M17 Protocol Specification

M17 Working Group

DRAFT

Authors:

Mark KR6ZY

Wojciech SP5WWP

Steve KC1AWV

Nikoloz SO3ALG

Juhani OH1CAU

Formatted by Steve Miller KC1AWV and Juhani Krekelä OH1CAU

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
30 May 2020	Removed a lot of outdated stuff, added error coding info – SP5WWP
6 Jun 2020	Added a table of acronyms, updated description of frame contents, added more info about error coding, changed structure a little – OH1CAU
9 Jun 2020	Added convolutional coding and puncturing info – SP5WWP
10 Aug 2020	Specified the bit and byte ordering as well as the CRC details – OH1CAU
15 Aug 2020	Added bibliography, voice coder rates for different data type indicators, link setup frame update – SP5WWP
18 Aug 2020	Added new encryption type and puncturing patterns – SP5WWP
22 Aug 2020	NONCE field structure update – SP5WWP
23 Aug 2020	NONCE field structure update – OH1CAU
29 Aug 2020	Added data whitening algorithm – SP5WWP
30 Aug 2020	Fix CRC test vectors – OH1CAU

Acronyms used in this document

FSK	Frequency Shift Keying
4FSK	Quaternary FSK
BPS	Bits Per Second
PTT	Push To Talk
V+D	Voice plus Data
AES	Advanced Encryption Standard
CTR	Counter (stream cipher mode)
LICH	Link Information CHannel
ECC	Error Correction Coding
FN	Frame number
CRC	Cyclic Redundancy Check

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 (9600 bits/s) with a deviation index *h*=0.33 for transmission in 6.25 kHz channel bandwidth. Channel spacing is 12.5 kHz. 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.

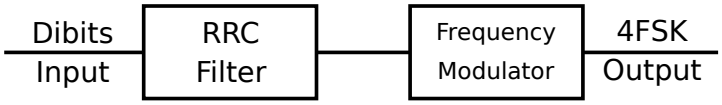

Figure 1: 4FSK modulator

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

The most significant bits are sent first, meaning that the byte 0xB4 in type 4 bits (see section 3) would be sent as the symbols -1 -3 +3 +1.

2 Preamble

Every transmission starts with a **preamble**, which shall consist of at least 40ms of alternating -3, +3... symbols. This is equivalent to 40 milliseconds of a 2400 Hz tone.

3 Bit types

The bits at different stages of the error correction coding are referred to with bit types, given in **Table 2.**

Table 2: Bit types

Type 1	Data link layer data
Type 2	Type 1 bits after appropriate encoding
Type 3	Type 2 bits after puncturing (only for convolutionally coded data, for other ECC schemes type 3 bits are the same as type 2 bits)
Type 4	Whitened and interleaved (re-ordered) type 3 bits

Type 4 bits are used for transmission over the RF. Incoming type 4 bits shall be decoded to type 1 bits, which are then used to extract all the frame fields.

4 Error correction coding schemes and bit type conversion

Two distinct ECC schemes are used for different parts of the transmission.

