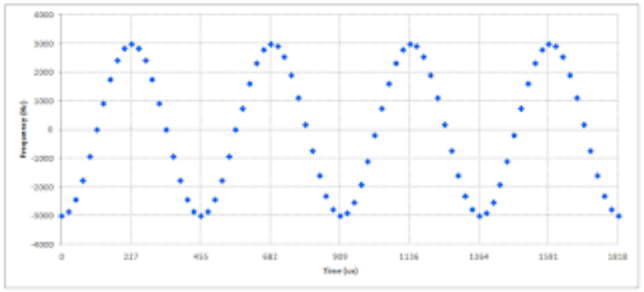
And here is an APRS packet in real time. 77 bytes in 513ms.

0:00 / 0:00

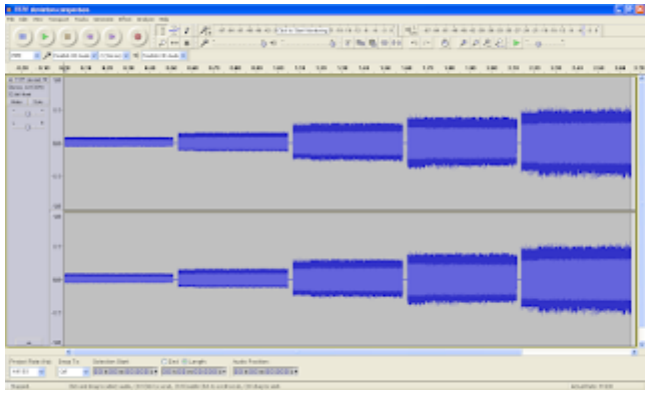
An example packet generated by Direwolf for download: TT7F example packet.wav. The audio signal can be examined more closely in Audacity. I used that a lot when working on the modulation.



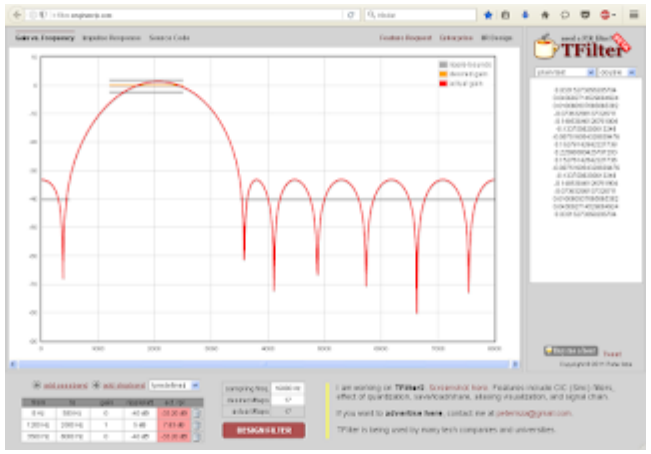
Now, how to achieve this with Si4060? One option is the provided **GFSK modulation** which smooths the transition between two programmed frequencies (-3kHz and +3kHz with respect to the carrier frequency - the illustration above). Simple binary FSK would just jump from one frequency to the other. Using the TimerCounter0 (TC0) running at 26400Hz (this rate is a common multiple of 2400Hz and 4400Hz which are two times the desired tones - 1200Hz and 2200Hz), I can periodically switch between the two frequencies and create an imitation of the AFSK modulation. The switching is done via the GPIO1 pin in a TX DIRECT mode and operated by the TC0's interrupt routine. Every 11th interrupt in case of the 1200Hz tone or 6th in case of the 2200Hz tone switches the pin from HIGH to LOW or vice versa.



I use the TC0's interrupt to oversee the next baud timing as well. Every 22nd interrupt sets a flag and informs the current while loop to move on to the next bit and adjust the transmitted tone accordingly.



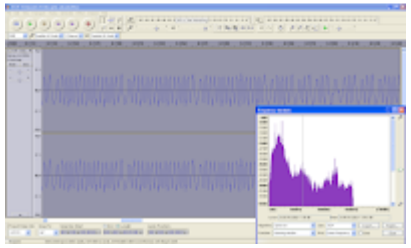
One of the parameters needed to be set is the frequency deviation. In this case the programmed deviation determines the amplitude of the demodulated audio signal as can be seen in the Audacity screen capture. The above are identical APRS packets using different deviations.

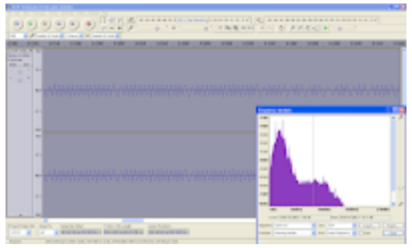


Another piece of the puzzle is a 17-tap finite impulse response (FIR) filter that is responsible for shaping the square wave of the original signal into the GFSK modulation. Since the filter is symmetrical, there are 9 values for the filter's coefficients to be set. The UBSEDS team provided a useful and enlightening notes on FIR filters in their Github repository. I tried to follow it and came up with a few filters of my own but none matched nor surpassed the performance of the UBSEDS filter coefficients, so in the end TT7F uses their filter values.

```
void SI4060_tx_APRS_GFSK_sync(void)
{
        SI4060_modulation(3, 0);    // GFSK synchronous modulation
        SI4060_data_rate(12000);    // 12000 (1200Hz), 22000 (2200Hz)

        SI4060_filter_coeffs();     // set up the FIR filter
        SI4060_frequency(APRS_tx_frequency);
        SI4060_frequency_deviation(TX_DEVIATION_APRS_1200);

        SI4060_power_level(SI4060_TX_power);
        SI4060_change_state(0x07);

        TC0_init_APRS_GFSK_sync(); // TC0 at 26400Hz

        ...

        while(APRSpacketReady)      // switching tones, processing the packet
        {
                ...
        }
        ...
}
```

The software implementation then works as follows. After initializing both SPI (ideally ran at 10MHz - the maximum Si4060's SPI speed) and Si4060 and constructing the APRS packet, SI4060_tx_GFSK_sync() function is called to transmit the packet. Inside, the modulation is set to synchronous GFSK. The data rate setting for GFSK needs to be a multiple of the desired data rate to properly synthesize the individual steps of the modulation. It is dependent on the TXOSR field in the MODEM_TX_NCO_MODE register. In this case the data rate is initially set to 12000 with the oversampling ratio set to 10 and the desired data rate being 1200Hz. Choosing 22000 instead simply changes the initial tone of the transmission to the higher 2200Hz. Following with setting the FIR filter coefficients, local APRS frequency and the deviation of the two frequencies as described above, the transmission is switched on followed by the TimerCounter0 initialization. Since then the code is in a while loop that responds to flags from the TC0's interrupt switching the tones as it steps through the individual bits. Whenever there is a change in tone required (1200Hz to 2200Hz or vice versa), the code changes the data rate and the timing of whether to respond to every 6th (2200Hz) or 11th (1200Hz) interrupt and toggle the GPIO1.

