The way the APRS Script Works:

APRS Pseudocode:

1. Collect all Data
   1. DEVID: “APRS”
   2. Call Sign “K5TRL”
   3. SSID “11”
   4. WIDE Path Configuration (cond: on altitude)
   5. Latitude (read from GPS script-serial)
   6. Longitude (read from GPS)
   7. Altitude (read from NMEA GPS)
   8. # of GPS satellites (read from GPS)
   9. Sensor Telemetry
      1. Accelerometer (Analog)
      2. Temperature (I2C - “)
      3. Humidity and Temperature (I2C)
      4. Pressure, Altitude, and Temperature (I2C - “Slave - 0x60” “8-bit Read - 0xC1” “8-bit Write 0xC0”)
2. Convert Data to AX.25
   1. Flag
      1. 7E (01111110) - - send at least once before message
   2. Address
      1. Calculate SSID value for
         1. Callsign
            1. K5TRL (SSID = 11)
         2. Destination
            1. APRS or GPS (SSID = 0)
         3. Via
            1. WIDE1-1 or WIDE2-1 (SSID = 1 for both)
      2. SSID should end in 0 if there are more call signs, and in 1 if there are no more call signs
   3. Control
      1. 3F
   4. Protocol ID
      1. FO
   5. Information
      1. Latitude
      2. Longitude
      3. Altitude
      4. No. of Satellites
      5. Sensor Data
   6. Frame Check Sequence
      1. Dunno, hard:
      2. <https://barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code>
      3. <http://practicingelectronics.com/articles/article-100003/CRC_CCITT_Generator.m>
      4. <https://en.wikipedia.org/wiki/Computation_of_cyclic_redundancy_checks>
      5. <https://github.com/tcort/va2epr-tnc/blob/master/firmware/aprs.c>
   7. Flag
      1. 7E -- send at least once
3. Form Sentence

***EXAMPLE of how the packet will be read by APRS-IS:***

K5TRL-11>APRS,WIDE2-1: packet data

***EXAMPLE of the data to send bit by bit:***

{7E, 7E, 7E, ‘A’, ‘P’, ‘R’, ‘S’, ‘ ’, ‘ ’, 011 0000 0, ‘K’, ‘5’, ‘T’, ‘R’, ‘L’, ‘ ’, 011 1011 0, ‘W’, ‘I’, ‘D’, ‘E’, ‘2’, ‘ ’, 011 0001 1, 3F, F0, ‘@’, ‘4’, ‘5’, ‘.’, ‘5’, ‘1’, ‘1’, ‘2’, ‘N’, ‘-’, ‘1’, ‘2’, ‘2’, ‘.’, ‘3’, ‘1’, ‘4’, ‘8’, … etc… , fcsHI, fcsLO, 7E, 7E, 7E}

1. Modulate Data to AFSK1200
   1. Flipping from 1200 to 2200 is a zero, staying the same is a one. (NRZI)
   2. Generate Sine Wave Data table for PWM comparator values
   3. Turn on FastPWM
2. Transmit
   1. Check Transmission Time
   2. Transmit (Interrupt Sequence if Time)
3. Collect all data
4. Calculate SSID values
5. Put data into sentence array with address and SSIDs and data to send
6. Process array by bit, calculating the FCS for each byte (but not if they are flag or fcs bytes)
   1. Do the bit stuffing procedure if there are more than 5 1s in a row

**SetupAPRS()**

Sets up the enable pin on the transceiver if it has one and is defined.

Sets the Timer 2 pins (pin 3 or 11) to fast PWM mode, and sets the defined data pin to OUTPUT. From what I can tell, Fast PWM mode increases the frequency of PWM, and it does a comparison of some type which gets stored in a bit named TIMER2\_OVF\_vect (which is defined in the arduino hardware). When that bit gets changed, it triggers the ISR (Interrupt Service Routine) which begins the process of sending out any data that is stored in the buffer \_txbuf.

**CheckAPRS()**

This script is constantly running in a loop (main loop) and is checking to see if it is time to make a transmission.

When it is time, it runs the tx\_aprs() function.

This script also tells when it is sending the APRS packet, and how much time it will be until the next packet.

**tx\_aprs()**

This script converts data from the GPS script (GPS.Latitude, GPS.Longitude, and GPS.Altitude) into the APRS compressed format.

If the altitude is above the defined APRS path switching setting, it switches to WIDE2 (more jumps).

It also uses the ax25\_base91enc() function to compress telemetry data into a base-91 value (data from sensors) into a string array it references called stlm. (string telemetry) stlm will be passed to the ax25\_frame() function which will compress everything into the correct format for tranmission.

If telemetry intervals are set (if we are sending telemetry we should uncomment out APRS\_TELEM\_INTERVAL) (basically, if we want it to send data gradually rather than all at once we would set the interval to a certain value), it changes the aprs\_mode each time it sends data and sends data a little bit at a time.

Normal mode: aprs\_mode == 0; All other modes involve gradually sending data.

All transmissions are prepended using the ax25\_frame() function which takes in data on the call sign, ssid, path, as well as latitude, longitude, altitude, telemetry data, and the comment. It puts this all into the correct format and handles sending as well I think.

