M17 Protocol Specification

M17 Working Group

DRAFT

Authors: Mark KR6ZY
Jeroen PE1RXQ
Wojciech SP5WWP
Steve KC1AWV
Nikoloz SO3ALG

Formatted by Steve Miller, KC1AWV

Document History

Date	Comments
19 Dec 2019	Initial formatting - KC1AWV
20 Dec 2019	Author credits - KC1AWV
28 Dec 2019	Defined Physical layer parameters, some clean up - KR6ZY
30 Dec 2019	Added modulation description - SP5WWP
15 Jan 2020	Added scrambling description – SO3ALG, SP5WWP

Table of Contents

I. M17 RF Protocol: Summary

M17 is an RF protocol that is:

- Completely open: open specification, open source code, open source hardware, open algorithms. Anyone must be able to build an M17 radio and interoperate with other M17 radios without having to pay anyone else for the right to do so.
- Optimized for amateur radio use.
- Simple to understand and implement.
- Capable of doing the things hams expect their digital protocols to do:
    - Voice (eg: DMR, D-Star, etc)
    - Point to point data (eg: Packet, D-Star, etc)
    - Broadcast telemetry (eg: APRS, etc)
- Extensible, so more capabilities can be added over time.

To do this, the M17 protocol is broken down into three protocol layers, like a network:

1. Physical Layer: How to encode 1s and 0s into RF. Specifies RF modulation, symbol rates, bits per symbol, etc.
2. Data Link Layer: How to packetize those 1s and 0s into usable data. Packet vs Stream modes, headers, addressing, etc.
3. Application Layer: Accomplishing activities. Voice and data streams, control packets, beacons, etc.

This document attempts to document these layers.

II. Physical Layer

1 4FSK generation

M17 standard uses 4FSK modulation running at 4800 symbols/s (9600bits/s) with a deviation index *h*=0.33 for transmission in 6.25kHz channel bandwidth. Channel spacing is 12.5kHz. The symbol stream is converted to a series of impulses which pass through a root-raised-cosine (α=0.5) shaping filter before frequency modulation at the transmitter and again after frequency demodulation at the receiver.

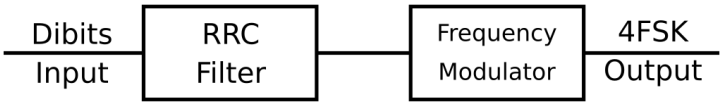

Figure 1: 4FSK modulator

The bit-to-symbol mapping is shown in ***Table 1.***

Table 1: Dibit symbol mapping to 4FSK deviation

Information bits		Symbol	4FSK deviation
Bit 1	Bit 0		
0	1	+3	+2.4 kHz
0	0	+1	+0.8 kHz
1	0	-1	-0.8 kHz
1	1	-3	-2.4 kHz

2 Preamble

In all uses of “Preamble” in higher level protocols, the Preamble is a repeating pattern of +3 to -3 symbols, or 0x77 in bits.

III. Data Link Layer

The Data Link layer is split into two modes:

1. Packet Mode: Data are sent in small bursts, on the order of 100s to 1000s of bytes at a time, after which the Physical layer stops sending data. eg: Start Stream messages, beacons, etc.
2. Stream Mode: Data are sent in a continuous stream for an indefinite amount of time, with no break in Physical layer output, until the stream ends. eg: Voice data, bulk data transfers, etc.

When the Physical Layer is idle (no RF being transmitted or received), the Data Link defaults to Packet mode. To switch to Stream mode, a Start Stream packet (detailed later) is sent, immediately followed by the switch to Stream mode; the Stream of data immediately follows the Start Stream packet without disabling the Physical layer. To switch out of Stream mode, the stream simply ends and returns the Physical layer to the idle state, and the Data Link defaults back to Packet mode.