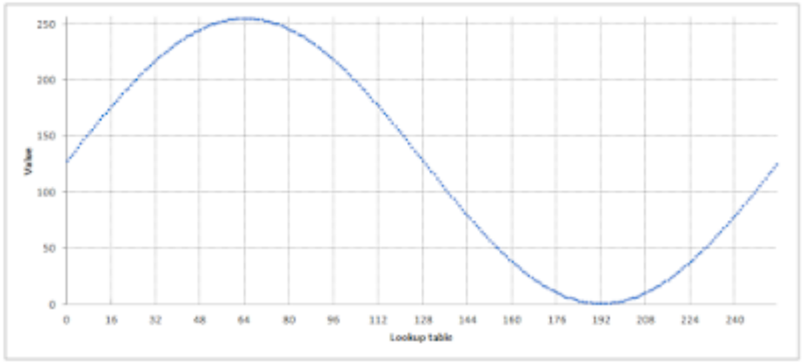




These are the resulting signals as transmitted by TT7F and received by an SDR dongle examined up close in Audacity. The first one uses UBSEDS filter coefficients the second then one of mine attempts for comparison. It seems the second one attenuates the out of band frequencies better, but at the same time it attenuates the pass band as well. Here is the audio for download: TT7F APRS GFSK (UBSEDS_fir).wav.

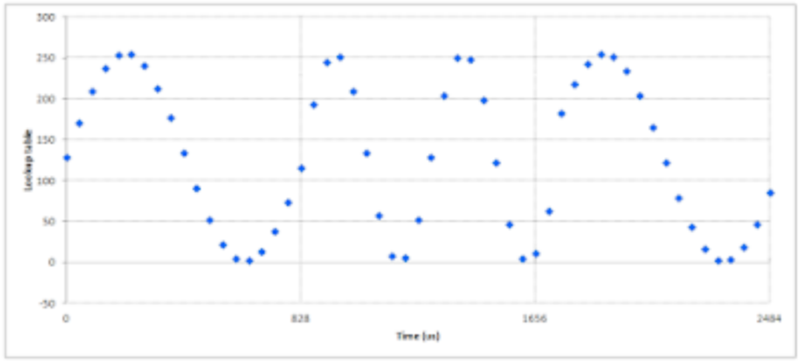
0:00 / 0:00

As can be seen from the way it is implemented and the frequency analysis, the signal shape is only an approximation of the desired AFSK modulation. For example the higher tone is more towards 2400Hz rather than 2200Hz. Nevertheless it decodes successfully.



```
static uint8_t SineLookUp[] = {
        128,131,134,137,140,144,147,150,153,156,159,162,165,168,171,174,
        177,179,182,185,188,191,193,196,199,201,204,206,209,211,213,216,
        218,220,222,224,226,228,230,232,234,235,237,239,240,241,243,244,
        245,246,248,249,250,250,251,252,253,253,254,254,254,255,255,255,
        255,255,255,255,254,254,254,253,253,252,251,250,250,249,248,246,
        245,244,243,241,240,239,237,235,234,232,230,228,226,224,222,220,
        218,216,213,211,209,206,204,201,199,196,193,191,188,185,182,179,
        177,174,171,168,165,162,159,156,153,150,147,144,140,137,134,131,
        128,125,122,119,116,112,109,106,103,100, 97, 94, 91, 88, 85, 82,
         79, 77, 74, 71, 68, 65, 63, 60, 57, 55, 52, 50, 47, 45, 43, 40,
         38, 36, 34, 32, 30, 28, 26, 24, 22, 21, 19, 17, 16, 15, 13, 12,
         11, 10,  8,  7,  6,  6,  5,  4,  3,  3,  2,  2,  2,  1,  1,  1,
          1,  1,  1,  1,  2,  2,  2,  3,  3,  4,  5,  6,  6,  7,  8, 10,
         11, 12, 13, 15, 16, 17, 19, 21, 22, 24, 26, 28, 30, 32, 34, 36,
         38, 40, 43, 45, 47, 50, 52, 55, 57, 60, 63, 65, 68, 71, 74, 77,
         79, 82, 85, 88, 91, 94, 97,100,103,106,109,112,116,119,122,125
        };
```

Another way to mimic the AFSK modulation is by offsetting a carrier wave (CW) based on values in a **LOOKUP table**. Inside the array of 256 bytes are values between 0 and 255 that constitute one wave length of a sine wave. In the background TimerCounter0 runs at 1200Hz setting a flag at the start of each baud. This implementation uses TimerCounter1 as well running it at 21739Hz. This timer sets a flag to force the next step inside the lookup table.



To switch between the tones all that needs to be done is to change the value of 'tableStep' variable from 14 (1200Hz) to 26 (2200Hz) or vice versa at the start of a baud. The progression of the algorithm throughout the table with two tone changes is illustrated in the image above.

```c
void SI4060_tx_APRS_look_up(void)
{
        SI4060_modulation(0, 1); // CW asynchronous modulation
        SI4060_frequency(APRS_tx_frequency - FREQ_OFFSET);

        SI4060_power_level(SI4060_TX_power);
        SI4060_change_state(0x07);

        TC0_init_APRS_lookup(); // TC0 at 1200Hz - Baud Rate
        TC1_init_APRS_lookup(); // TC1 at 21739Hz - Lookup table step

        ...

        uint8_t tableStep = LOOKUP_TBL_STEP_1200; // 14 (1200Hz), 26 (2200Hz)

        while(APRSpacketReady)
        {
                ...
        }
        ...
}
```

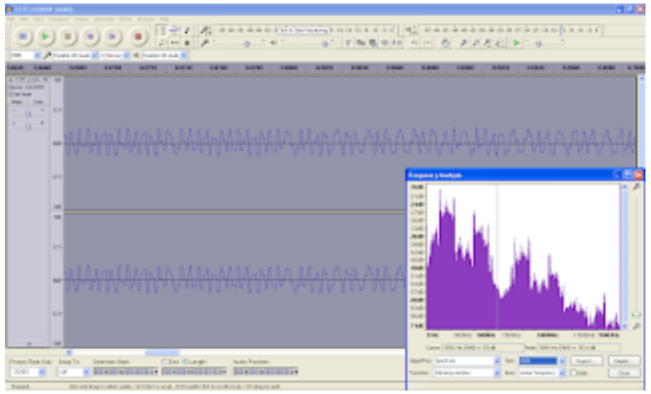
In software it looks like this. The preceding initializations are the same as with the GFSK version (SPI_init(), SI4060_init(), SI4060_setup_pins(), APRS_packet_construct()). Then the SI4060_tx_look_up() function is called.