4.1 Link setup frame

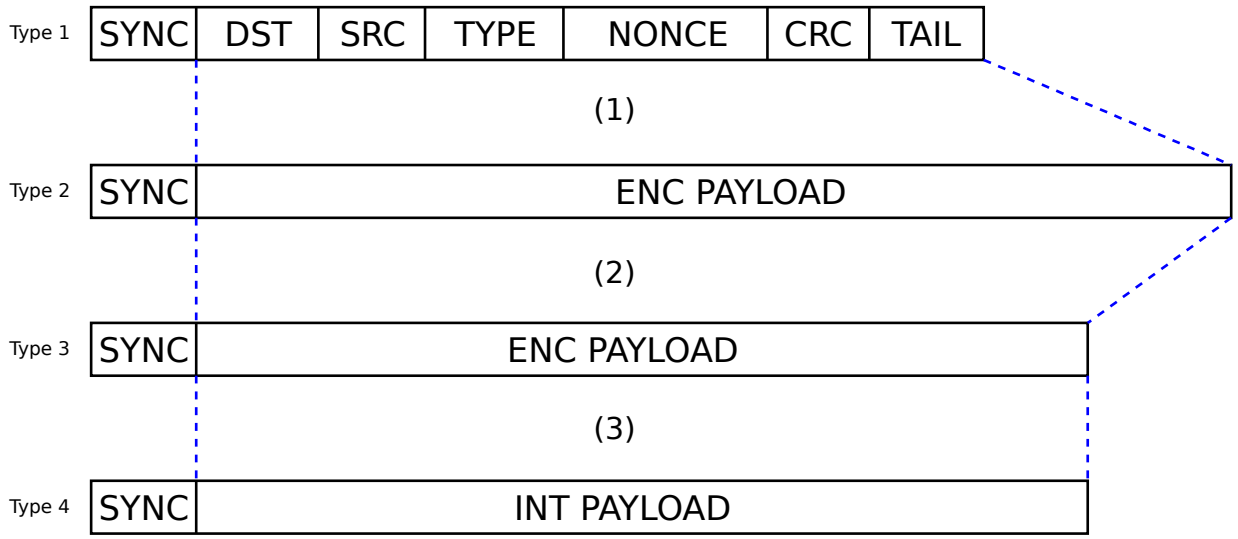

Figure 2: ECC stages for the link setup frame

240 **DST**, **SRC**, **TYPE, NONCE** and **CRC** type 1 bits are convolutionally coded using rate 1/2 coder with constraint K=5. 4 tail bits are used to flush the encoder's state register, giving a total of 244 bits being encoded. Resulting 488 type 2 bits are retained for type 3 bits computation. Type 3 bits are

computed by puncturing type 2 bits using a scheme shown in chapter **4.4**. This results in 368 bits, which in conjunction with the **synchronization burst** gives 384 bits (384 bits / 9600bps = 40 ms).

Interleaving type 3 bits produce type 4 bits that are ready to be transmitted. Interleaving is used to combat error bursts.

