

KEN-A PROTOCOL

By: Ken McCaughey

On: 2025-03-21

Ver 1.1.0

Ken's Electronic Network (KEN) Advanced Protocol


This serial communications protocol is intended for small micro-controller applications. This data link layer protocol specifies frame elements that define a variety of features. This protocol does not attempt to define all the rules in how they should or could be used. There are considerable permutations and it is up to the user to decide what fits a given application. Not all permutations that can be created are compatible with each other. The intent of this protocol is to have a set of tools and a structure to facilitate communications to meet a particular application. Only implement what is needed for a given application and tailor as needed.

Universal compatibility is not the goal of this protocol. Ease of use and implementation is the goal.

Contents

- License
- Features
- Not Included
- Basic Rules
- Frame Format
- Flags and Header Control Elements
- Error Management
- Data
- Protocol Type
- Usage Tips
- Complexity Hierarchy
- Example Messages
- References
- Older Ideas

LICENSE

Ken's Electronic Network (KEN) Advanced Protocol (KEN-A), copyright 2025, by Ken McCaughey is licensed under [CC BY-SA 4.0](#) 

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material for any purpose, even commercially.

This license is acceptable for Free Cultural Works.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

FEATURES

- Intended for small micro-controllers.
- Good for point-to-point messaging as well as small networks.
- Borrows ideas from MIDI, BiSync, and S.N.A.P. protocols.
- Features modular/scalable header elements.
 - Trades link overhead efficiency for modularity and parsing simplicity.
 - Header components can be mixed and matched as needed.
 - Allows for custom fields or features.
- Designed to be somewhat human readable with a hex viewer.
 - Most payload data is 7-bit (ASCII) with limited support for binary data.
 - Headers are nibble based.
 - First nibble denotes element function.
 - Second nibble contains variables.
- All control features in a header have the msb=1.
- Message sizes scalable based on complexity and options used.
 - Can be as short as 3-bytes.
 - Can optionally specify data length in header.
 - Nominal maximum data payload up to 14 ASCII bytes.
 - Can optionally extended data payload to 127 bytes.
 - Can have unlimited length if header data length control is not used.
 - Parsing data based on header flags only.
 - Checksum effectiveness decreases with messages longer than 256 bytes.
- Supports optional addressing.
 - Default simple addressing is up to 14 nodes.
 - Can optionally be extended to 127 nodes.
- Supports optional connection and error management.
 - Ack/nack.
 - Connect/disconnect.
 - Can use checksum or CRC error detection.
 - Supports message sequence numbering.
- Supports frame numbering with total frame count.
 - Used for messages spanning up to 127 frames.

NOT INCLUDED

These are the things NOT included or specified in this protocol:

- Physical layer.
 - Timing rules.
 - Rules for managing connections and error management.
 - Header size optimization.
 - Byte (or bit) stuffing.
 - CRC-32 checksum.
 - Mechanism to auto assign addresses.
 - Rules to enforce compatibility between different applications.
 - Feature for universal plug-and-play like USB or TCP/IP.
-

BASIC RULES

- Keep it simple.
 - Only used features that are needed.
 - Add complexity only as needed.
- All frame header elements are binary with msb=1.
- Frame payload data is ASCII with msb=0.
 - There is a limited provision for binary payload data.
- All frames have a required beginning and ending flags.
- Other than the two required frame flags, all other features are optional and can be mixed and matched as needed.
- If this is not a good fit, take a look at KEN-B or Ken-C.

KEN-A FRAME FORMAT

Note that most significant byte is abbreviated as MSB (upper case) and most significant bit is msb (lower case). Similar for least significant byte or bits, LSB and lsb respectively. All header elements have MSB=1 to distinguish from data with MSB=0.

All message frames begin with `0xFB` and end with `0xFE` flags. All other used header elements (`0x8n` to `0xEn`) are generally in numerical order before the data flag. The checksum, if used, is at the end prior to the end of message flag. Some header parameters may be followed by ASCII variables.

Frame highlights:

- ++ Frame flags, most optional (hexadecimal)
- ** Header control elements, optional (hexadecimal)
- -- Flag or header ASCII component
- == Frame payload data
- ~~ Data covered by checksum

```

Flag [Header/Control Elements] Flag [Data] Flag [Check] Flag
++++ ***** +++++ ===== +++++ ----- +++++
~~~~~

```

```

FB  8n  9n  An  Bn  Cn  Dn  En  FD  [Data]  FC  [Check]  FE
++  **  **  **  **  **  **  **  ++  =====  ++  -----  ++
|   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |--end of frame flag *
|   |   |   |   |   |   |   |   |   |   |   |--checksum
|   |   |   |   |   |   |   |   |   |   |   |-----start check flag
|   |   |   |   |   |   |   |   |   |   |   |--ASCII data
|   |   |   |   |   |   |   |   |   |   |   |-----start of data flag #
|   |   |   |   |   |   |   |   |   |   |   |--error control
|   |   |   |   |   |   |   |   |   |   |   |-----data length
|   |   |   |   |   |   |   |   |   |   |   |-----connection control
|   |   |   |   |   |   |   |   |   |   |   |--to address
|   |   |   |   |   |   |   |   |   |   |   |-----from address
|   |   |   |   |   |   |   |   |   |   |   |--sequence number
|   |   |   |   |   |   |   |   |   |   |   |-----checksum type (start of checksummed characters)
+--beginning of frame flag *

```

* required frame element

```
# there are a different flags for different data types
```

Flags and Header Control Elements

A messaging system can be built using whichever header and control elements are needed. Don't use any that are not needed to reduce overhead. Header elements have their own most significant nibble identifier with msb=1. Header elements can easily be parsed based on most significant nibble. Least significant nibble contains header element parameters. Most header elements are limited to one byte. The complexity and overhead can be tailored for the application. Each of the header and control elements are described in detail below.

0xFn - Frame Flags (0xFB and 0xFE required, all others optional)

Flags are used to mark parts of the frame. Frame elements are located between 0xFB (beginning) & 0xFE (end). In general, place flags at the beginning of the frame after 0xFB, except 0xFC and 0xFE.