Inside, the modulation is set to asynchronous CW (carrier wave). The tx frequency is offset by a value equal to half of the desired modulation width. The reason is that at every step through the lookup table I am adding a positive offset to the base frequency. For example to transmit the packet at 144.8MHz with 6kHz wide modulation I would set the base frequency to 144800000Hz minus an offset of 3000Hz. The packet would then be centered at 144.8MHz. Following that TC0 is enabled at 1200Hz to time individual bit processing and TC1 at 21739Hz which essentially calls SI4060_frequency_offset(SineLookUp[] * multiplier) and sets the new frequency offset i.e. the next step within the table.

$$MODEM\_FREQ\_OFFSET = \frac{2^{19} \times outdiv \times Desired\_Offset\_Hz}{N_{presc} \times freq\_xo}$$

$$MODEM\_FREQ\_OFFSET = \frac{2^{19} \times 24 \times Desired\_Offset\_Hz}{2 \times 32000000}$$

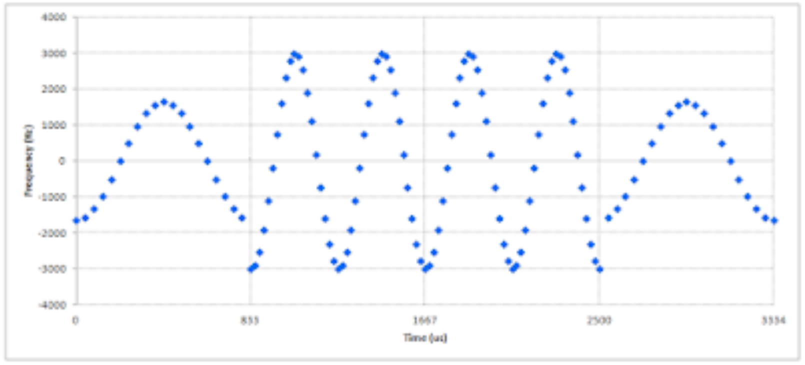
Since the values in the table range from 0 to 255 only and the function expects a value according to the above equation (Desired_Offset_Hz represents the peak deviation - deviation from the carrier, the deviation of the two peaks will be double the Desired_Offset_Hz), the 'multiplier' mediates this conversion and consequently defines the modulation width.



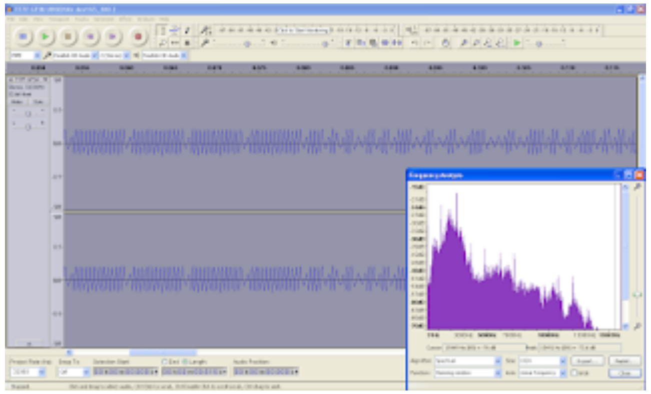
The audio recorded by SDR# for download: TT7F APRS LOOKUP (64MHz).wav.

0:00 / 0:00

This implementation is unfortunately limited by the duration it takes to the Si4060 to respond to the offset command, hence the 21739Hz frequency of TC1. That means a full wave length or two full wave lengths need to be 'drawn' in just 18 steps. Had the response been faster, I would have been able to create the sine wave in more detail. Regardless, the transmitted APRS packets are decodable.



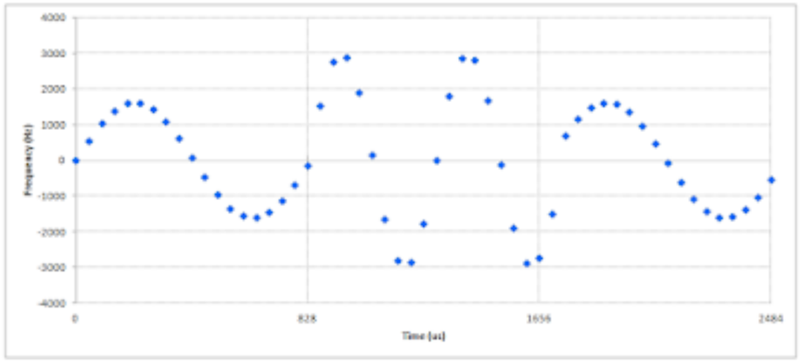
One aspect that relates to both implementations and I haven't addressed it properly yet is **PRE-EMPHASIS**. Apparently the FM transmitters and receivers generally used for APRS transmissions attenuate higher frequencies by 6dB per octave which translates to 5dB difference in amplitude for the 1200Hz and 2200Hz tones. Consequently the signal transmitted by TT7F should reflect this and adjust the deviations for each tone adequately. In the GFSK case (illustration above) it means that aside from changing the data rate I also change the deviation (SI4060_frequency_deviation()) for each tone. Googling around I arrived at the following values: 3.3kHz for the low tone (1.65kHz peak deviation) and 6kHz for the high tone (3kHz peak deviation).



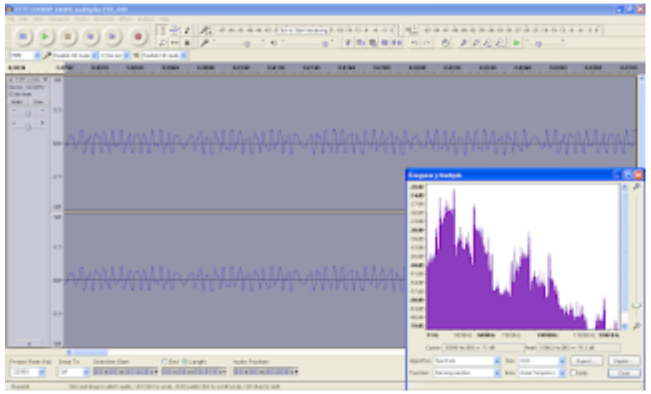
Audio sample for download: TT7F APRS GFSK (UBSEDS_fir) preemphasis.wav.