4.2 Subsequent frames

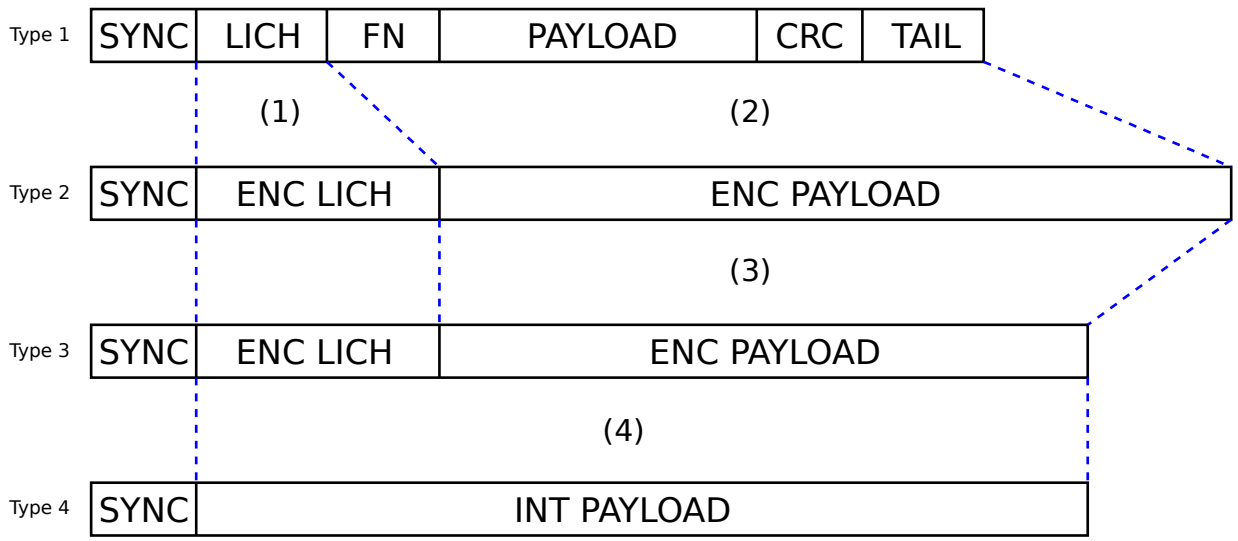

Figure 3: ECC stages of subsequent frames

A 48-bit (type 1) chunk of **LICH** is partitioned into 4 12-bit parts and encoded using Golay (24, 12) code. This produces 96 encoded **LICH** bits of type 2.

164 **FN**, **payload** and **CRC** bits are convolutionally encoded in a manner analogous to that of the link setup frame. A total of 168 bits is being encoded resulting in 336 type 2 bits. These bits are punctured to generate 272 type 3 bits.

96 type 2 bits of **LICH** are concatenated with 272 type 3 bits and re-ordered to form type 4 bits for transmission. This, along with 16-bit sync in the beginning of frame, gives a total of 384 bits.

4.3 Convolutional encoder

The convolutional code shall encode the input bit sequence after appending 4 tail bits at the end of the sequence. Rate of the coder is R=½ with constraint length K=5. The encoder diagram and generating polynomials are shown below.

$$G_1(D) = 1 + D^3 + D^4$$
$$G_2(D) = 1 + D + D^2 + D^4$$

The output from the encoder must be read alternately.

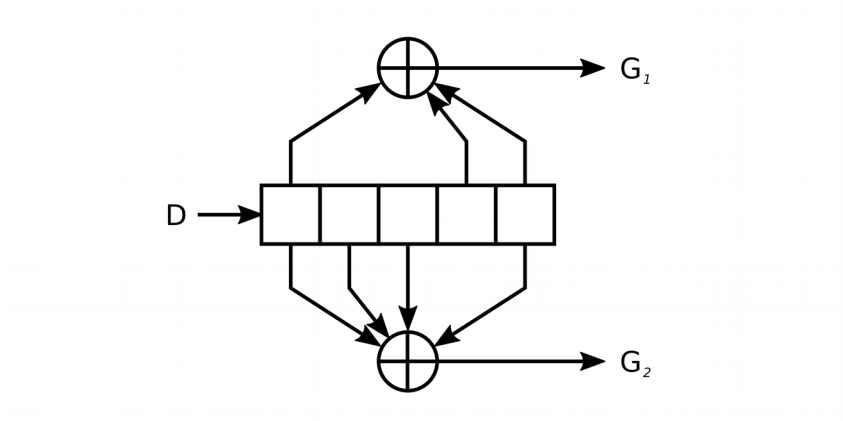

Figure 4: Convolutional coder diagram

4.4 Code puncturing

Removing some of the bits from the convolutional coder's output is called *code puncturing*. The nominal coding rate of the encoder used in M17 is ½. This means the encoder ouputs two bits for every bit of the input data stream. To get other (higher) coding rates, a puncturing scheme has to be used.

Two different puncturing schemes are used in M17:

I. leaving 46 from 61 encoded bits

II. leaving 34 from 41 encoded bits

Both puncturers are defined by their puncturing matrices:

$$P_1 = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & X \end{bmatrix}$$

$$P_2 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & X \end{bmatrix}$$

Scheme I is used for the initial LICH link setup info, while scheme II is for frames (excluding LICH chunks, which are coded differently).

TODO: explain what's the X for

4.5 Data whitening

To avoid transmitting long sequences of constant symbols (e.g. 010101…), a simple data whitening algorithm is used. All 46 bytes of type 4 bits shall be XORed with a pseudorandom, predefined stream[1]. The same algorithm has to be used for incoming bits at the receiver to get the original data stream.

1 See **Appendix 2** for details.

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: 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.~~

As is the convention with networking protocols, all quantities larger than 8 bits are encoded in big-endian.