```
n = 0 = Null message (optional).
        Recipient ignores this. Can be used for message padding.

n = 1 = Request header feature flags (optional)

n = 2 = Feature flags in following ASCII byte (optional).
        Can requested, with 0xF1, or always specified in a frame.
        Feature flags ASCII byte, set bits to '1' if used.
```

msb					lsb				
0	x	x	x	x	x	x	x	x	x
-	-	-	-	-	-	-	-	-	-
							+	-	checksum (0x8n)
							+	-	sequence number (0x9n)
						+	-	-	from address (0xA _n)
					+	-	-	-	to address (0xB _n)
			+	-	-	-	-	-	connection control (0xC _n)
		+	-	-	-	-	-	-	data length (0xD _n)
	+	-	-	-	-	-	-	-	error control (0xE _n)
+	-	-	-	-	-	-	-	-	always '0'

```
n = 3 = Sync character (optional).
        Sync has the property that no rotation of the bits in this
        byte will equal to the original. Used ahead of a frame. Can
        be used as idle data on communication link between frames.
```

- n = 4 = Nibble encoded data start flag (optional).
Data in lower nibble. Required before data bytes. The msb 4-bits can be all zeros. Or 3-bits used as a count down for the number of nibbles to be concatenated. This permits up to 8 (32-bits) nibbles to be grouped. This type of data immediately follows this flag. Useful if mixing different data types in different messages. This flag can only appear once in a frame and not mixed with other data type flags.
- n = 5 = Ping, requesting a pong (optional).
Used to test a link. Usually used without a data payload.
Cannot have a ping and pong in same message.
- n = 6 = 12-bit ASCII data start flag (optional).
This is for data that encodes 6-bits per byte. Required before data bytes. Assumes data is in pairs. This type of data immediately follows this flag. Useful if mixing different data types in different messages. This flag can only appear once in a frame and not mixed with other data type flags.
- n = 7 = User defined data type flag (optional).
Can have custom data types in following ASCII byte. This flag can only appear once in a frame. The custom data can be ASCII or binary. If binary, then the Data Length header element is required.
- n = 8 = Start of 8-bit binary data flag (optional).
This is for data that encodes 8-bits per byte. This flag is required for binary data along with the data length (0xDn or 0xDF, 0xN) header. The number of binary bytes that immediately follow this flag must match the data length count. Useful if mixing different data types in different messages. This flag can only appear once in a frame and not mixed with other data type flags.
- n = 9 = Sub-frame number and total number of sub-frames. (optional).
Numbers follow in next two bytes, respectively. Can be used if a data set requires spanning multiple messages sub-frames. Sub-frame number is limited to 7-bits (127 frames). If total number of sub-frames is not known or is not needed, then the number of sub-frames byte is set to zero. Place near beginning of the frame.
- n = A = Answer to a ping (i.e. pong).
Used to test a link. Usually used without a data payload.
Cannot have a ping and pong in same message.
- n = B = Beginning of frame (BOF) flag (required).
Always the first byte of a frame.

n = C = Checksum data start flag (optional).
Required if checksum option is used. Place checksum after the data to be covered and before EOF marker. Typically after the data. It could be used prior to the data flag, which would mean the checksum covers only the header. The number of bytes composing the checksum itself is defined by the checksum type flag (0x8n).

n = D = Data start flag for ASCII data (optional).
This is the default data type. Used before ASCII data bytes. Use if the start of data need to be explicitly identified. This type of data immediately follows this flag. This flag can only appear once in a frame. This flag can only appear once in a frame and not mixed with other data type flags.

n = E = End of frame (EOF) flag (required last character in a frame).
Multiple end of frame markers are permitted.

n = F = Follow on ASCII byte has custom flag (optional).
Could be used to indicate a command or message type. Place before data and/or checksum flags. This flag can only appear once in a frame.

Examples:

- Minimal message = 0xFB 0xFD [ASCII message] 0xFE
- Minimal null message = 0xFB 0xF0 0xFE
- Ping 0xFB 0xF5 0xFE with a 0xFB 0xFA 0xFE pong response.
- Nibble data for 0x35 = 0xF4 0x03 0x05
- Binary data for 0x85 = 0xD1 0xF8 0x85

0x8n - Checksum Type (optional)

This feature specifies the type of checksum if one is being used. This is optional. The `0x8n` must immediately follow the `0xFB` flag and acts as a marker for the beginning of the data covered by the specified checksum. The `0xFC` flag marks the end of the check-summed block. The checksum block includes both the `0x8n` and `0xFC` markers. The `0xFB` flag and any data after the `0xFC` flag are not covered by the checksum. Typically the `0xFC` is after the data, thus encompassing the entire frame. It can be used before the data flag, and would thus only be a checksum of the header. Checksum data is nibble based with the countdown. This flag can only appear once in a frame.

`n = 0 = None. Not used or not supported. (no 0xFC marker)`

`n = 1 = 8-bit Modulo Checksum in 2 nibble bytes with countdown.`

`n = 2 = 16-bit Modulo Checksum in 4 nibble bytes with countdown.`

`n = 3 = Fletcher-16 checksum in 4 nibble bytes with countdown.`

`n = 8 = CRC-8 in 2 nibble bytes with countdown.`

`Poly = $x^8+x^5+x^3+x^2+x+1$`

`Covers 10 bytes with Hamming distance of 3 and 4.`

`n = 9 = CRC-12 6sub8 in 3 nibble bytes with countdown.`

`Poly = $x^{12}+x^8+x^7+x^6+x^5+x^2+x+1$`

`Covers 254 bytes with Hamming distance 3 and 4.`

`n = A = CRC-16 6sub8 in 4 nibble bytes with countdown.`

`Poly = $x^{16}+x^8+x^4+x^3+x+1$`

`Covers 3580 bytes with Hamming distance of 3 and 4.`

`Covers 12 bytes with Hamming distance of 5 and 6.`

`n = B = CRC-16 M16 in 4 nibble bytes with countdown.`

`Poly = $x^{16}+x^{14}+x^{12}+x^{11}+x^8+x^5+x^4+x^2+1$`

`Covers 30 bytes with Hamming distance of 3, 4, and 5.`

`n = F = Follow on ASCII byte has custom checksum type code.`

`n = 4 to 7 = Reserved for future non-CRC checksums.`

`n = C to E = Reserved for future CRC polynomials.`

Examples:

- `0x81` and checksum of 34 after the data, `0xFC 0x13 0x04 0xFE` .
- `0x8A` and checksum of 1234 after the data, `0xFC 0x31 0x22 0x13 0x04 0xFE` .

0x9n - Sequence Number (optional)

This optional control element is use if there is a need to keep messages in order, detect a missing message, or identify a repeated message. This is used at the lower link level for channel management and error control. This is not the same as the sub-frame number (see header flag 0xF9), which would be used by higher application level. This can be used with error detection and error control. The count starts at one and increments until it rolls over at 0xE (14). An application could have the rollover set to a lower value. This flag can only appear once in a frame. A larger sequence number can be used with the extended sequence number (0x9F). The following ASCII byte (7-bits, msb=0) is the sequence number.

n = 0 = None. Not being used or not supported.

n = 1 to E = Sequence number (14).

n = F = Follow on ASCII byte has custom sequence number.

0xAn - Address, From (ie. from A to B) (optional)

Use if addressing is needed. Not needed on a point-to-point link. Can be used as an ID code for a set of hardware. Simple addressing uses 1-byte, with up to 14 addresses. Extended addressing permits up to 128 addresses. This flag can only appear once in a frame.

n = 0 = Node does not have an assigned address.

n = 1 to E = simple address (14).

n = F = Following ASCII byte is the address.

N = 00 to 7F = Extended address (7-bits, 0 to 127).

Examples:

- Address 3 = 0xA3 = 0xAF 0x03
- Address "3" = 0xAF 0x33

0xBn - Address, To (ie. from A to B) (optional)

Use if addressing is needed. Not needed on a point-to-point link. Simple addressing uses 1-byte, with up to 14 addresses. Extended addressing permits up to 128 addresses. This flag can only appear once in a frame.

n = 0 = All (Broadcast).

n = 1 to E = simple address (14).

n = F = Following ASCII byte is the address.

N = 00 to 7F = Extended address (7-bits, 0 to 127).

Examples:

- Address 3 = 0xB3 = 0xBF 0x03
- Address 64 = 0xBF 0x40
- Broadcast to all = 0xB0

0xCn - Connection Control (optional)

Use this feature if needing to manage a connected link. Connect request frames can include data. Multiple connected links need to use addressing to distinguish separate links. This flag can only appear once in a frame.

