The serial radio looks almost identical to the normal Serial communications.

By default the radios are configured to talk to each other. To change the settings we can use the AT commands.

AT Command Code:

```ino
void setup()
{
  Serial.begin(9600);
  Serial1.begin(57600); // connected to RX (pin0), TX (pin1) on arduino giga
}

void loop()
{
  // If we are trying to talk to the raido
  while (Serial.available() > 0)
  {
    char data = Serial.read(); // Read what we want to send
    Serial1.print(data);       // Send it to the radio
  }

  // If the radio wants to talk to us
  while (Serial1.available() > 0)
  {
    char data = Serial1.read(); // Read what it is saying
    Serial.print(data);         // Send it to our monitor
  }
}
```

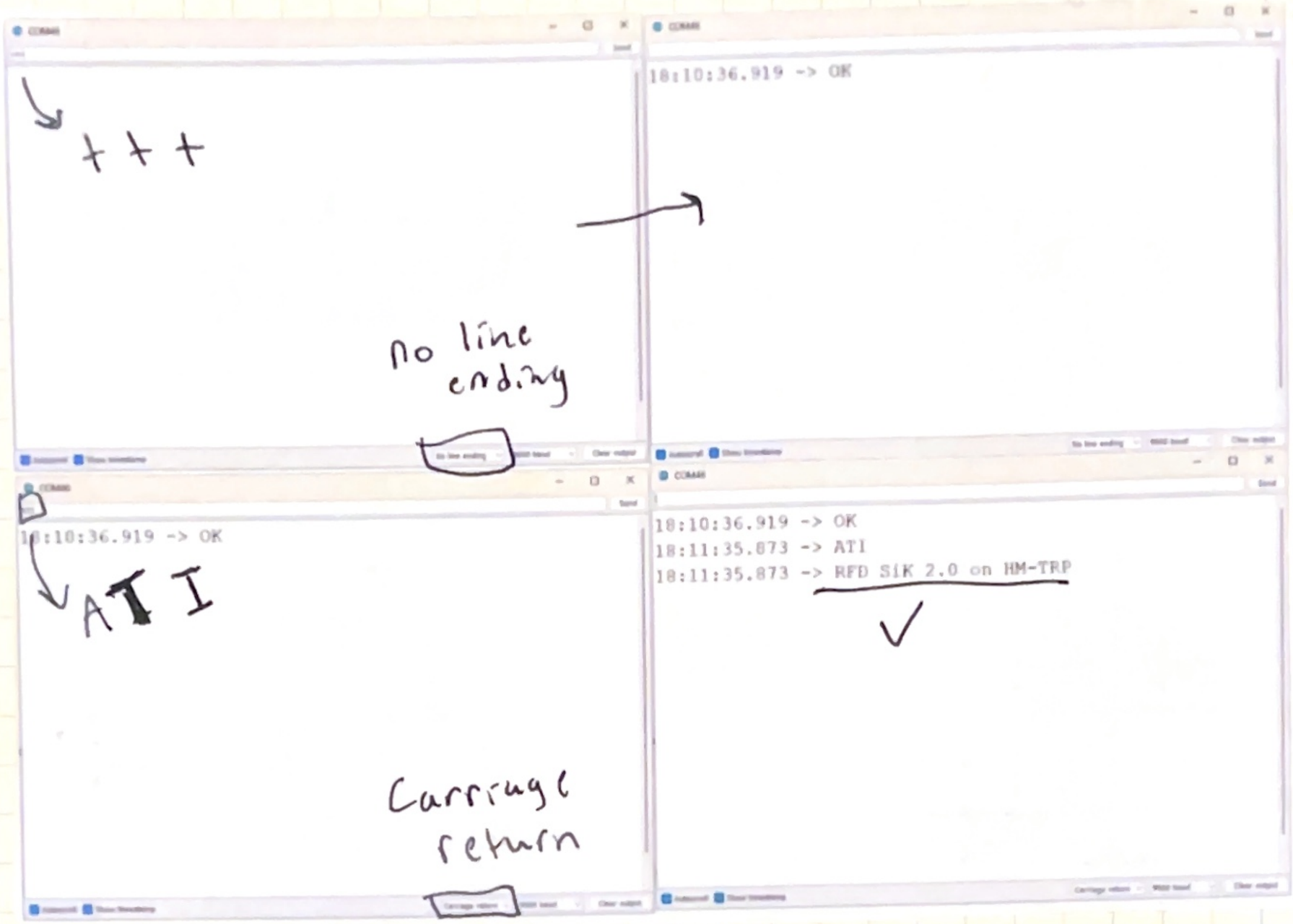
default

The deEault baud rate is 57,600.

To enter command mode we send '+++' with no line ending. From there we can send AT commands with the carriage return selected.