1 Packet Mode

In Packet Mode, a finite amount of payload data (eg: Start Stream messages, or Application Layer data) is wrapped with a packet, sent over the Physical Layer, and is completed when done. Any acknowledgement or error correction is done at the application layer.

1.1 Packet Format

- Preamble: 8 bytes (value defined in Physical Link section.)
- Sync: 2 bytes, 0x3243 (Pi in Hexidecimal, 0xF6 if 3 bytes)
- Packet Indicator/Stream Sequence Number: 2 bytes
    - 0x0000 indicates a Packet.
- Destination address: 6 bytes (See below for address encoding.)
- Source address: 6 bytes (See below for address encoding.)
- Length: 2 bytes
    - Number of bytes in payload, not including any headers.
- Packet Type: 1 byte
- Payload: N bytes
- CRC: 4 bytes
    - 32-bit CRC of the entire frame, not including the Preamble, Sync word, or Packet Indicator (which are all constants.) Includes Destination, Source, Lengh, Packet Type, and Payload.

TODO More detail here about endianness, etc.

2 Stream Mode

In Stream Mode, an indefinite amount of payload data is sent continuously without breaks in the Physical layer. The Stream is broken up into parts, called Frames to not confuse them with Packets sent in Packet mode. Frames contain payload data wrapped with framing (similar to packets).

A portion of each Frame contains a portion of the Start Stream packet that was used to establish the Stream. Frames are grouped into Super Frames, which is the group of Frames that contain everything needed to rebuild the original Start Stream packet, so that a receiver who starts listening in the middle of a stream is eventually able to reconstruct the Start Stream message and understand how to receive the in-progress stream.

2.1 Stream Frame Format

All Stream frames are 96 bytes long.

Frames have the following format:

- Sync: 2 bytes, 0x3243 (Pi in Hexidecimal (0xF6 if we switch back to 3 bytes)
- Packet Indicator/Stream Sequence Number: 2 bytes
    - The Start Stream Packet that starts a stream is Sequence Number 0x0000. The first stream frame starts at 0x0001 and increases from there.
- Payload: 84 bytes
- CRC: 4 bytes
    - A 32-bit CRC of the entire frame, not including the Sync word. Includes Sequence Number, Payload, four bytes of 0x00 where the message integrity value goes in the frame, and the Preamble/Start Stream.
- Preamble/Start Stream: 4 bytes
    - Every frame of the Super Frame except the last, this contains 4 bytes of the Start Stream message that established this stream.
    - The last frame of the Super Frame, this contains 4 bytes of Preamble (value defined in Physical Link section.) Combined with the immediately following Sync header of the next frame, these two will wake-up a receiver to the stream in progress, if it wasn't already awake.

Assuming a 9600bps Physical layer:

- A Stream frame is sent every 80ms.
- This Stream Frame format gives 8400bps of payload throughput, an 87.5% efficiency.

All forward error correction, if required, is done at the application layer.

2.2 Super Frames

Frames are grouped together into super frames, enough frames in a super frame to resend the entire StartStream packet, 4 bytes at a time, in the Preamble/Start Stream section of the frame, plus one more frame to send a Preamble. The Preamble signifies the beginning of a new super frame.

IV. Application Layer

This section describes the actual Packet and Stream payloads.

1 Packet Formats

1.1 Start Stream

Data Link Layer values:

- Length = 16 bytes
- Packet Type = 0x00

Format:

- Stream Type: 2 bytes
- Stream Subtype: 2 bytes
- Encryption Type: 2 bytes
- Encryption Subtype: 2 bytes
- Encryption Key Index: 4 bytes
    - Index to a known preshared key, or a recently negotiated session key.
- Padding Type: 2 bytes
- Padding Subtype: 2 bytes

Stream Types and Subtypes are defined below in `Stream Types`. Encryption Type and Subtype are defined below in `Encryption Types`. Padding Type and Subtype are defined below in `Padding Types`.