```
n = 0 = Not supported.

n = 1 = Used, but idle.

n = A = Ask to connect.

n = B = Break connection request.

n = C = Connected state.

n = D = Disconnected state.

n = E = Error in connection (rejected).

n = F = Follow on ASCII byte has custom connection code.

n = 2 to 9 = Reserved.
```

Examples:

- To connect, a requester starts with 0xCA in a message. Requestee responds with 0xCC, 0xCD, or 0xCE in a replay message.
- While connected all messages contain 0xCC to reflect the state.
- To disconnect the requester sends a 0xCB. Requestee responds with 0xCD, indicating state is disconnected.
- Connection error, 0xCE, can be sent if either node disagrees with the state.

0xDn - Data Length (optional **)

This optional feature specifies the total number of data bytes after the data flag(s). Use this feature if data length needs to be explicitly specified. This is required if sending binary data (using the 0xF8 flag). Minimum length is 0 bytes, and up to 127 with extended byte. This element can be omitted if there are zero data bytes. This flag can only appear once in a frame.

** The data length header is required for binary data using the 0xF8 flag.

```
n = 0 to E = Length is 0 to 14.
```

```
n = F = Following ASCII byte is the length (7-bits) plus one,  
        up to 128 bytes.
```

```
N = 00 to 7F = Length is 0 to 127 for extended data length.
```

Examples:

- 1-byte data length = 0xD1 or 0xDF 0x01
- 2-bytes data length = 0xD2 or 0xDF 0x02
- 17-bytes data length = 0xDF 0x11
- 127-bytes data length = 0xDF 0x7F

0xEn - Error Control (optional)

Only use this optional feature if error control and management is needed. This flag can only appear once in a frame.

n = 0 = Not supported.

n = 1 = Used, but idle.

n = 5 = Ack/nack request.

n = A = Ack.

n = C = Checksum error.

n = E = Error = Nack.

n = F = Follow on ASCII byte has custom error code.

n = 2 to 4, 6 to 9, B, D = Reserved.

Examples:

- Send message with 0xE0, response is 0xE0.
- Send message with 0xE5, response is 0xEA or 0xEE.
- If only the checksum fails, response is 0xEC.

ERROR MANAGEMENT

Since this protocol depends on parsing based on key flags, there are inherent vulnerabilities if the flags are corrupted or other data is corrupted to match a flag. There are a number of different error scenarios that could be encountered.

- Corrupted message
- Checksum flag
 - False flag
 - Corrupted flag
- Incomplete message
- Failed checksum
- Missing BOM flag
- Missing EOM flag
- Missing frame

The protocol features that help mitigate these issues are flags, checksums, ack/nack support, and sequence number support. These mitigations are intended to provide some error resistance, not make it error proof. The methods used provide for some level of error detection only. If errors are found the frame would be rejected. Overcoming rejected frames requires retransmission.

If an ack is requested, and not using a sequence number, the sender should wait for a reply before sending the next message. Valid replies are ack or nack. If the sequence number is used, the error response should include the sequence number of the originating message.

An error is declared if the frame length exceeds the maximum number of bytes, or if another beginning of frame flag is detected without an end of message flag.

Other means to manage errors are to use elements with this protocol but with fixed header elements and fixed message lengths. This would permit frame position formatting. Another approach is to CRC the data length information within the frame. This can be done by using custom flags or two messages, first one with the length of the following message.

There are a limited number checksum methods supported with varying abilities and complexity. Since this protocol is targeted to small embedded systems not all possible methods are used. When using a checksum, the message length should be limited to the maximum size specified by the header data length element. This helps stay more within the capabilities on the checksum. It is recommended that a given implementation pick one checksum method.

The CRC polynomials selected here provide the best Hamming distance based on the message size limits. All polynomials here have a Hamming distance of at least 4. While these are not "standard" they have the best properties for what this is intended to do. See References below for sources of CRC information.

The 8-bit CRC is suitable for implementations with very short messages of 10-bytes or less. The 12-bit CRC is suitable for about 254 bytes, which works well with messages that limit the data to 128 bytes and any combination of header elements. The 16-bit CRC is suitable for checksum data being less than about 1K-bytes with Hamming distance of 4.

The missing frame scenario is best detected using the sequence number. This will not detect multiples of the maximum sequence number. The larger the sequence number the more likely missing messages can be detected. In the case of large message loss, the number of messages lost cannot be easily determined.

8-bit Modulo Checksum

- Checksum = (sum of bytes)%256
- Checksum will be last 8-bits.

16-bit Modulo Checksum

- Checksum = (sum of bytes)%65536
- Checksum will be last 16-bits.

Fletcher-16 Checksum

- C0_initial = 0
- C1_initial = 0
- CB0 = 255 - ((C0 + C1) mod 255)
- CB1 = 255 - ((C0 + CB0) mod 255)
- Checksum will be 16-bits.

CRC-8

Koopman 0x97 ; 0x12F explicit.

Hex:	1				2				F
Bits:	08	07	06	05	04	03	02	01	00
Binary:	1	0	0	1	0	1	1	1	1

- Poly = $x^8+x^5+x^3+x^2+x+1$
- Initialize with 0x00 .
- Result is 8-bits.
- Performance:
 - 10 bytes - Hamming distance = 3
 - 10 bytes - Hamming distance = 4
 - 0 bytes - Hamming distance = 5
- Decent Hamming distance only for very small number of bytes.

CRC-12 6sub8

Koopman 0x8F3 ; 0x11E7 explicit.

Hex:	1				1				E				7
Bits:	12	11	10	09	08	07	06	05	04	03	02	01	00
Binary:	1	0	0	0	1	1	1	1	0	0	1	1	1

- Poly = $x^{12}+x^8+x^7+x^6+x^5+x^2+x+1$
- Initialize with 0x00
- Result is 12-bits.
- Performance:
 - 254 bytes - Hamming distance = 3
 - 254 bytes - Hamming distance = 4
 - 4 bytes - Hamming distance = 5
 - 4 bytes - Hamming distance = 6
- Decent Hamming distance for a reasonable number of bytes.

CRC-16 6sub8

Koopman 0x808D ; 0x1011B explicit.

Hex:	1				0					1				1					B
Bits:	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00		
Binary:	1	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	1		

- Poly = $x^{16}+x^8+x^4+x^3+x+1$
- Initialize with 0x00
- Result is 16-bits.
- Performance:
 - 3580 bytes - Hamming distance = 3
 - 3580 bytes - Hamming distance = 4
 - 12 bytes - Hamming distance = 5
 - 12 bytes - Hamming distance = 6
- Decent Hamming distance for large number of bytes.

CRC-16 M17

Koopman 0xAC9A ; 0x15935 explicit.

Hex:	1				5					9				3					5
Bits:	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00		
Binary:	1	0	1	0	1	1	0	0	1	0	0	1	1	0	1	0	1		

- Poly = $x^{16}+x^{14}+x^{12}+x^{11}+x^8+x^5+x^4+x^2+1$
- Initialize with 0xFFFF
- Result is 16-bits.
- Performance:
 - 30 bytes - Hamming distance = 3
 - 30 bytes - Hamming distance = 4
 - 30 bytes - Hamming distance = 5
 - 4 bytes - Hamming distance = 6
- Good Hamming distance for small number of bytes.

DATA

In general, all payload data is intended to be ASCII characters. This applies to data types (`0xF4` , `0xF6` , `0xFD`). If no data type is called out then the data payload is assumed to be all ASCII characters.

Limited support for Binary payload characters is available with the `0xF8` data type flag. This must be used with the data length feature to avoid confusion with control header features. Alternatively binary data can be sent in the Hex-ASCII format.

The custom data type flag of `0xF7` allows the user to define a custom data payload. The custom format still needs to comply with the protocol rules. If custom data is within the ASCII character set then there are few issues. If the custom data is Binary, then the data length must be specified.

In general, for ASCII data payloads, the application can encode whatever it wants as long as msb=0. Below are a few different methods to encode different sizes of data, such as from an ADC. The protocol has some optional flags which can be used to explicitly identify different data formats to ease decoding.

Other schemes are possible and should be defined by the specific application.

ASCII Data (msb=0)

Normal printable ASCII text characters are from `0x20` to `0x7E` .