1 Packet Mode

In *packet mode*, a finite amount of payload data (for example – text 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

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 interleaved with frame signalling (similar to packets). Frame signalling is contained within the ***Link Information Channel*** (**LICH**).

All frames are preceded by a 16-bit ***synchronization burst***, which consists of 0x3243 (first 16-bit of pi) in type 4 bits.

2.1 Link setup frame

First frame of the transmission contains full **LICH** data. It’s called the *link setup frame*, and is not part of any superframes.

Table 3: Link setup frame fields

DST	48 bits	Destination address - Encoded callsign or a special number (eg. a group
SRC	48 bits	Source address - Encoded callsign of the originator or a special number (eg. a group)
TYPE	16 bits	Information about the incoming data stream
NONCE	112 bits	Nonce for encryption
CRC	16 bits	CRC for the link setup data
TAIL	4 bits	Flushing bits for the convolutional encoder that do not carry any information

Table 4: Bitfields of type field

Bit 0	Packet/stream indicator, 0=packet, 1=stream
Bits 1, 2	Data type indicator, 01_2=data (D), 10_2=voice (V), 11_2=V+D, 00_2=reserved
Bits 3, 4	Encryption type, 00_2=none, 01_2=AES, 10_2=scrambling, 11_2=other/reserved
Bits 5, 6	Encryption subtype (meaning of values depends on encryption type)
Bits 7…15	Reserved (don't care)

The fields in **Table 3** (except **tail**) form initial **LICH**. It contains all information needed to establish M17 link. Later in the transmission, the initial **LICH** is divided into 5 "chunks" and transmitted interleaved with data. The purpose of that is to allow late-joiners to receive the **LICH** at any point of the transmission. The process of collecting full **LICH** takes 5 frames or 5*40 ms = 200 ms. Four TAIL bits are needed for the convolutional coder to go back to state 0, so also the ending trellis position is known.

Voice coder rate is inferred from TYPE field, bits 1 and 2.

Table 5: Voice coder rates for different data type indicators

Data type indicator	Voice coder rate
00_2	none/reserved
01_2	no voice
10_2	3200 bps
11_2	1600 bps

2.2 Subsequent frames

Table 6: Fields for frames other than the link setup frame

LICH	48 bits	LICH chunk, one of 5
FN	16 bits	Frame number, starts from 0 and increments every frame
PAYLOAD	128 bits	Payload/data, can contain arbitrary data
CRC	16 bits	This field contains 16-bit value used to check data integrity, see section 2.4 for details
TAIL	4 bits	Flushing bits for the convolutional encoder that don't carry any information

2.3 Superframes

Each frame contains a chunk of the **LICH** frame that was used to establish the stream. Frames are grouped into ***superframes***, which is the group of 5 frames that contain everything needed to rebuild the original **LICH** packet, so that the user who starts listening in the middle of a stream (*late-joiner*) is eventually able to reconstruct the **LICH** message and understand how to receive the in-progress stream.

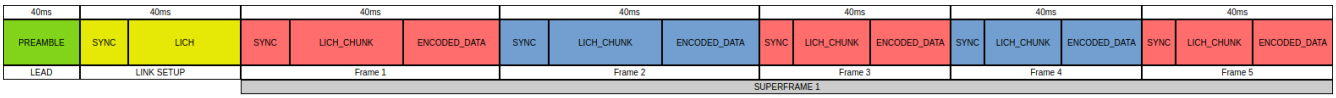

Figure 5: Stream consisting of one superframe

2.4 CRC

M17 uses a non-standard version of 16-bit CRC with polynomial $x^{16}+x^{14}+x^{12}+x^{11}+x^{8}+x^{5}+x^{4}+x^{2}+1$ or 0xAC9A and initial value of 0xFFFF. This polynomial allows for detecting all errors up to hamming distance of 5 with payloads up to 241 bits[2], which is less than the amount of data in each frame.