0:00 / 0:00

Both the signal as shown in Audacity and the frequency analysis show the tones having different amplitudes. Reading of the y axis the difference seems to be about 5dB as desired.



In the LOOKUP table implementation the previously mentioned 'multiplier' takes care of the pre-emphasis. Just as I change the 'tableStep' variable for the specific tone I change 'lookup_tbl_multiplier' as well. The values for the multiplier were precalculated as follows: 6000 (Hz desired modulation width) / 5.09 (Hz minimum offset) / 255 (maximum value in the table) = 4.6. In case of the lower tone: 3300 / 5.09 / 255 = 2.5. The lower tone, however, requires a fixed offset to be added to account for centering the two sine waves around a common axis. This offset is present in 'LookUpOffset' variable and was computed as follows: (6000 - 3300) / 2 / 5.09 = 265.



Audio sample for download: TT7F APRS LOOKUP (64MHz) preemphasis.wav.

0:00 / 0:00

Contrary to the GFSK version it is much harder to distinguish the amplitudes of the two tones in the signal itself, but the frequency analysis suggest the pre-emphasis was successfully implemented in this case as well.

Transmission
Now to address the transmission itself. The previous sections described the protocol and managed to construct an APRS packet that should now be inside a buffer. Following that another section outlined the shape of the modulation, initialized the transmitter and finished by entering a loop in which the packet is supposed to be processed bit by bit. To finally transmit the packet a few more things need to be mentioned.

0x03	0xF0	!	/	5	L	E	G
00000011	11110000	00100001	00101111	00110101	01001100	01000101	01000111
11000000	**00001111**	**10000100**	**11110100**	**10101100**	**00110010**	**10100010**	**11100010**

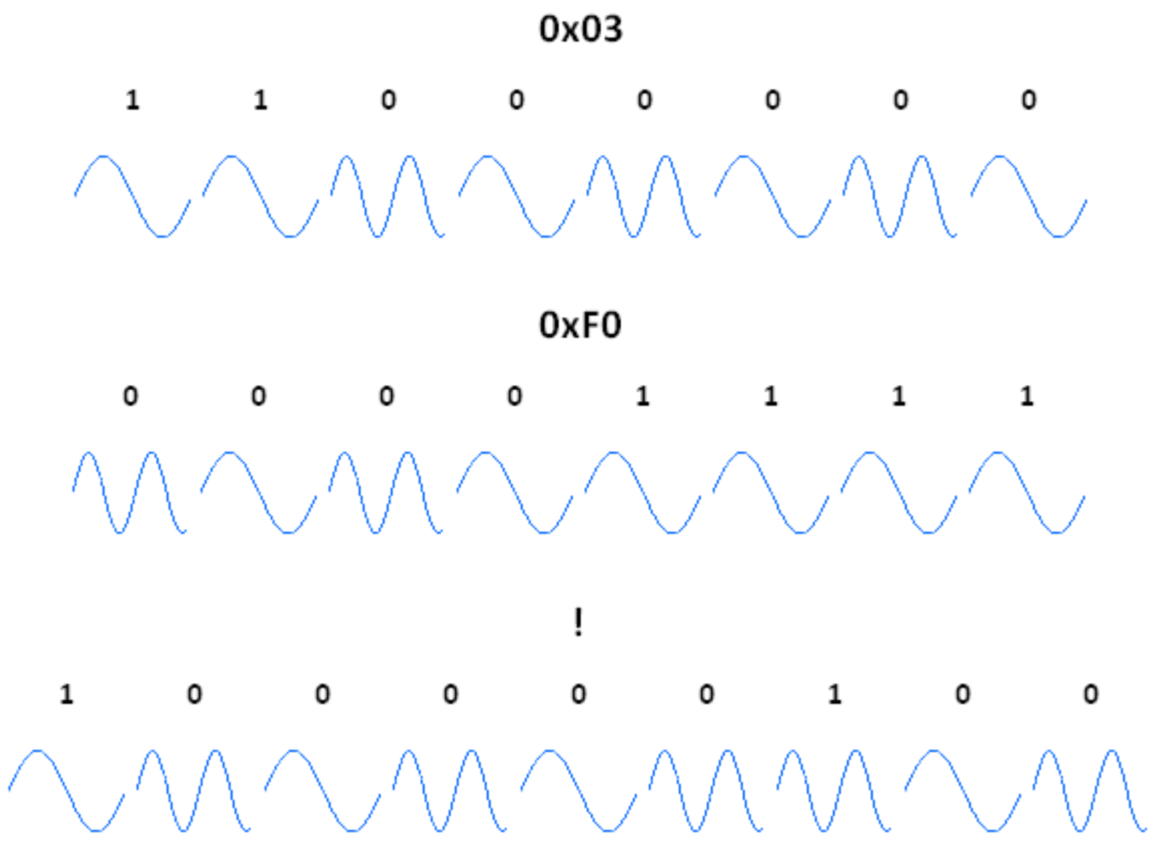
The individual bytes of the packet are transmitted **least significant bit first (LSB)**. The bottom row in the table above shows bits ordered as they are supposed to be transmitted.

```
if(dataByte & 0x01)         // BIT '1'
{
        ...

        dataByte >>= 1;     // right shift for the next bit
}
else                        // BIT '0'
{
        ...

        dataByte >>= 1;     // right shift for the next bit
}
```

In software it is arranged simply by putting the current byte inside a variable (dataByte) and right shifting the variable after each bitwise AND with a 1 when processing the data for the current bit to transmit.

0x03	0xF0	!	/	5	L	E	G
11000000	00001111	10000100	11110100	10101100	00110010	10100010	11100010
11000000	**00001111**	**100000100**	**11110100**	**10101100**	**00110010**	**10100010**	**11100010**

As mentioned when talking about the APRS 0x7E flags, the receivers use the bit pattern (six '1' in a row) to delimit the packet. However, at certain circumstances there may be a combination of bytes that results in six or more '1' bits in a row. This is dealt with by **stuffing in a '0' bit** each time there are five '1' bits in a row. The resulting bit pattern can be seen in the table above.