```
0 A A A A A A A
- =====
| |
| +----- A = 7-bit ASCII data
+----- 0 = Data in "ASCII"
```

Encoding in Hexadecimal-ASCII (Hex-ASCII)

Binary data can be contained in ASCII data by using Hex-ASCII across two bytes. Data encoded in this manner is preceded with the `0xFD` flag. There is no means to distinguish the first and second parts other than keeping track of pairs. This results in printable ASCII characters. It is up to the user if letters need to be upper case, lower case, or can be both.

```
# = Bin  = Hex  = ASCII
-----
0 = 0000 = 0x30 = "0"
1 = 0001 = 0x31 = "1"
2 = 0010 = 0x32 = "2"
3 = 0011 = 0x33 = "3"
4 = 0100 = 0x34 = "4"
5 = 0101 = 0x35 = "5"
6 = 0110 = 0x36 = "6"
7 = 0111 = 0x37 = "7"
8 = 1000 = 0x38 = "8"   Hex  = ASCII
9 = 1001 = 0x39 = "9"   -----
A = 1010 = 0x41 = "A" = 0x61 = "a"
B = 1011 = 0x42 = "B" = 0x62 = "b"
C = 1100 = 0x43 = "C" = 0x63 = "c"
D = 1101 = 0x44 = "D" = 0x64 = "d"
E = 1110 = 0x45 = "E" = 0x65 = "e"
F = 1111 = 0x46 = "F" = 0x66 = "f"
```

Examples:

- Sending `0xA5` in Hex-ASCII becomes two bytes of `0x41 0x35`.
- Characters `0x41` and `0x35` are rendered on a terminal as "A", "5".
- Send with most significant byte first.

Nibble Data Encoding in ASCII

The method described below encodes multiples of 4-bits of binary data across multiple ASCII bytes. Data encoded in this manner is preceded with the `0xF4` flag. This is not very compact, but is useful for sending binary data with less processing overhead compared encoding into Hex-ASCII. Data is in lsb 4-bits of a byte. The msb 4-bits can be all zeros. Or 3-bits can be used as a countdown for the number of nibbles to be concatenated. The nibble order uses Big-endianness. The highest order nibble will have the largest countdown number. This effectively tells how far to left shift the nibbles when decoding before OR'ing together. This method permits up to 8 nibbles to be grouped (32-bits). This takes the same amount of bytes to encode data as Hex-ASCII. Note that not all resulting ASCII characters will be printable as in Hex-ASCII. Checksum data (after `0xFC` flag) uses the nibble data with countdown.

Nibble data:

```
0 0 0 0 x x x x
----- =====
```

0 c c c x x x x, where 'ccc' is a countdown to last nibble in the group

```
- ~~~~~ =====
```

|

+-- Always zero

Examples:

- 8-bits without countdown `0x34` = `0x03, 0x04` (2-bytes)
- 8-bits with countdown `0x34` = `0x13, 0x04` (2-bytes)
- 12-bits with countdown `0x234` = `0x22, 0x13, 0x04` (3-bytes)
- 16-bits with countdown `0x1234` = `0x31, 0x22, 0x13, 0x04` (4-bytes)
- 32-bits `0x12345678` = `0x71, 0x62, 0x53, 0x44, 0x35, 0x26, 0x17, 0x08`

Example conversion:

SUM of $((16^{(\text{upper-nibble})}) \times \text{lower-nibble})$

4660 = 16-bits ``0x1234`` = ``0x31, 0x22, 0x13, 0x04`` (4-bytes)

0x31	0x22	0x13	0x04	(4-bytes of data)
0x 3 1	0x 2 2	0x 1 3	0x 0 4	(separate nibbles)
$(16^3) \times 1 +$	$(16^2) \times 2 +$	$(16^1) \times 3 +$	$(16^0) \times 4$	
4096 x 1 +	256 x 4 +	16 x 3 +	1 x 4	
4096 +	512 +	48 +	4	= 4660