1.2 Identity Beacon Packet

Data Link Layer Packet Type = 0x01

TODO Flesh this out

Notes:

- The sender's callsign is already encoded in the header, so we don't need to include that here.
- Everything is an optional field? I can't think of anything to REQUIRE here.
- Optional fields:
    - Arbitrary string of personal data? eg: "@SmittyHalibut on Twitter" or "Mark's Car". Akin to APRS's "Status message."
    - Location data: Lat/Long
    - Station Type/Icon: Literally copy from APRS?

1.3 Key Negotiation

Used to negotiate a session key with a station with whom you do not have a preshared key. Probably a whole protocol in here. **TODO**

1.4 Ping

Ping? Pong. Test availability to a particular station. Both the echo request and reply are defined. **TODO**

2 Stream Types

These formats must fit in the 84 byte payload of the Stream Mode Frame specified above. A Frame will be sent 12.5 times a second, or one frame every 80ms.

2.1 Voice Streams

The Start Stream packet that establishes the Voice Stream specifies which stream type is used.

The Start Stream packet also specifies what the padding is, whether it's just empty padding, or some other data stream. But for the purposes of the primary Voice Stream, it's just padding. Any use of the padding data is outside the scope of the primary voice stream.

CODEC2, All bit rates

Stream Type = 0xC2

CODEC2 modes operate on either 20ms frames, or 40ms frames. Stream frames are sent every 80ms, so will contain either 2 or 4 CODEC frames, depending on the CODEC mode used. Different CODEC bit-rates result in varying amounts of FEC and padding available for other purposes.

All 20ms modes put 4 CODEC frames per Stream frame. All 40ms modes put 2 CODEC frames per Stream frame.

TODO Provide more detail here about the FEC encoding.

CODEC2 3200bps

Stream Type = 0xC2, Stream Subtype = 0x32

CODEC2 3200 uses a 64 bit (8 byte) CODEC frame every 20ms. FEC rate is ½.

Frame format:

- CODEC2: 8 bytes
- CODEC2: 8 bytes
- CODEC2: 8 bytes
- CODEC2: 8 bytes
- FEC: 32 bytes
- Padding: 20 bytes

CODEC2 2400bps

Stream Type = 0xC2, Stream Subtype = 0x24 CODEC2 2400 uses a 48 bit (6 bytes) CODEC frame every 20ms. FEC rate is ½.

Frame format:

- CODEC2: 6 bytes
- CODEC2: 6 bytes
- CODEC2: 6 bytes
- CODEC2: 6 bytes
- FEC: 24 bytes
- Padding: 36 bytes

CODEC2 2400bps, Resilient Mode

Stream Type = 0xC2, Stream Subtype = 0x25 **TODO** Come up with a better name than "Resilient Mode."

CODEC2 2400 uses a 48 bit (6 bytes) CODEC frame every 20ms. FEC rate is 1/3 (twice as much FEC as data).

Frame format:

- CODEC2: 6 bytes
- CODEC2: 6 bytes
- CODEC2: 6 bytes
- CODEC2: 6 bytes
- FEC: 48 bytes
- Padding: 12 bytes

CODEC2_MODE_1600

Stream Type = 0xC2, Stream Subtype = 0x16

CODEC2 1600 uses a 64 bit (8 bytes) CODEC frame every 40ms. FEC rate is ½.

Frame format:

- CODEC2: 8 bytes
- CODEC2: 8 bytes
- FEC: 32 bytes
- Padding: 36 bytes

CODEC2_MODE_1400

Stream Type = 0xC2, Stream Subtype = 0x14

CODEC2_MODE_1300

Stream Type = 0xC2, Stream Subtype = 0x13

CODEC2_MODE_1200

Stream Type = 0xC2, Stream Subtype = 0x12

CODEC2_MODE_700

Stream Type = 0xC2, Stream Subtype = 0x07

2.2 File Transfer Stream

TODO Notes:

- Include filename, size, MIME type, etc.