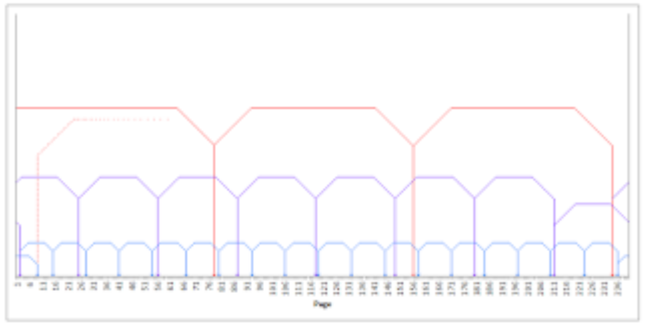


Last thing to employ is **Non-return-to-zero inverted (NRZI)** encoding. In this type of transmission, transmitting bit '1' means continuing with the current tone, while transmitting bit '0' means changing to the other tone that is being used. The illustration above should demonstrate the desired signal clearly. This is also another reason for the previously mentioned bit stuffing. Long series of '1' bits result in the signal remaining in one tone while transitions between the tones help the receiver to align with the signal.

All least significant bit ordering, bit stuffing and NRZI encoding is done inside a while loop that is entered in the SI4060_tx_APRS_look_up() or SI4060_tx_APRS_GFSK_sync() functions. The proper timing is provided by

TimerCounter0's and TimerCounter1's interrupt routines setting appropriate flags.

Backlog
ARM_EEFC.h
ARM_EEFC.c
The original goal for this APRS implementation was to provide automatic position reporting as the balloon floats around the Earth. The non-uniform distribution of receiving stations on the planet's surface, however, often leads to situations where the balloon stays out of reach for a number of days. The balloon's whereabouts during these periods are thus unknown unless a backlogging feature is implemented. The tracker periodically saves the current telemetry and re-transmits it later. To overcome the loss of stored data in the volatile memory when the tracker looses power, the historical information is stored in the MCU's flash memory. Writing to flash during runtime is done by using the Enhanced Embedded Flash Controller (EEFC). The functions are in the library above. This specific implementation expects SAM3S8B with its 256 bytes per each of 2048 pages in the 524,288 bytes of flash memory. In case of using SAM3S4 for example (only 256kB) minor adjustments to where within the flash the backlog will be located need to be done.



The backlog algorithm uses 241 pages of flash memory to store 240 historical strings of time, position and telemetry, and one pointer to the last position within this circular buffer. The tracker periodically (for example once per hour) encodes a string using APRS_encode_backlog() function and stores it on one page inside the flash memory via APRS_store_backlog(). This automatically updates a pointer to the last backlog and saves it inside the flash as well. Once the 240 slots are filled the algorithm wraps around and starts rewriting the oldest backlogs.

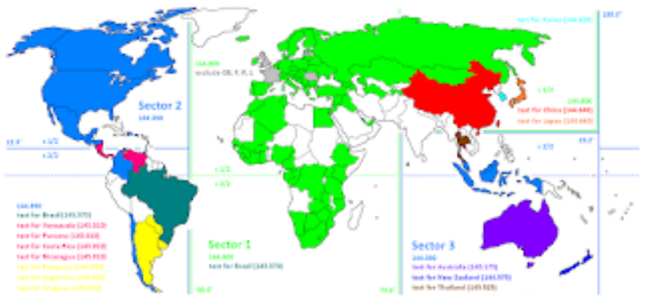
```
!/5MHwS(KVOHMW 0K/.45MI1S(KVHR!!$G+_0{!-|n0!!$?+[0z!+!$|
```

The backlogs are included in TT7F's APRS packet between the current position and current telemetry by using APRS_comment_backlog() function within APRS_packet_construct(). The algorithm that chooses the specific backlog is illustrated in the image above. It alternates between fewer longer jumps and a number of shorter jumps among the backlogs. This provides a quick overview of the balloon's past whereabouts and fills the gaps later provided the balloon stays within hearing distance. The 240 slots can hold 10 days worth of backlogging if a new log is created every hour.

Attribute	ASCII	Equation	Result
Year	0	'0' - 48 + 2016 =	2016
Month	K	'K' - 64 =	11
Day	/	'/' - 33 =	14
Hour	.	'.' - 33 =	13
Minute	4	'4' - 33 =	19
Latitude	5MI1	see Base91 decoding	49.4687
Longitude	S(KV	see Base91 decoding	18.1508
Altitude	HR	see Base91 decoding	403.7
Value 1	!!	('!' - 33) * 91 + ('!' - 33) =	0
Value 2	$G	('$' - 33) * 91 + ('G' - 33) =	311
Value 3	+_	('+' - 33) * 91 + ('_' - 33) =	972
Value 4	0{	('0' - 33) * 91 + ('{' - 33) =	1455
Value 5	!-	('!' - 33) * 91 + ('-' - 33) =	12

The structure of the backlog including its decoding is outlined in the table above. Upon reception the backlog is treated as a comment and has to be decoded manually.

Geofence
ARM_GEOFENCE.h
ARM_GEOFENCE.c
One inconvenience with APRS is that the transmit frequency isn't the same for the whole world. Hence the tracker needs to use its current GPS coordinates to decide which frequency to use for transmission each time.



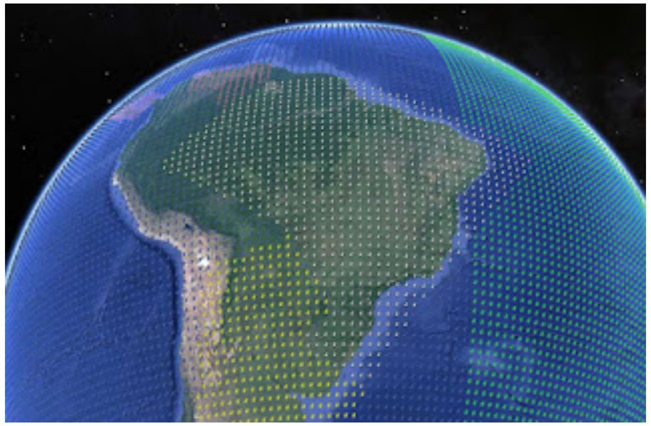
These are the frequencies I've managed to collect from around the Internet and other projects. The different sources agree on most of the frequencies, but there is a couple minor countries that I found referenced by only one source.

	MHz		**MHz**
Africa	144.800	Chile	144.390
Europe	144.800	Argentina	144.930
Russia	144.800	Paraguay	144.930
Canada	144.390	Uruguay	144.930
Mexico	144.390	China	144.640
USA	144.390	Japan	144.660
Costa Rica	145.010	South Korea	144.620
Nicaragua	145.010	Thailand	145.525
Panama	145.010	Australia	145.175
Venezuela	145.010	New Zealand	144.575
Brazil	145.570	Indonesia	144.390
Colombia	144.390	Malaysia	144.390

There is also a few places that do not allow airborne APRS. These countries are geofenced and the tracker doesn't transmit within the simplified outline of their borders.

no Airborne APRS
France
Latvia
Romania
United Kingdom

In software I first test the current GPS coordinates for a sector as outlined in the map above. Then I use the pointInPolygonF() function to test for a specific country present in the sector. The polygons outlining the countries in question are in the library above. They are simplified shapes to save memory. All the testing takes place inside GEOFENCE_position() function. The proper frequency for the upcoming transmission is then saved in 'GEOFENCE_APRS_frequency' global variable.