12-bit Data Encoding ASCII

The method described below encodes 12-bits of binary data across two ASCII bytes. Data encoded in this manner is preceded with the `0xF6` flag. This is useful for sending data from a 10 or 12-bit ADC. Or sending 8-bit ADC data with a 4-bit channel prefix (i.e. MUX channel). Data is sent in two bytes. Note that not all resulting ASCII characters will be printable as in Hex-ASCII. 12-bit binary `Bxxxxxxyyyyyy` needs to be split in half. The most significant bits are 0. To keep the pairs together, bit-7 of the first byte is 1 and 0 for the second byte.

```
12-bit data:                10-bit data:
0 1 x x x x x x      0 0 y y y y y y      0 1 0 0 x x x x      0 0 y y y y y y
--- =====          --- =====          --- ~~~ =====          --- =====
MSB                  LSB                  MSB                  LSB

8-bit data with 4-bit channel prefix:
0 1 c c c c d d      0 0 d d d d d d
--- ~~~~~ ==          --- =====
MSB                  LSB
```

Example:

Sending 12-bit of data `0x4A5` in 12-bit ASCII goes as follows:

`0x4A5 = 0100 1010 0101 = 010010, 010101`

`010010 = MSB = 0101 0010 = 0x72 (first byte)`
===== ---= =====

`010101 = LSB = 0001 0101 = 0x15 (second byte)`
===== ---= =====

Results in two bytes of `0x72, 0x15`.

PROTOCOL TYPE

There are a considerable number of permutations of KEN protocol implementations. The goal is to only use what is needed in a design. How do you compare different implementations? The protocol type generates a succinct uniform description. This is a form a configuration control. It can help to determine if two systems might be compatible, or how close they might be. This is useful information to be included in source code comments.

The type, or "flavor", of the KEN protocol can be expressed with four fields separated by dashes '-'. Each field is a two digit hexadecimal number. This is prefaced with the protocol version. See below for definitions.

KEN Protocol Type

```
KEN-A:7F-FF-F4-3F <--- All features supported or used!
  -- -- -- --
  |  |  |  |
  |  |  | +------ Optional flags supported
  |  | +------ Data types supported
  | +------ Extended features
  +------ Features used (same as 0xF2 flag data)
```

Features Used (set bits to '1' if used)

This is the same data that would follow the `0xF2` flag.

```
msb          lsb
0 x x x x    x x x x
- - - -      - - - -
| | | |      | | | |
| | | |      | | | +-- checksum (0x8n)
| | | |      | | +---- sequence number (0x9n)
| | | |      | +----- from address (0xAn)
| | | |      +----- to address (0xBn)
| | | |
| | | +----- connection control (0xCn)
| | +----- data length (0xDn)
| +----- error control (0xEn)
+----- always '0'
```

Extended Features (set bits to '1' if used)

Note that the feature itself must be marked as being used before it is valid to have an extended feature. The only exception is the custom flag (0xFF).

```
msb          lsb
x x x x    x x x x
- - - -    - - - -
| | | |    | | | |
| | | |    | | | +-- custom checksum type (0x8F)
| | | |    | | +---- custom sequence number (0x9F)
| | | |    | +----- extended from address (0xAF)
| | | |    +----- extended to address (0xBF)
| | | |
| | | +----- custom connection control (0xCF)
| | +----- extended data length (0xDF)
| +----- custom error control (0xEF)
+----- custom flag (0xFF)
```

Data Types Supported (set bits to '1' if used)

ASCII data is the default and most common.

```
msb          lsb
x x x x    x 0 0 0
- - - -    - - - -
| | | |    | | | |
| | | |    | | | +-- always '0' (reserved)
| | | |    | | +---- always '0' (reserved)
| | | |    | +----- always '0' (reserved)
| | | |    +----- Binary data type (0xF8)
| | | |
| | | +----- nibble data type (0xF4)
| | +----- ASCII data type (default) (0xFD)
| +----- 12-bit data type (0xF6)
+----- custom data type (0xF7)
```


Optional Flags Supported (set bits to '1' if used)

If these are supported, it does not necessarily mean they will be used.

```
msb          lsb
0 0 x x      x x x x
- - - -      - - - -
| | | |      | | | |
| | | |      | | | +-- null (0xF0)
| | | |      | | +---- feature request (0xF1)
| | | |      | +----- feature list automatically provided (0xF2)
| | | |      +----- sync (0xF3)
| | | |
| | | +----- Sub-frame numbering (0xF9)
| | +----- ping & pong (0xF5, 0xFA)
| +----- always '0' (reserved)
+----- always '0' (reserved)
```

Example Types

Below are some examples of different protocol types. Note that there are numerous permutations. The examples below are from simple to more complex.

Type `KEN-A:00-00-20-00`

- ASCII data type flag, beginning, and end of frame flags.
- `FB FD [ASCII message] FE`

Type `KEN-A:0C-00-20-00`

- From/to addressing (simple only)
- No extended features used.
- ASCII data type.
- `FB An Bn FD [ASCII message] FE`

Type `KEN-A:2D-AC-20-00`

- Checksum, from/to addressing, data length.
- Custom flag, extended addressing, extended data length.
- ASCII data type.
- `FB 8n AF N BF N DF N FF N FD [ASCII message] FC [checksum] FE`

USAGE TIPS

This is intended for point designs, not general compatibility with other KEN protocol designs. Implementing all the features of this protocol for an application is not envisioned. This is intended to be a toolkit to allow for a rich number of features. In general it is recommended to only use the minimal features necessary for a given application.

If extended bytes, custom features, and the checksum are not used, the only ASCII bytes will be the data payload. This makes parsing simpler as well as overhead smaller.

If using the extended address bytes (i.e. `0xAF 0xN`) the address can be a single ASCII character. There are no provisions for automatically assigning addresses. It is not required to use both the from (`0xAn`) and to (`0xBn`) features to use addressing. In a network with one master, the to address can be used without the from, or visa versa. The addressing can mix the basic and extended versions in a message (i.e. `0xAn` and `0xBF 0xN`).

In a single point-to-point channel application (i.e. RS-232), the addressing could be repurposed as a means to identify message content. For, example, `0xAF "T"` could mean telemetry and `0xAF "S"` could mean settings. Or it could represent a data channel.

Most features that have a `'0'` as the second nibble usually means that feature is not being used or not supported. Control feature codes that are reserved should not be used.

Most features that have a `'F'` as the second nibble usually means that feature support a user customization. Typically the additional data is in the following ASCII byte. It is limited to only one byte. This creates opportunities for user customization without changing the protocol. The same parsing function can be used on all msb=1 bytes to find additional bytes, if used. While this allows for and adds flexibility, it does grow the header size. Use of custom features should be carefully considered. Any custom features used are considered application specific and thus not documented here.

Only the `0xFn` flags for data type can be followed by a variable number of bytes. The `0xF2` flag has one follow on ASCII byte and `0xF9` has two ASCII bytes. ASCII data is always followed by a byte with msb=1. This could be end of frame flag, or checksum data flag. This makes it relatively easy to find the end of a data string without needing data length information.

This protocol is primarily designed for ASCII data (or ASCII encoded data) payloads. There is a provision for binary, but it comes with limitations. If using the `0xF8` flag for binary data, the data length header is required. The data can only be recovered based on that length information. This also limits the length of binary data to the data length range.