As M17's native bit order is most significant bit first, neither the input nor the output of the CRC algorithm gets reflected.

The input to the CRC algorithm consists of the 48 bits of LICH, 16 bits of FN, 128 bits of payload, and then depending on whether the CRC is being computed or verified either 16 zero bits or the received CRC.

The test vectors in **Table 6** are calculated by feeding the given message and then 16 zero bits to the CRC algorithm.

2 https://users.ece.cmu.edu/~koopman/crc/

Table 7: CRC test vectors

Message	CRC output
(empty string)	0xFFFF
ASCII string “A”	0xCDB4
ASCII string “123456789”	0x9630
Bytes from 0x00 to 0xFF	0x6496

IV. Application Layer

PARTS 1 AND 2 REMOVED – will add this later.

3 Encryption Types

Encryption is optional and disabled by default. The use of it is only allowed if local laws allow to do so.

3.1 Null Encryption

Encryption type = 00_2

No encryption is performed, payload is sent in clear text.

3.2 Scrambler

Encryption type = 10_2

Scrambling is an encryption by 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 and 24-bit. Each shift register has an associated polynomial. The polynomials are listed in **Table 7**. The LFSR is initialised with a *seed value* of the same length as the shift register. Seed value acts as an encryption key for the scrambler algorithm. **Figures 5 to 8** show block diagrams of the algorithm.

Table 8: LFSR scrambler polynomials

Encryption subtype	LFSR polynomial	Seed length	Sequence period
00_2	$x^8+x^6+x^5+x^4+1$	8 bits	255
01_2	$x^{16}+x^{15}+x^{13}+x^4+1$	16 bits	65,535
10_2	$x^{24}+x^{23}+x^{22}+x^{17}+1$	24 bits	16,777,215