A quick test of the algorithm outputting different placemark colours for different frequencies. Visualized in Google Earth.

Example Code

To illustrate how to put it all together, here is an example of a main function that upon powerup waits for GPS fix and then indefinitely loops through updating GPS data, finding out which frequency to tx on, constructing an APRS packet, storing a backlog, transmitting the packet and transitioning into wait mode.

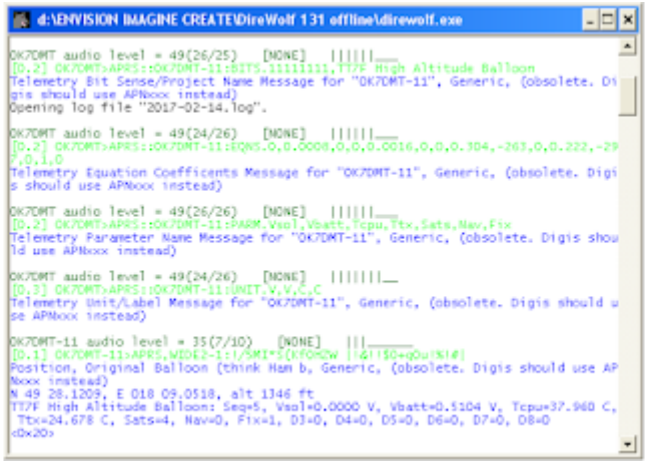
```
int main(void)
{
    SystemInit();
    WATCHDOG_disable();
    SysTick_delay_init();

    LED_PA0_init();
    LED_PB5_init();
    ADC_init();
    UART1_init();

    SysTick_delay_ms(1000);
    UBLOX_request_UBX(setNMEAoff, 28, 10, UBLOX_parse_ACK);
    UBLOX_request_UBX(setNAVmode, 44, 10, UBLOX_parse_ACK);

    GPSfix = 0;                                 // wait for GPS FIX
    while(GPSfix != 3)
    {
        UBLOX_request_UBX(request0107, 8, 100, UBLOX_parse_0107);
        LED_PB5_blink(5);
    }

    while (1)
    {
        SystemInit();

        telemCount++;

        ADC_start();
        AD3data = ADC_sample(3, 100);
        AD9data = ADC_sample(9, 100);
        AD15data = ADC_sample_temperature(100);
        ADC_stop();

        UBLOX_request_UBX(request0107, 8, 100, UBLOX_parse_0107);

        GEOFENCE_position(GPS_UBX_latitude_Float, GPS_UBX_longitude_Float);
        APRS_tx_frequency = GEOFENCE_APRS_frequency;

        APRShour                    = GPShour;
        APRSminute                  = GPSminute;
        APRSsecond                  = GPSsecond;
        APRSyear                    = GPSyear;
        APRSmonth                   = GPSmonth;
        APRSday                    = GPSday;
        APRSlatitude               = GPS_UBX_latitude_Float;
        APRSlongitude              = GPS_UBX_longitude_Float;
        APRSaltitude               = GPSaltitude;
        APRSsequence               = telemCount;
        APRSvalue1                 = AD3data;
        APRSvalue2                 = AD9data;
        APRSvalue3                 = AD15data;
        APRSvalue4                 = 0;
        APRSvalue5                 = GPSsats;
        APRSbitfield               = 0;
        if(GPSnavigation == 6)     APRSbitfield |= (1 << 0);
        if(GPSfix == 3)            APRSbitfield |= (1 << 1);

        TC_rtty_gfsk_lookup        = 1;        // GFSK_SYNC modulation
        APRS_send_path             = 2;        // path WIDE2-1
        APRS_show_alt_B91          = 1;        // enable B91 altitude

        APRS_packet_mode           = 1;        // choose APRS packet

        LED_PA0_blink(5);

        SPI_init();
        SI4060_init();

        APRSvalue4 = SI4060_get_temperature();
        APRS_packet_construct(APRSpacket);

        if(telemCount % 30 == 0) APRS_store_backlog();

        if(!GEOFENCE_no_tx)
        {
            SI4060_setup_pins(0x02, 0x04, 0x02, 0x02, 0x00, 0x00);
            SI4060_frequency_offset(0);

            SI4060_tx_APRS_GFSK_sync();
        }

        SI4060_deinit();
        SPI_deinit();

        RTT_init(30, 0x8000, 0);
```

```
87          PS_switch_MCK_to_FastRC(0, 0);
88          PS_enter_Wait_Mode(1, 0, 0);
89          RTT_off();
90      }
91 }
```

Worth noting is the delay (12) before the first GPS configuration messages. The module needs some time after power up to be able to receive commands. There is also a loop (17) at the beginning that waits for the GPS module to acquire valid fix. At the start of the main loop (25) the MCU initializes and enables PLL to provide 64MHz clock for MCK. This is necessary, because at the end of the loop the system goes to a low power Wait mode (87, 88). The microcontroller then requests UBX-NAV-PVT message (35) from the GPS and passes the positional data to the geofencing algorithm (37) that then provides the frequency to transmit on, or whether to transmit at all. After passing the fresh data to appropriate variables (40-57) the system chooses the desired modulation (59), GFSK this time, sets the packet's path (60) as WIDE2-1 and selects the specific packet to construct (63). One last data point needed is the temperature on the Si4060 transmitter which can be requested (70) only after the module is switched on (67, 68). After that everything is ready for constructing the packet itself (71), this time Base91 encoded position and telemetry with a string from the backlog. Every preset number of loops the tracker stores a new backlog (73). If the balloon is outside the no airborne APRS zones (75), it transmits the packet (80). In the end it sets a timer (86) and switches the MCU to Wait mode (87, 88).