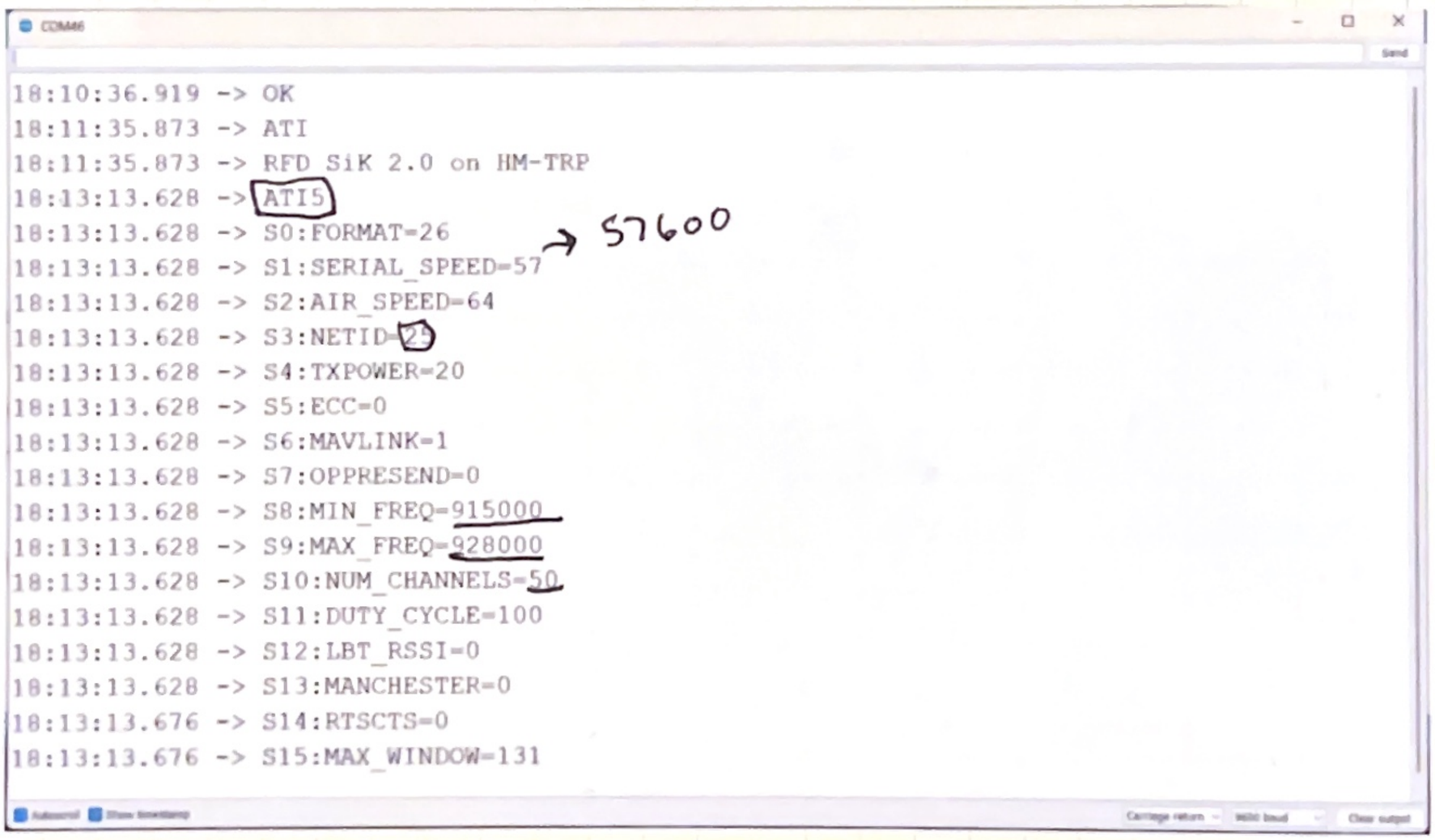
Serial Radio Basics

07/01/2025
Wesley Cadd



Example of entering AT command mode

Use ATIS to see the settings:



More on AT commands:

Once in AT command mode, you can give the radio either 'AT' commands to control the local radio, or (if successfully connected) you can use 'RT' commands to control the remote radio.

default 57,600

The AT commands available are:

- ATI - show radio version
- ATI2 - show board type
- ATI3 - show board frequency
- ATI4 - show board version
- ATI5 - show all user settable EEPROM parameters
- ATI6 - display TDM timing report
- ATI7 - display RSSI signal report
- ATO - exit AT command mode
- ATSn? - display radio parameter number 'n'
- ATSn=X - set radio parameter number 'n' to 'X'
- ATZ - reboot the radio
- AT&W - write current parameters to EEPROM
- AT&F - reset all parameters to factory default
- AT&T=RSSI - enable RSSI debug reporting
- AT&T=TDM - enable TDM debug reporting
- AT&T - disable debug reporting

all of these commands, except for ATO, may be used on a connected remote radio by replacing 'AT' with 'RT'.

Perhaps the most useful command is 'ATI5' which displays all user settable EEPROM parameters. That will produce a report like this:

More on AT commands

For the out put of ATIS:

The first column is the S register to set if you want to change that parameter. So for example, to set the transmit power to 10dBm, use 'ATS4=10'.