3 Encryption Types

TODO Notes:

- Encryption uses CTR mode block ciphers and user the Sequence Number as the counter. The 16 bit counter and 80ms frames can provide for over 87 minutes of streaming without rolling over the counter.

3.1 Null Encryption

Encryption Type = 0x00, Encryption Subtype = 0x00. No encryption is performed, payload is sent in clear text.

3.2 Scrambler

Encryption type = 0x21

Scrambling is an encryption algorithm that is a bit inversion using a bitwise *exclusive-or* (XOR) operation between bit sequence of data and pseudorandom bit sequence.

Encrypting bitstream is generated using a Fibonacci-topology *Linear-Feedback Shift Register* (LFSR). Three different LFSR sizes are available: 8, 16 ant 24-bit. Each shift register has an associated polynomial. The polynomials are listed in ***Table 2***. The LFSR is initialised with a *seed value* of the same length, as the shift register. Seed value acts as an encryption key. for Figures 2 to 4 show block diagrams of the algorithm.

Table 2: LFSR scrambler polynomials

Scrambling subtype	LFSR polynomial	Seed length	Sequence period
0x00	$x^8+x^6+x^5+x^4+1$	8-bit	255
0x01	$x^{16}+x^{15}+x^{13}+x^4+1$	16-bit	65,535
0x02	$x^{24}+x^{23}+x^{22}+x^{17}+1$	24-bit	16,777,215