The ping (0xF5) and pong (0xFA) can be used to test a link before sending messages. This could be most useful with RF links to confirm the link is up and working.

Adding error checking requires that an error disposition plan be developed. What happens next? This is added complexity that should be considered carefully. This protocol does not define error processing rules. It does provide some tools to help with implementing rules.

Checksum should be used where having bad data is worse than missing or delayed data. The checksum will not aid in correcting the message. The options are to just loose the data or ask for a resend. Resending requires more complexity and more channel bandwidth. Checksum usage should also be based on the quality of the data channel and its bit error rate. Also if the messages are going to be encapsulated in packets with error handling (i.e. TCP/IP), the checksum could be redundant and just more unnecessary overhead.

Checksum starts at beginning of frame. The checksum type header element (0x8n) should be the first character after the frame beginning (0xFB) flag. The characters are covered from, and including, the checksum header (0x8n) to and including the checksum flag (0xFC). The checksum flag is usually after the data to cover as much of the frame as possible. It could be placed before the data to effectively checksum only the frame header, and not the data payload. The 0x8n should trigger the start of the checking and the 0xFC triggers the end.

Adding connected links with error control may require some additional rules will need to be defined. These include time out periods and retry counts. If also using sequence numbers, then more rules need to be defined for managing missing frames. Since individual applications can vary, this protocol does not attempt to define these details.

The sub-frame number (0xF9) flag feature can be used to break up a larger message into smaller pieces. This could be useful if the channel has a message size limitation (i.e. 32-byte limit for an nRF24L01+). The full message can be re-assembled based in the frame number and total number of expected frames. The cost is three additional bytes of overhead.

The sub-frame number is not to be confused with the sequence number option (0x9n) header control feature. The latter is intended to help detect a missed or lost message. This is useful for potentially noisy channels where knowledge of lost messages is important. The counter simply increments and then rolls over. The roll over can be set to less than the maximum allowed. This does support a custom counter for a maximum of 127. What to do if lost messages are detected are up to the developer.

Any application should specify the Protocol Type prominently in the code comments. Be sure to update this when making any application revisions.

A hexadecimal viewer is your friend in debugging messages. This protocol was designed to make it easier to decipher messages in this manner.

COMPLEXITY HIERARCHY

The list below defines a general hierarchy from simple to most complex features. This is a generalized progression in which features would most likely be added.

1. Basic flags (0xFB , 0xFD , 0xFE). Simplest.
 2. Adding addressing (0xA_n , 0xB_n).
 3. Adding data length (0xD_n).
 4. Using different data types (0xF4 , 0xF6 , 0xF7 , 0xF8).
 5. Adding checksums (0x8_n).
 6. Adding sequence numbers (0x9_n) and/or sub-frame numbers (0xF9).
 7. Adding connections (0xC_n).
 8. Adding error control (0xE_n). Most complex.
-

EXAMPLE MESSAGES

Below are a number of example messages demonstrating the various features. The features are demonstrated individually and in combination. Not all permutations are shown. This is included for illustrative purposes only, and not a prescription of how to use this protocol. Most data shown is in hexadecimal with data flowing left (left side are the first bits/bytes).

Frame highlights:

- ++ Frame flags
- ** Header control elements
- -- Flag or header ASCII component
- == Frame payload data
 - 31 = Hex 0x31
==
 - 1 = ASCII "1"
=
 - 10 01 = Nibble format, 0x01
=====
- ^^ Binary data
- ~~ Data covered by checksum

Flag Use Examples

Null message. Note the frame beginning frame end characters. These are required for all frames messages.

```
FB F0 FE
++ ++ ++
```

Sync with null message. The sync character is optional. The number of times the character need to sent is not defined by this protocol.

```
F3 F3 F3 FB F0 FE
++ ++ ++ ++ ++ ++
```

Request features used.

```
FB F1 FE
++ ++ ++
```

Report features used (none).

```
FB F2 00 FE
++ ++ == ++
```

Ping request message.

```
FB F5 FE
++ ++ ++
```

Pong response message.

```
FB FA FE
++ ++ ++
```

Sub-frame numbering. Frame 1 of 3.

```
FB F9 01 03 FD 33 34 FE
++ ++ - - - - ++ == == ++
```

Sub-frame numbering. Frame 1 of unknown set size.

```
FB F9 01 00 FD 33 34 FE
++ ++ ----- ++ == == ++
```

Using sub-frames numbers to break up a larger message across frames. The data `Hello world!` is sent across two frames. Sub-frame numbering allows proper ordering and knowing that all sub-frames have been received. The application layer would assemble the complete message.

```
FB F9 01 02 FD H e l l o   FE
++ ++ ----- ++ = = = = = ++
```

```
FB F9 02 02 FD w o r l d ! FE
++ ++ ----- ++ = = = = = ++
```

User defined flag `0x01` , after `0xFF` . Only one byte allowed after `0xFF` .

```
FB FF 01 FD 33 34 FE
++ ++ -- ++ == == ++
```

Data Examples

Twelve ASCII data bytes with minimum overhead. Most minimalist case. Note: Data length is unrestricted if no length is specified. 2-bytes of overhead.

```
FB 4B 45 4E 20 50 52 4F 54 4F 43 4F 4C FE
++ == == == == == == == == == == == ++
```

Twelve ASCII data bytes with minimum overhead and data flag. Note: Data length is unrestricted if no length is specified. 3-bytes of overhead.

```
FB FD 4B 45 4E 20 50 52 4F 54 4F 43 4F 4C FE
++ ++ == == == == == == == == == == ++
```

Four data bytes (0x12 , 0x34 , 0x56 , 0x78) with nibble encoding. Countdown used to pair the nibbles. Data must be pairs.

```
FB F4 11 02 13 04 15 06 17 08 FE
++ ++ ===== ++
```

32-bits (0x12345678) with nibble encoding. Countdown used to group the nibbles.

```
FB F4 71 62 53 44 35 26 17 08 FE
++ ++ ===== ++
```

Two 32-bit double words (0x12345678 , 0x87654321) with nibble encoding. Countdown used to group the nibbles.

```
FB F4 71 62 53 44 35 26 17 08 78 67 56 45 34 23 12 01 FE
++ ++ ===== ++
```

One 12-bit data (0x4A5) using 6-bit data. Data must be pairs.

```
FB F6 52 15 FE
++ ++ ===== ++
```

One 10-bit (0x234) using 6-bit data. Data must be in pairs.

```
FB F6 78 34 FE
++ ++ ===== ++
```

Two bytes of binary data using the 0xF8 binary data type flag. The data length is required.

```
FB D2 F8 81 85 FE
++ ** ++ ^^ ^^ ++
```