Most parameters only take effect on the next reboot. So the usual pattern is to set the parameters you want, then use 'AT&W' to write the parameters to EEPROM, then reboot using 'ATZ'. The exception is the transmit power, which changes immediately (although it will revert to the old setting on reboot unless you use AT&W).

The meaning of the parameter is as follows:

- `FORMAT` - this is for EEPROM format version. Don't change it
- `SERIAL_SPEED` - this is the serial speed in 'one byte form' (see below)
- `AIR_SPEED` - this is the air data rate in 'one byte form'
- `NETID` - this is the network ID. It must be the same for both your radios
- `TXPOWER` - this is the transmit power in dBm. The maximum is 20dBm
- `ECC` - this enables/disables the golay error correcting code
- `MAVLINK` - this controls MAVLink framing and reporting. 0=no MAVLink framing, 1=frame mavlink, 2=low latency mavlink
- `MIN_FREQ` - minimum frequency in kHz
- `MAX_FREQ` - maximum frequency in kHz
- `NUM_CHANNELS` - number of frequency hopping channels
- `DUTY_CYCLE` - the percentage of time to allow transmit
- `LBT_RSSI` - Listen Before Talk threshold (see docs below)
- `MAX_WINDOW` - max transmit window in msecs, 131 is the default, 33 recommended for low latency (but lower bandwidth)

For two radios to communicate the following must be the same at both ends of the link:

- the radio firmware version
- the AIR_SPEED
- the MIN_FREQ
- the MAX_FREQ
- the NUM_CHANNELS
- the NETID
- the ECC setting

- the LBT_RSSI setting
- the MAX_WINDOW setting

the other settings may be different at either end of the link, although you will usually set them up the same at both ends.