**\_a25\_callsign(\*s, \*callsign, ssid)**

This script uses a for loop to copy the values of the callsign character array to the array pointed to by s. Each time it change the address of s by 1, and returns the final address of s once it’s iterated through all of the call sign followed by a space, followed by the ssid. Some stuff is happening with that << operator, which I still don’t understand.

**ax25\_base91enc()**

This script converts some value into a base-91 representation as a string. Uses for loop to change the value interatively into a base 91 number string. (using modulo 91, and successive divisions by 91 counting down from the null)

**ax25\_frame()**

This may be the crux of the APRS script here and may or may not handle the actual transmission. For sure, it processes the call sign, SSID, the WIDE paths, and other data for the final time before it is transmitted.

The callsign and ID data gets put into an array with a pointer s using the \_ax25\_callsign() function, which iteratively changes the address of s (using s++) and returns the last value of the s address. It then appends the path data (either wide1 and wide 2 or just wide2 [ttl1 or ttl2], )

It also appends a checksum value.

It also points the ***interrupt[[1]](#footnote-0)*** at the data to be transmit starting at the beginning of the “frame” array which contains the total data. The interrupt is dealt with in the ISR() function which is a special kind of function called “Interrupt Service Routine” that runs code when it senses an interrupt from the TIMER2 pins (11 and 3 -- where APRS\_DATA is connected) and sends the data pointed to by \_txbuf (which points to frame) for the length \_txlen (non-pointer).

After which it does something with the timer2 interrupt (TIMSK2 |= \_BV(TOIE2); (x |= y → x = x | y) which apparently enables the timer and keys? the radio (my guess is that starts the transmission).

This allows the ISR() function to start sending data bit by bit once it is triggered.

NEW: \_BV(TOIE2) -- TOIE2 is a bit that when set to 1 along with some other bit, it will ENABLE the overflow interrupt sequence[[2]](#footnote-1). So this is needed for the ISR() function to even run. At the end of the ISR function it uses the code ~(\_BV(TOIE2) which disables the interrupt process until it is enabled again by the ax25\_frame() function.

**ISR(TIMER2\_OVF\_vect)**

This is a special type of function called an interrupt service routine function that triggers at particular points that have something to do with the Timer2 PWM. The timer is started by the ax25\_frame function and thus interrupts probably start happening beginning with that function. My best understanding of what it does is that it uses the sine\_table array[[3]](#footnote-2) to create special tones or frequencies which are transmitted through the radio’s voice/data pin. It looks like it takes global input from \_txbuf which was be previous set to the address of the start of the processed data to be transmitted, and then runs for the length of \_txlen at which point it sets rest to true, and then disables the radio and interrupt sequence.

**General Pseudocode:**

|  |
| --- |
| CHECK FOR APRS TRANSMIT TIME INTERVAL AND OTHER CONDITIONS // Conditions: millis() is greater than the NextAPRS time value, the number of GPS Satellites connected to is 4 or above, and there is no data in the buffer  IF TRUE CONVERT DATA FROM GPS INTO APRS COMPRESSED FORMAT SET WIDEPATH VALUES CONVERT TELEMETRY SENSOR DATA INTO BASE 91  CONSTRUCT THE DATA TO SEND INTO A "SENTENCE" WITH ARRAY ADDRESS \*S  START SENTENCE APPEND CALL SIGN APPEND WIDEPATH VALUES APPEND CONTROL AND PROTOCOL IDS (0X03 AND 0XF0) // These are some protocol codes which are probably part of some regulatory code or something APPEND LATITUDE, LONGITUDE, ALTITUDE, TELEMETRY, AND COMMENT DATA END SENTENCE  APPEND CHECKSUM VALUE  POINT \_txbuf TO START OF DATA DEFINE \_txlen TO TOTAL LENGTH OF SENTENCE  ENABLE TIMER (AND INTERRUPT SEQUENCE) ENABLE TRANSCEIVER  (INTERRUPT SEQUENCE) READ SENTENCE BIT BY BIT AND PROCESS APPROPRIATELY FOR TRANSMISSION // Probably reads one bit, transmits, toggles the transceiver, and then does it again for the entirety of \_txlen. This is like a FOR loop but is probably triggered by the PWM on the timer and not programmatically. DISABLE TRANSCEIVER AND TIMER2 INTERRUPT SEQUENCE  CALCULATE THE NEXT APRS TIME INTERVAL PLUS OR MINUS A RANDOM TIME INTERVAL |

**More Information:**

[Interrupt Service Routines Article](https://www.allaboutcircuits.com/technical-articles/using-interrupts-on-arduino/)

[Fast Pulse Width Modulation](https://www.arduino.cc/en/Tutorial/SecretsOfArduinoPWM)

[More Fast PWM](http://www.eprojectszone.com/how-to-modify-the-pwm-frequency-on-the-arduino-part1/)

1. <http://www.instructables.com/id/Arduino-Timer-Interrupts/> [↑](#footnote-ref-0)
2. <http://web.ics.purdue.edu/~jricha14/Timer_Stuff/TIMSK.htm> [↑](#footnote-ref-1)
3. <https://www.avrfreaks.net/sites/default/files/forum_attachments/afsk_avr.cpp> [↑](#footnote-ref-2)