Twenty seven ASCII data bytes with minimum overhead and data flag. 3-bytes of overhead.

```
FB FD KEN Protocol - Hello World! FE
++ ++ ===== ++
```

Twenty four bytes of CSV ASCII data (,12.41,12.03,05.01,03.33) with minimum overhead and data flag. 3-bytes of overhead.

```
FB FD ,12.41,12.03,05.01,03.33 FE
++ ++ ===== ++
```

Forty eight bytes of CSV ASCII data with extended from addressing and split across two frames using the frame counter with from address. Each frame is 32-bytes with 8-bytes of overhead per frame.

```
FB AF 31 F9 01 02 FD ,12.41,12.03,05.01,03.33 FE
++ ** -- ++ ----- ++ ===== ++
```

```
FB AF 31 F9 02 02 FD ,02.21,01.25,05.01,03.33 FE
++ ** -- ++ ----- ++ ===== ++
```

User defined data format 0x01 , 0x02 after 0xF7 flag. Multiple bytes allowed after 0xF7 .

```
FB F7 01 02 FE
++ ++ ----- ++
```


Addressing Examples

Sixteen ASCII data bytes with simple addressing (from node 1 to 2).

```
FB A1 B2 FD 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F FE
++ ** ** ++ == == == == == == == == == == == ++
```

Sixteen ASCII data bytes with extended addressing (from node 1 to 2).

```
FB AF 01 BF 02 FD 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F FE
++ ** - - ** - - ++ == == == == == == == == == == == ++
```

Ping with addressing and no data (from node 1 to 2).

```
FB A1 B2 F5 FE
++ ** ** ++ ++
```

Pong with addressing and no data (from node 2 to 1).

```
FB A2 B1 FA FE
++ ** ** ++ ++
```

Short message only using the to address (assuming from a master).

```
FB B2 FD 40 41 42 43 FE
++ ** ++ == == == == ++
```

Short message only using the from address (assuming to a master).

```
FB A2 FD 40 41 42 43 FE
++ ** ++ == == == == ++
```

Using mixed address sizes.

```
FB A0 BF 32 FD 40 41 42 43 FE
++ ** ** - - ++ == == == == ++
```

Data Length Examples

Two ASCII data bytes basic data length, data flag.

```
FB D2 FD 7A 7B FE
++ ** ++ ===== ++
```

Fifteen ASCII data bytes with basic data length. Note: Single data length control byte limits data length to 15-bytes.

```
FB DE FD 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D FE
++ ** ++ == == == == == == == == == == == == ++
```

Fifteen ASCII data bytes with extended data length. Note: Single data length control byte limits data length to 15-bytes.

```
FB DF 0F FD 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E FE
++ ** -- ++ == == == == == == == == == == == ++
```

Sixteen ASCII data bytes with extended data length. Note: Extended data length control limits data to 128 bytes.

```
FB DF 10 FD 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F FE
++ ** -- ++ == == == == == == == == == == == ++
```

Seventeen ASCII data bytes with extended length and data flag.

```
FB DF 11 FD 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 FE
++ ** -- ++ == == == == == == == == == == == ++
```

Eight Binary bytes with addressing, length (required for binary data), and data flag (required for binary data).

```
FB D7 F8 47 48 49 4A 4B 4C 4D FE
++ ** ++ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ++
      Binary data
```

Checksum Examples

Twelve ASCII data bytes with CRC-16 checksum. Note: Data length is unrestricted if no length is specified. 9-bytes of overhead.

```
FB 8A FD 4B 45 4E 20 50 52 4F 54 4F 43 4F 4C FC 31 22 13 04 FE
++ ** ++ == == == == == == == == == == ++ ----- ++
~~~~~
```

Two ASCII data bytes with addressing, checksum, length, data flag. 10-bytes of overhead.

```
FB 81 A1 B2 D2 FD 7A 7B FC 11 02 FE
++ ** ** ** ** ++ == == ++ ----- ++
~~~~~
```

Two ASCII data bytes with addressing, length, ack request, and CRC-16 checksum. 13-bytes of overhead.

```
FB 8A A1 B2 D2 EA FD 7A 7B FC 31 22 13 04 FE
++ ** ** ** ** ** ** ++ == == ++ ----- ++
~~~~~
```

Two ASCII data bytes with addressing, checksum, length, data flag. CRC for header only, data follows checksum. 10-bytes of overhead.

```
FB 81 A1 B2 D1 FC 11 02 FD 7A 7B FE
++ ** ** ** * ++ ----- ++ == == ++
~~~~~
```

Custom checksum covering the header only. Data follows checksum. Two ASCII data bytes with addressing, checksum, length, data flag. 10-bytes of overhead.

```
FB 8F 01 A1 B2 D1 FC 11 02 FD 7A 7B FE
++ ** - ** ** * ++ ----- ++ == == ++
~~~~~
```

Error Control Examples

One ASCII data byte with addressing and no error control actions requested.

```
FB A1 B2 E0 FD 7A FE
++ ** ** ** ++ == ++
```

One ASCII data byte with addressing and ack/nack request.

```
FB A1 B2 E5 FD 7A FE
++ ** ** ** ++ == ++
```

Response with ack.

```
FB A2 B1 EA FE
++ ** ** ** ++
```

Response with nack.

```
FB A2 B1 EE FE
++ ** ** ** ++
```

Response reporting checksum error (assuming a received message had a checksum).

```
FB A2 B1 EC FE
++ ** ** ** ++
```

Response with user custom error code (`0x01`).

```
FB A2 B1 EF 01 FE
++ ** ** ** - - ++
```

Connection Examples

Connect request with addressing (node 1 asking to connect to 2).

```
FB A1 B2 CA FE
++ ** ** ** ++
```

Connect granted with addressing (node 2 granting connect to 1).

```
FB A2 B1 CC FE
++ ** ** ** ++
```

Connected message with sixteen ASCII data bytes with simple addressing (from node 1 to 2).

```
FB A1 B2 CC FD 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F FE
++ ** ** ** ++ == == == == == == == == == == == == == ++
```

Connection error or rejected (node 2 rejecting 1).

```
FB A2 B1 CE FE
++ ** ** ** ++
```

Connect break request with addressing (node 1 asking to connect to 2).

```
FB A1 B2 CB FE
++ ** ** ** ++
```

Connection disconnected (node 2 disconnecting from 1).

```
FB A2 B1 CD FE
++ ** ** ** ++
```

Connection custom user defined byte `0x01` .

```
FB A2 B1 CF 01 FE
++ ** ** ** - - ++
```

Sequence Number Examples

Sequence number, addressing, length, CRC-8 checksum, ack request.