Sample code:

Sending side

```ino
float pi    = 3.14;
int   myInt = 314;

void setup()
{
  Serial.begin(9600);
  Serial1.begin(57600);
}

void loop()
{
  // Send a header byte
  Serial1.print('R');
  // Send our data with a return
  Serial1.print(pi);    Serial1.print('\r');
  Serial1.print(myInt); Serial1.print('\r');
  Serial1.print('E'); // footer
  delay(10000);
}
```

Receiver side

```ino
void setup()
{
  Serial.begin(9600);
  Serial1.begin(57600);
}

void loop()
{
  // If the radio has something to say
  while(Serial1.available() >= 11)
  {
    // Check the header
    char header = Serial1.read();
    if (header == 'R') // look for header
    {
      // Parse our data
      float pi  = Serial1.parseFloat(); Serial1.read(); // for the \r
      int myNum = Serial1.parseInt();   Serial1.read();

      // Check that our footer is 'E'
      if ((char) Serial1.read() == 'E')
      {
        Serial.print("Pi is:    "); Serial.println(pi);
        Serial.print("myNum is: "); Serial.println(myNum);
      }
    }
  }
}
```

do something with the data.

```ino
typedef struct {    // Define our structure
  float pi;         // A structure is like an array, but of different types
  int myInt;
  uint8_t setting;
  long HaugerLong;
} SensorData;
SensorData SendData; // Create an instance of "SensorData" type
                     // that will contain the data we want to send
void setup()
{
  Serial.begin(9600);
  Serial1.begin(57600); // Start the raido

  SendData.pi       = 3.14;          // Fill the structure.
  SendData.myInt    = 314;           // This would normally happen in your loop
  SendData.setting = 0x52;           // But we are faking the data
  SendData.HaugerLong = 321000000;
}

void loop()
{
  // Treat the start of the structure as a byte array
  // Tell arduino that the memory address (&SendData) of the structure
  // is a pointer to the start of a byte array (byte *)
  byte * b = (byte *) &SendData;
  Serial1.print('R');                    // Header
  Serial1.write(b, sizeof(SendData)); // All our variables
  Serial1.print('E');                    // Footer
  delay(5000);
}
```

```ino
typedef struct {          // Define our structure
  float pi;               // This needs to match the sender
  int myInt;
  uint8_t setting;
  long HaugerLong;
} SensorData;
SensorData RecData;           // Where we plan on storing the data
byte rec[sizeof(SensorData)]; // An array to keep up with the bytes
                              // from the serial monitor
void setup()
{
  Serial.begin(9600);
  Serial1.begin(57600);
}

void loop()
{
  while(Serial1.available() >= sizeof(SensorData)+2) // Size of the struct
  {                                                  // and header/footer
    char header = Serial1.read();
    if (header == 'R')
    {
      for(int ii=0; ii<sizeof(RecData); ii++)        // Read the struct data
      {                                              // into our rec array
        rec[ii] = Serial1.read();
      }
      char footer = Serial1.read();                  // Check the footer
      if (footer == 'E')
      {
        // memcpy(dest, source, length);
        // Tells arduino to take the data from the rec array
        // and put it at the place defined by the memory address of RecData.
        memcpy(&RecData, rec, sizeof(RecData));

        Serial.print("Pi is     : "); Serial.println(RecData.pi);
        Serial.print("myInt is  : "); Serial.println(RecData.myInt);
        Serial.print("Setting is: "); Serial.println(RecData.setting);
        Serial.print("Long is   : "); Serial.println(RecData.HaugerLong);
      }
    }
  }

}
```

When sending a lot of data, using the ASCII text leaves variability in the number of bytes transmitted

For example:

"3.14" and "100.256" are

both floats and have 4 bytes in memory. But the # of ASCII bytes are 4 and 7 respectively.

This makes it hard to know when to start receiving into memory.

If we use a struct and serial.write() instead, we know for certain how many bytes we are looking for.