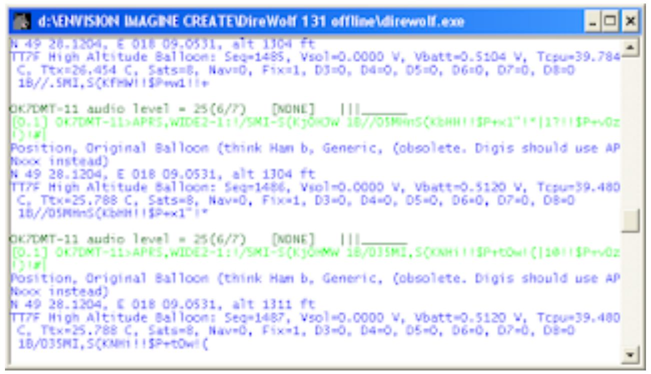
Reception

To receive APRS I have been using a 2-meter dipole antenna, an SDR dongle, SDR# to demodulate the signals to audio frequencies, Virtual Audio Cable to stream the data between software and Direwolf for the final decoding.


d:\ENVISION IMAGINE CREATE\DireWolf 131 offline\direwolf.exe

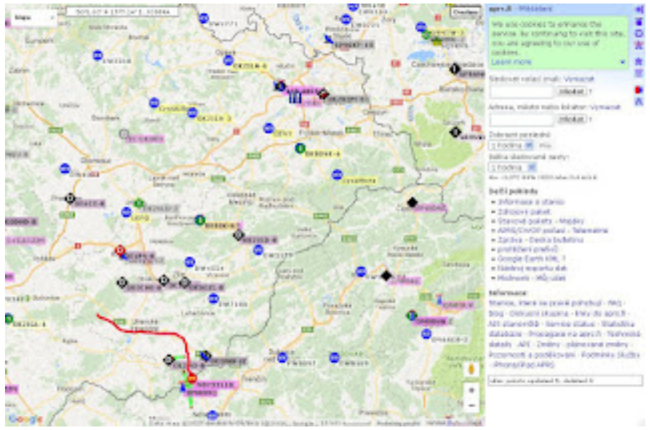
```
OK7DMT audio level = 49(26/25)   [NONE]   ||||||___
[0.2] OK7DMT>APRS::OK7DMT-11:BITS.11111111,TT7F High Altitude Balloon
Telemetry Bit Sense/Project Name Message for "OK7DMT-11", Generic, (obsolete. Di
gis should use APNxxx instead)
Opening log file "2017-02-14.log".

OK7DMT audio level = 49(24/26)   [NONE]   ||||||___
[0.2] OK7DMT>APRS::OK7DMT-11:EQNS.0,0.0008,0,0,0.0016,0,0,0.304,-263,0,0.222,-29
7,0,1,0
Telemetry Equation Coefficents Message for "OK7DMT-11", Generic, (obsolete. Digi
s should use APNxxx instead)

OK7DMT audio level = 49(26/26)   [NONE]   ||||||___
[0.2] OK7DMT>APRS::OK7DMT-11:PARM.Vsol,Vbatt,Tcpu,Ttx,Sats,Nav,Fix
Telemetry Parameter Name Message for "OK7DMT-11", Generic, (obsolete. Digis shou
ld use APNxxx instead)

OK7DMT audio level = 49(24/26)   [NONE]   |||||||__
[0.3] OK7DMT>APRS::OK7DMT-11:UNIT.V,V,C,C
Telemetry Unit/Label Message for "OK7DMT-11", Generic, (obsolete. Digis should u
se APNxxx instead)

OK7DMT-11 audio level = 35(7/10)   [NONE]   |||______
[0.1] OK7DMT-11>APRS,WIDE2-1:!/5MI*S(KfOHZW |!&!!$0+qOu!%!#|
Position, Original Balloon (think Ham b, Generic, (obsolete. Digis should use AP
Nxxx instead)
N 49 28.1209, E 018 09.0518, alt 1346 ft
TT7F High Altitude Balloon: Seq=5, Vsol=0.0000 V, Vbatt=0.5104 V, Tcpu=37.960 C,
 Ttx=24.678 C, Sats=4, Nav=0, Fix=1, D3=0, D4=0, D5=0, D6=0, D7=0, D8=0
<0x20>
```

A simple way to input the telemetry configuration packets into Direwolf or IGate them to the main servers via Direwolf is by having gen_packets.exe (distributed with Direwolf) create a wav file of these messages. The audio files can then be opened in Audacity and with the output set to the Virtual Audio Cable simply played. Direwolf will decode them as if they were coming from SDR#. The TT7F's configuration messages in a wav file for download here: TT7F telemetry settings.wav.

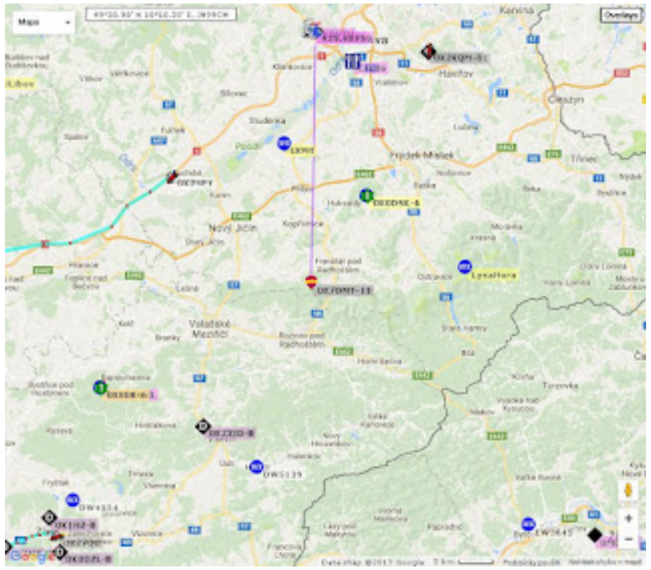


```
d:\ENVISION IMAGINE CREATE\DireWolf 131 offline\direwolf.exe
N 49 28.1204, E 018 09.0531, alt 1304 ft
TT7F High Altitude Balloon: Seq=1485, Vsol=0.0000 V, Vbatt=0.5104 V, Tcpu=39.784
 C, Ttx=26.454 C, Sats=8, Nav=0, Fix=1, D3=0, D4=0, D5=0, D6=0, D7=0, D8=0
 1B//.5MI,S(KfHW!!$P+w1!!+

OK7DMT-11 audio level = 25(6/7)    [NONE]   |||______
[0.1] OK7DMT-11>APRS,WIDE2-1:!/5MI-S(KjOHOW 1B//05MHnS(KbHH!!$P+x1"!*|17!!$P+vOz
!)!#|
Position, Original Balloon (think Ham b, Generic, (obsolete. Digis should use AP
Nxxx instead)
N 49 28.1204, E 018 09.0531, alt 1304 ft
TT7F High Altitude Balloon: Seq=1486, Vsol=0.0000 V, Vbatt=0.5120 V, Tcpu=39.480
 C, Ttx=25.788 C, Sats=8, Nav=0, Fix=1, D3=0, D4=0, D5=0, D6=0, D7=0, D8=0
 1B//05MHnS(KbHH!!$P+x1"!*

OK7DMT-11 audio level = 25(6/7)    [NONE]   |||______
[0.1] OK7DMT-11>APRS,WIDE2-1:!/5MI-S(KjOHMW 1B/035MI,S(KNHi!!$P+tOw!(|10!!$P+vOz
!)!#|
Position, Original Balloon (think Ham b, Generic, (obsolete. Digis should use AP
Nxxx instead)
N 49 28.1204, E 018 09.0531, alt 1311 ft
TT7F High Altitude Balloon: Seq=1487, Vsol=0.0000 V, Vbatt=0.5120 V, Tcpu=39.480
 C, Ttx=25.788 C, Sats=8, Nav=0, Fix=1, D3=0, D4=0, D5=0, D6=0, D7=0, D8=0
 1B/035MI,S(KNHi!!$P+tOw!(
```