```
FB 88 91 A1 B2 D2 E5 FD 7A 7B FC 11 02 FE
++ ** ** ** ** ** ** ** ** ** ++ == == ++ - - - - ++
~~~~~
```

Reply with sequence number, acknowledgement, no data, and CRC-8 checksum.

```
FB 88 91 A2 B1 EA FC 11 02 FE
++ ** ** ** ** ** ** ++ - - - - ++
~~~~~
```

Sequence number, addressing, error control acknowledgement, and no data.

```
FB 91 A2 B1 EA FE
++ ** ** ** ** ++
```

Sequence number available but not used, addressing, error control acknowledgement, and no data.

```
FB 90 A2 B1 EA FE
++ ** ** ** ++
```

Sequence number custom byte `0x01` .

```
FB 9F 01 A2 B1 EA FE
++ ** - - ** ** ++
```

Compound Examples

Checksum, simple addressing. 9-bytes of overhead.

```
FB 88 A1 B2 FD 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E FC 13 04 FE
++ ** ** ** ++ == == == == == == == == == ++ ===== ++
~~~~~
```

Simple addressing, basic data length of 15 (max) ASCII data bytes. 6-bytes of overhead.

```
FB A1 B2 DE FD 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E FE
++ ** ** ** ++ == == == == == == == == == ++
```

Simple addressing, extended data length, 16 ASCII data bytes. 7-bytes of overhead.

```
FB A1 B2 DF 0F FD 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F FE
++ ** ** ** - - ++ == == == == == == == == == ++
```

Extended addressing, extended data length, 16 ASCII data bytes. 9-bytes of overhead.

```
FB AF 01 BF 02 DF 10 FD 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F FE
++ ** - - ** - - ++ == == == == == == == == == ++
```

Checksum, simple addressing, basic data length, 14 ASCII data bytes. 10-bytes of overhead.

```
FB 81 A1 B2 DE FD 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E FC 11 02 FE
++ ** ** ** ** ++ == == == == == == == == == ++ ----- ++
~~~~~
```

Three nibbles bytes with addressing, length (3) and data flag.

```
FB A1 B2 D3 F4 20 11 02 FE
++ ** ** ** ++ ===== ++
      Nibbles
```

Fully Loaded Examples

CRC-8 checksum, sequence number 1, addressing from 1 to 2, connect request, 2-byte data length (1=2-bytes), ack requested, feature list (all), frame 1 of 2, user defined flag (0x01), data (2-bytes ASCII), checksum. 20-bytes of overhead.

```
FB 88 90 A1 B2 CA D2 E5 F2 7F F9 01 02 FF 01 FD 31 32 FC 13 04 FE
++ ** ** ** ** ** ** ** ** ** ** ++ -- ++ -- -- ++ -- ++ == == ++ ----- ++
~~~~~
```

CRC-8 checksum, sequence number 1, extended addressing from 1 to 2, connect request, extended 1-byte data length, ack requested, feature list (all), frame 1 of 2, user defined flag (0x01), data (1-byte ASCII), checksum. 23-bytes of overhead.

```
FB 88 90 AF 01 BF 02 CA DF 01 E5 F2 7F F9 01 02 FF 01 FD 31 FC 13 04 FE
++ ** ** ** -- ** -- ** ** -- ** ++ -- ++ -- -- ++ -- ++ == ++ ----- ++
~~~~~
```

Telemetry Examples

Message that fits within the single packet size of 32-bytes for nRF24L01+ applications. Eleven bytes reserved for frame overhead and 21-bytes for data payload. Label can be up to 10 ASCII characters and units up to 3.

Frame with simple 1-byte addressing.

```
FB 9# A# F9 01 01 FF ## FD <Label...>, +##.##, UUU FE
++ ** ** ++ -- -- ++ -- ++ ===== ++
```

Frame with extended 2-byte addressing.

```
FB 9# AF ## F9 01 01 FF ## FD <Label...>, +##.##, UUU FE
++ ** ** -- ++ -- -- ++ -- ++ ===== ++
```

Example Garage temperature with 17-bytes payload data, 27-byte frame.

```
FB 91 A2 F9 03 03 FF 11 FD Garage T, +25.00, C FE
++ ** ** ** -- -- ++ -- ++ ===== ++
```

REFERENCES

1. Serial Input Basics - updated
<https://forum.arduino.cc/index.php?topic=396450>
2. Checksum for Serial communication
<https://forum.arduino.cc/index.php?topic=347504.0>
3. Create a short checksum for a String
<https://forum.arduino.cc/index.php?topic=380192.0>
4. What's the best CRC polynomial to use?
<https://betterembsw.blogspot.com/2010/05/whats-best-crc-polynomial-to-use.html>
5. Best CRC Polynomials
<https://users.ece.cmu.edu/~koopman/crc/>
6. Checksum Calculator
<https://crccalc.com/>
7. Online Checksum Calculator
<https://www.scadacore.com/tools/programming-calculators/online-checksum-calculator/>
8. Fletcher's checksum
https://en.wikipedia.org/wiki/Fletcher%27s_checksum
9. Fletcher 16 Checksum generator
<https://ozeki.hu/index.php?owpn=1613&fletcher=&submit=Submit>
[https://gchq.github.io/CyberChef/#recipe=Fletcher-16_Checksum\(\)&input=VGZzdA](https://gchq.github.io/CyberChef/#recipe=Fletcher-16_Checksum()&input=VGZzdA)

Older Ideas

Notes on older ideas that have been retired, but want to retain for possible reconsideration at a later date.

- Data length was the number plus one. Allowed for 15 bytes in the basic version and 128 bytes in the extended version. Problem is that reading the frames the number is always off by one. Just makes for easier human readability to have the number match the number of bytes. Keep it simple.
- Did not allow binary data since the msb is used as a marker for control features. This is easy to test for in software. Determined binary data could be supported only if the data length feature is used. Redefined the `0xF8` flag from Hex-ASCII (which is really just ASCII) to binary data (i.e. 8-bits). Added notes that `0xF8` requires `0xDn`.
- No longer allow mixed data types in a message (i.e. ASCII and nibble) in the same frame. The corresponding flag would precede each data type. This complicated the data length calculation since the flags were to be ignored. After including a binary data option the data length becomes critical. Following the KISS mode, determined that it was just simpler to not mix data types in a frame.
- Thought about using the basic data length to indicate discrete data block sizes (i.e. `0xD4` = 32 bytes). This would allow for a greater range of data without using the extended data length feature. It could also allow for much greater data sizes (i.e. 256 or 512 bytes) beyond what even the extended data length supports. The drawbacks were that it is more cryptic and data padding could often be needed. Decided readability of the data length fields were more important than having larger data block sizes.
- Decided to not include a CRC-32, which could support larger frame sizes. Want this to work with small systems that often don't have a lot of memory or power. The CRC's selected cover reasonable size frames. Also CRC's are not always needed if the frame is being transported via a method that does EDC.
- Extended feature bytes are intentionally limited to one immediately following byte. Having an increased or variable number just complicates the software.

END OF DOCUMENT