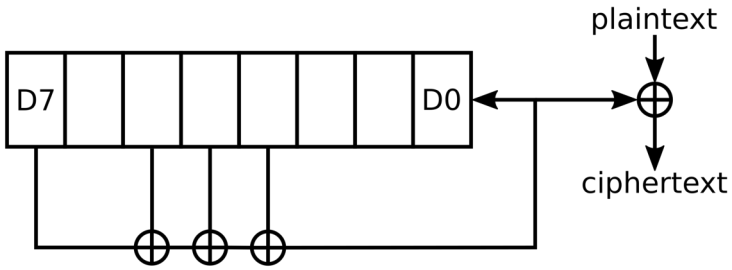

Figure 2: 8-bit LFSR taps

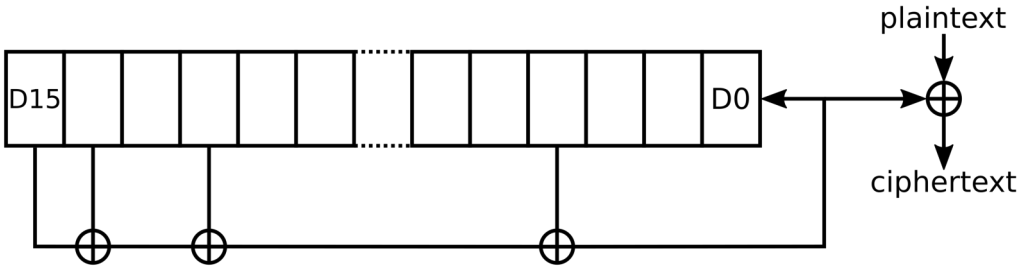

Figure 3: 16-bit LFSR taps

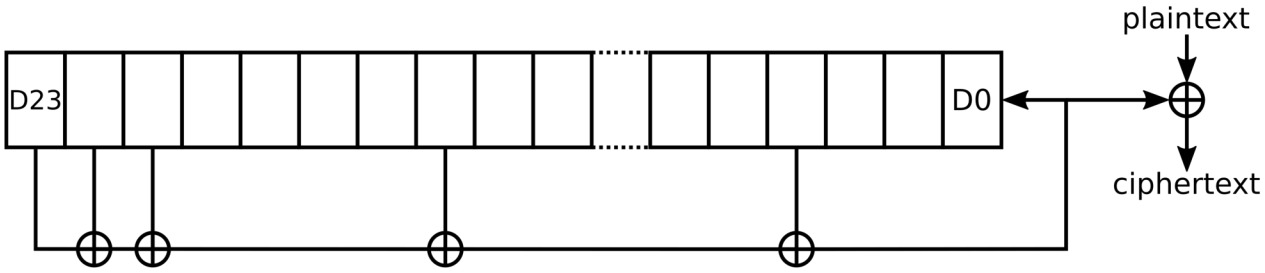

Figure 4: 24-bit LFSR taps

4 Padding Types

These might be very similar to Stream Types, but they have variable number of bytes per frame available to them. There is no guaranteed delivery rate like there is for a Stream Type.

4.1 Null Padding

Padding Type = 0x00, Padding Subtype = 0x00. Simply fill the empty space with 0x00.

4.2 HMAC

Padding Type = 0x01, Padding Subtype = 0xNN. The first 32 bytes of the padding is filled with an HMAC-SHA256 of the payload, keyed on a private key pre-shared between the sender and recipient. The Padding Subtype is used as a key index if the sender and recipient have multiple pre-shared keys. Any remaining bytes of padding are filled with 0x00 (Null padding.)

If only N<32 bytes of padding are available, then the N most significant bytes of the HMAC output are used. If N<4, the recipient may choose to not trust the sender.

V. Data Examples

Here's an example of what a Start Stream packet, followed by a CODEC2 3200bps Voice Stream might look like:

- Data Link:
    - Preamble: 0b10101010 0b10101010
    - Sync: 0x3243F6
    - Packet Indicator: 0x0000
    - Destination:
    - Source:
    - Length: 0x0010
    - Packet Type: 0x00
- Application: *

VI. Address Encoding

M17 addresses are 48 bits, 6 bytes long. Callsigns (and other addresses) are encoded into these 6 bytes in the following ways:

- An address of 0 is invalid.
    - **TODO** Do we want to use zero as a flag value of some kind?
- Address values between 1 and 262143999999999 (which is (40^9)-1), up to 9 characters of text are encoded using base40, described below.
- Address values between 262144000000000 (40^9) and 281474976710654 ((2^48)-2) are invalid
    - **TODO** Can we think of something to do with these 19330976710654 addresses?
- An address of 0xFFFFFFFFFFFF is a broadcast. All stations should receive and listen to this message.

1 Callsign Encoding: base40

9 characters from an alphabet of 40 possible characters can be encoded into 48 bits, 6 bytes. The base40 alphabet is:

- 0: An invalid character, something not in the alphabet was provided.
- 1-26: 'A' through 'Z'
- 27-36: '0' through '9'
- 37: '-'
- 38: '/'
- 39: TBD

Encoding is little endian. That is, the right most characters in the encoded string are the most significant bits in the resulting encoding.

1.1 Example code: encode_base40()

```
uint64_t encode_callsign_base40(const char *callsign) {
  uint64_t encoded = 0;
  for (const char *p = (callsign + strlen(callsign) - 1); p >= callsign; p-- ) {
    encoded *= 40;
    // If speed is more important than code space, you can replace this with a lookup into a 256 byte array.
    if (*p >= 'A' && *p <= 'Z')  // 1-26
      encoded += *p - 'A' + 1;
    else if (*p >= '0' && *p <= '9')  // 27-36
      encoded += *p - '0' + 27;
    else if (*p == '-')  // 37
      encoded += 37;
    // These are just place holders. If other characters make more sense, change these.
    // Be sure to change them in the decode array below too.
    else if (*p == '/')  // 38
      encoded += 38;
    else if (*p == '.')  // 39
      encoded += 39;
    else
      // Invalid character, represented by 0.
      //encoded += 0;
      ;
  }
  return encoded;
}
```

1.2 Example code: decode_base40()

```
char *decode_callsign_base40(uint64_t encoded, char *callsign) {
  if (encoded >= 262144000000000) {   // 40^9
    *callsign = 0;
    return callsign;
  }

  char *p = callsign;
  for (; encoded > 0; p++) {
    *p = "xABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-/."[encoded % 40];
    encoded /= 40;
  }
  *p = 0;
  return callsign;
}
```

1.3 Why base40?

The longest commonly assigned callsign from the FCC is 6 characters. The minimum alphabet of A-Z, 0-9, and a "done" character mean the most compact encoding of an American callsign could be: log2(37^6)=31.26 bits, or 4 bytes.