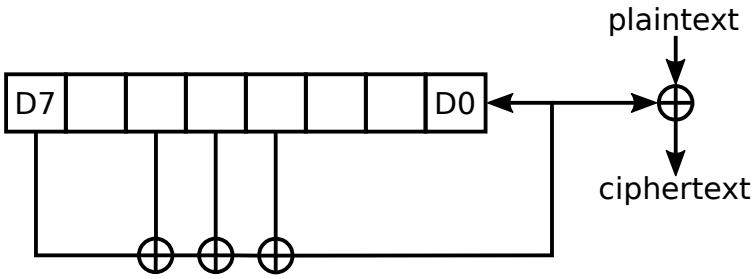

Figure 6: 8-bit LFSR taps

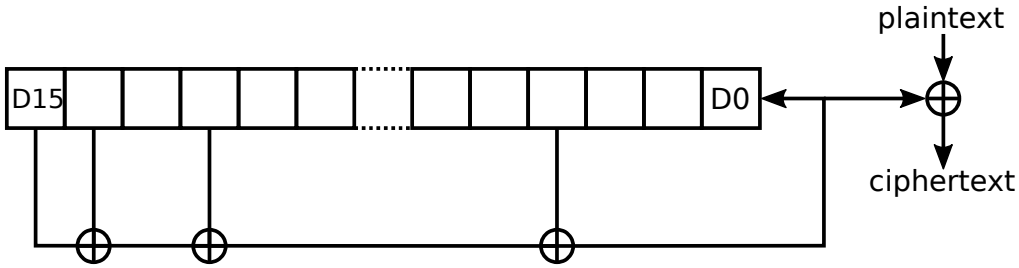

Figure 7: 16-bit LFSR taps

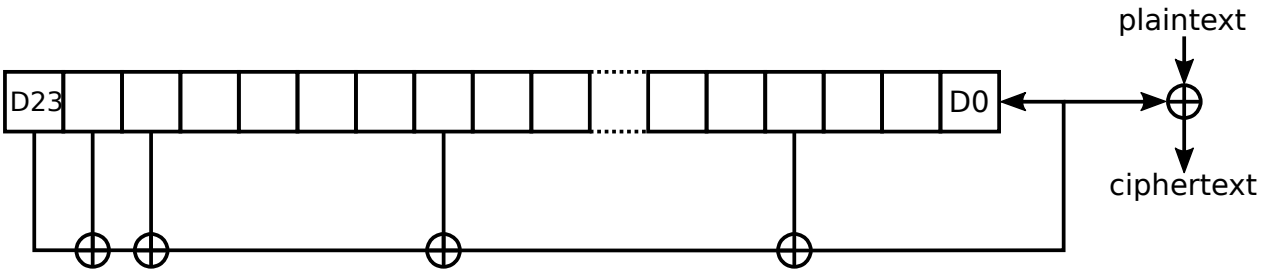

Figure 8: 24-bit LFSR taps

3.3 Advanced Encryption Standard (AES)

Encryption type = 10_2

This method uses AES block cipher in counter (CTR) mode. 96-bit nonce value is extracted from the NONCE field, as the 96 most significant bits of it. The highest 16 bits of the counter are the remaining 16 bits of the NONCE field. FN field value is then used as the counter. The 16 bit frame counter and 40 ms frames can provide for over 43 minutes of streaming without rolling over the counter. This method adapts 16-bit counter to the standard 32-bit CTR for the encryption. FN counter always start from 0 (zero).

The nonce value should be generated with a hardware random number generator or any other method of generating non-repeating values. Nonce values must be used only once. It is obvious that with a finite number of nonce bits, the probability of nonce collision approaches 1. We assume that the transmission is secure for 2^{37} frames using a single key. It is recommended to change keys after that period.

To combat replay attacks, a 64-bit timestamp shall be embedded into the NONCE field. The field structure is shown in **Table 9**. Timestamp is the number of seconds that elapsed since the beginning of 1970-01-01, 00:00:00 UTC, minus leap seconds (a.k.a. “unix time”).

Table 9: NONCE field structure

TIMESTAMP	NONCE	CTR_HIGH
64	32	16

CTR_HIGH field initializes the highest 16 bits of the CTR, with the rest of the counter being equal to the **FN** counter.

Appendix 1. Address Encoding

M17 uses 48 bits (6 bytes) long addresses. 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: “-” (hyphen)
- 38: “/” (slash)
- 39: “.” (dot)

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: $\log_2(37^6)=31.26$ 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.

Appendix 2. Data whitening sequence

Seq. number	Value
0	0xD6
1	0xB5
2	0xE2
3	0x30
4	0x82
5	0xFF
6	0x84
7	0x62
8	0xBA
9	0x4E
10	0x96
11	0x90
12	0xD8
13	0x98
14	0xDD
15	0x5D
16	0x0C
17	0xC8
18	0x52
19	0x43
20	0x91
21	0x1D
22	0xF8

Seq. number	Value
23	0x6E
24	0x68
25	0x2F
26	0x35
27	0xDA
28	0x14
29	0xEA
30	0xCD
31	0x76
32	0x19
33	0x8D
34	0xD5
35	0x80
36	0xD1
37	0x33
38	0x87
39	0x13
40	0x57
41	0x18
42	0x2D
43	0x29
44	0x78
45	0xC3

Bibliography

1. Moreira, Jorge C.; Farrell, Patrick G. “Essentials of Error Control Coding”
   Wiley 2006, ISBN: 9780470029206
2. Dunlop, John; Girma, Demessie; Irvine, James “Digital Mobile Communications and the TETRA System”
   Wiley 1999, ISBN: 9780471987925
3. NXDN Technical Specifications, Part 1: Air Interface; Sub-part A: Common Air Interface
4. ETSI TS 102 361-1 V2.2.1 (2013-02): “Electromagnetic compatibility and Radio spectrum Matters (ERM); Digital Mobile Radio (DMR) Systems; Part 1: DMR Air Interface (AI) protocol”
   https://www.etsi.org/deliver/etsi_ts/102300_102399/10236101/02.02.01_60/ts_10236101v020201p.pdf