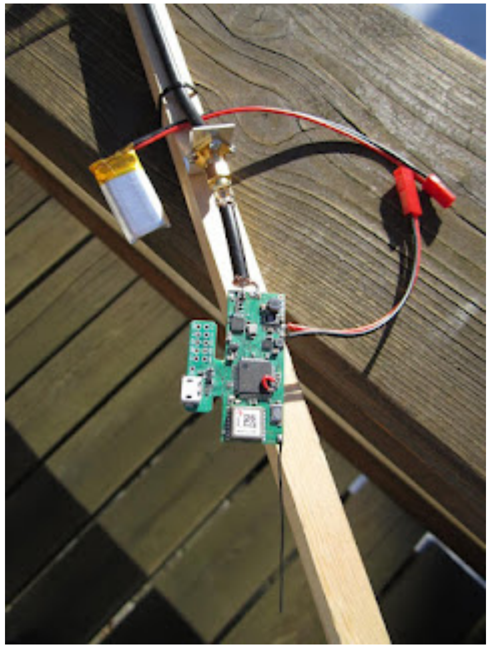
Here is an example of a packet with backlog from the above example code. The only difference was a shorter low power period so I didn't have to wait 10 days to fill the backlog. With 5 seconds between packets first backlogs started to appear after about 15 minutes. Naturally, this high packet frequency was only for off air testing.



For visualizing the APRS traffic I usually use APRS.fi. To IGate one's own packets to APRS servers I found it to be easiest with Direwolf again. Simply uncomment a couple of lines in direwolf.conf, which is located in the same directory as the exe file, and fill a few details. All is sufficiently described inside the conf file.



My standard location, unfortunately, doesn't have a line of sight to any digipeater nor IGate. That makes testing somewhat difficult, because every time I change something I have to drive to a suitable location. When working on the transmitter, I naturally tested it by receiving and IGating it myself, but in the end I needed to verify it on real world receivers. Although I had initially problems to get through to an IGate, after all Si4060 transmits at 10mW at best, it eventually worked out as evidenced by the screen above. The IGate was 37km away. Airborne, the coverage should be even better.



A couple of photos of the outdoor test setup for a better idea. Generally held in a hand at arms length to minimize the detuning. I also tested it with a wire dipole antenna. It worked as well, but it didn't do as good as the solid dipole.

Posted by TT7 at 02:10

5 comments:

Unknown 6 August 2018 at 06:59

I was just having a look at the audio waveforms and a 0x7E is the flag char but it appears to be 0x81 in the waveforms. I thought the bell 202 was a '1' 1200Hz, and '0' 2200Hz. I may be wrong here, I was just interested in doing something similar in Matlab.

Reply

TT7 6 August 2018 at 13:29

I don't recall reading Bell 202 specification, so can't comment on that. But for APRS, bits '1' and '0' are not fixed to either tone. Bit '1' means the tone stays the same, whichever it currently is, for the duration of the bit, while bit '0' means changing the tone either 1200Hz to 2200Hz or vice versa.

Reply

Richard 10 June 2019 at 12:06

Hi Tomas,

I have been looking at your APRS_packet_construct function (line 865 at https://github.com/TomasTT7/TT7F-Float-Tracker/blob/master/Software/ARM_APRS.c) and seeing what characters it outputs into the transmission buffer, but I can’t work out where I am going wrong in terms of not being able to decipher the example you give at line 28 of

https://github.com/TomasTT7/TT7F-Float-Tracker/blob/master/Software/ARM_TT7F5.c

namely,

~~~~~~~~~~~~, ¤¦@@àž–n<0x88>š¨v®’<0x88>Šd@c<0x03><0xF0>!/5MGoS(L_0GzW |!$!!#p+30n!!!"|~~~

In the first loop in the APRS_packet_construct function, line 870 in

https://github.com/TomasTT7/TT7F-Float-Tracker/blob/master/Software/ARM_TT7F5.c

```
// Flags
for(uint32_t i = 0 ; i < APRSFLAGS; i++)
{
buffer[num++] = 0x7E;
}
```

I can see the leading 12 0x7E bytes manifesting themselves as ~ or equivalently character 126 in the ASCII table) in the example string above

In the second loop, line

// Destination Address
~~~~~~~~~~~~

```
for(uint8_t i = 0; i < 6; i++)
{
buffer[num++] = APRS_destination[i];
}
```

Based on my reading of the protocol, I would expect to see 'APRS ' as the destination field coming after the string of ~ in the example, but instead the example above says those six characters are ', ¤¦@@', yet neither the characters nor the ASCII numbers for them seem to correspond to anything. As the @@ are both the same, it seems that this indeed corresponds to 'APRS ', but I am clearly not reading it correctly!

I didn't know whether you could tell me where I am going wrong with my reading of it please?

Many thanks

Richard

Reply

Replies

TT7 10 June 2019 at 21:23

Hi Richard,

all the address bytes (destination + ssid, source + ssid, path) are left shifted by 1 bit. Hence the weird characters. If you right shift them by one bit, you should get 'APRS' and so on back. It is done by the for loop at line 959. Also, the bit[0] of the last byte of these address/path fields is changed to '1' after the byte was left shifted to signal the end of that section. All of this is part of the protocol.

Tomas

Richard 11 June 2019 at 09:50

Many thanks Tomas - I right shifted and got back the text as I expected to see it as you say. You deserve a Nobel prize for figuring out the protocol....

Reply