Some countries use longer callsigns, and the US sometimes issues longer special event callsigns. Also, we want to extend our callsigns (see below). So we want more than 6 characters. How many bits do we need to represent more characters:

- 7 characters: log2(37^7)=36.47 bits, 5 bytes
- 8 characters: log2(37^8)=41.67 bits, 6 bytes
- 9 characters: log2(37^9)=46.89 bits, 6 bytes
- 10 characters: log2(37^10)=52.09 bits, 7 bytes.

Of these, 9 characters into 6 bytes seems the sweet spot. Given 9 characters, how large can we make the alphabet without using more than 6 bytes?

- 37 alphabet: log2(37^9)=46.89 bits, 6 bytes
- 38 alphabet: log2(38^9)=47.23 bits, 6 bytes
- 39 alphabet: log2(39^9)=47.57 bits, 6 bytes
- 40 alphabet: log2(40^9)=47.90 bits, 6 bytes
- 41 alphabet: log2(41^9)=48.22 bits, 7 bytes

Given this, 9 characters from an alphabet of 40 possible characters, makes maximal use of 6 bytes.

2 Callsign Formats

Government issued callsigns should be able to encode directly with no changes.

2.1 Multiple Stations

To allow for multiple stations by the same operator, we borrow the use of the '-' character from AX.25 and the SSID field. A callsign such as "KR6ZY-1" is considered a different station than "KR6ZY-2" or even "KR6ZY", but it is understood that these all belong to the same operator, "KR6ZY".

2.2 Temporary Modifiers

Similarly, suffixes are often added to callsign to indicate temporary changes of status, such as "KR6ZY/M" for a mobile station, or "KR6ZY/AE" to signify that I have Amateur Extra operating privileges even though the FCC database may not yet be updated. So the '/' is included in the base40 alphabet.

The difference between '-' and '/' is that '-' are considered different stations, but '/' are NOT. They are considered to be a temporary modification to the same station. **TODO** I'm not sure what impact this actually has.

2.3 Interoperability

It may be desirable to bridge information between M17 and other networks. The 9 character base40 encoding allows for this:

TODO Define more interoperability standards here. System Fusion? P25? IRLP? AllStar?

2.3.1 DMR

DMR unfortunately doesn't have a guaranteed single name space. Individual IDs are reasonably well recognized to be managed by https://www.radioid.net/database/search#! but Talk Groups are much less well managed. Talk Group XYZ on Brandmeister may be (and often is) different than Talk Group XYZ on a private cBridge system.

- DMR IDs are encoded as: `D<number>` eg: `D3106728` for KR6ZY
- DMR Talk Groups are encoded by their network. Currently, the following networks are defined:
    - Brandmeister: `BM<number>` eg: `BM31075`
    - More networks to be defined here.

2.3.2 D-Star

D-Star reflectors have well defined names: REFxxxY which are encoded directly into base40.

TODO Individuals? Just callsigns?

2.3.3 Interoperability Challenges

- We'll need to provide a source ID on the other network. Not sure how to do that, and it'll probably be unique for each network we want to interoperate with. Maybe write the DMR/BM gateway to automatically lookup a callsign in the DMR database and map it to a DMR ID? Just thinking out loud.
- We will have to transcode CODEC2 to whatever the other network uses (pretty much AMBE of one flavor or another.) I'd be curious to see how that sounds.