

KEN-C PROTOCOL

By: Ken McCaughey

On: 2025-02-08

Ver 1.0.0

Ken's Electronic Network (KEN) Compact Protocol

This protocol is derived from the original Ken's Electronic Network (KEN) Protocol. It is a scaled down and a very Compact version that trades features and some human readability for mostly fixed overhead. While this protocol borrows features from the original KEN-A and KEN-B, is not compatible with it (KEN-A != KEN-B != KEN-C).


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) Compact Protocol (KEN-C), 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.
- Derived from the original Ken Electronic Network (KEN) Protocol.
- Borrows ideas from MIDI, BiSync, and S.N.A.P. protocols.
- Features very compact header elements.
 - Trades features and flexibility for link overhead efficiency.
 - Header components are fixed.
- All control features in a header are position dependent.
 - Nibble based with limited capabilities.
- Message sizes scalable based on complexity and options used.
 - Can be as short as 6-bytes.
 - One data byte.
 - Maximum frame length limited to 127 bytes.
 - Using all header features, 7-bytes overhead.
 - Maximum payload data of 120 bytes with checksum.
- Supports optional addressing.
 - Up to 15 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 15 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.
- Human readability.
- 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.
- Custom header features (see original KEN Protocol).

BASIC RULES

- Keep it simple.
 - Only use if most of the features are needed.
- All header elements are position dependent.
- If this is not a good fit, take a look at KEN-A or Ken-B.

FRAME FORMAT

All message frames begin with a Frame Length (FL) byte with msb = 1. The remaining four header control bytes use are required and the order is fixed. The FL is of the entire frame, including any checksum data. Therefore the maximum frame size is 127 bytes. Payload data can range from 0 bytes to 122, with no checksum, and 120 with a CRC-16 checksum.

Frame highlights:

- ++ Frame length (FL)
- ** Header control elements (hexadecimal)
- -- Flag or header ASCII component
- == Frame payload data
- ~~ Data covered by checksum

```
Framea Len [Header/Control Elements] [Data] [Check]
+++++++ ***** ===== -
~~~~~
```

```
1 2 3 4 5 <-- Header Bytes
| | | | |
FL 89 AB CE Fn [Data] [Check]
++ ** ** ** ** == -
| || || | | |
| || || || | | +-checksum (up to 2-bytes) #
| || || || | +-payload data
| || || || +-frame numbers
| || || |+-error control
| || | +---connection control
| || |+-to address
| || +---from address
| |+-sequence number
| +---checksum type
+--frame length (FL)
```

depends on checksum type

Frame Length (FL), Header Byte 1

This required feature specifies the total number byte in the frame, including the required header and checksum, if used. Thus a minimum frame length is 5 bytes and up to 127.

Data payload size = 127 - 5 header bytes - checksum bytes (if used)

The msb of the FL is always set to 1. The remaining 7-bit define the length of the entire frame.

```
`0x80` || nn = 05 to 7F = Length is 5 to 127.
```

Examples:

- 5-byte data length = 0x85
- 17-bytes data length = 0x91
- 127-bytes data length = 0xFF

Checksum Type, Header Byte 2, Upper Nibble

This feature specifies the type of checksum if one is being used. This is optional. The value is in the upper nibble of the header byte after the frame length.

The checksum block includes the FL to the end of the data payload. The only supported checksums are listed below.

n = 0 = None. (no checksum data)

n = 1 = 8-bit Modulo Checksum in 1-byte.

n = 2 = 16-bit Modulo Checksum in 2-bytes.

n = 3 = Fletcher-16 checksum in 2-bytes.

n = 8 = CRC-8 in 1-byte.

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 2-bytes.

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 2-bytes.

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

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

n = F = Reserved.

Sequence Number, Header Byte 2, Lower Nibble

This optional control element is used if there is a need to keep messages in order, detect a missing message, or identify a repeated message. The value is in the lower nibble of the header byte after the frame length.

This is used at the lower link level for channel management and error control. This is not the same as the sub-frame number, 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.

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

```
n = 0, F = Reserved.
```

Address, From (ie. from A to B), Header Byte 3, Upper Nibble

Required header element located in Header byte 3 in the upper nibble.

It is recommended to set this value to no-zero to avoid too many consecutive zeros in the frame header. Can be used as an ID code for a set of hardware.

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

```
n = 1 to F = address (15).
```

Address, To (ie. from A to B), Header Byte 3, Lower Nibble

Required header element located in Header byte 3 in the lower nibble.

It is recommended to set this value to no-zero to avoid too many consecutive zeros in the frame header. Can be used as a command code for hardware.

```
n = 0 = All (Broadcast).
```

```
n = 1 to F = address (15).
```


Connection Control, Header Byte 4, Upper Nibble

Required header element located in Header byte 4 in the upper nibble. Use this feature to manage a connected link.

It is recommended to set this value to one when not doing connection control to avoid too many consecutive zeros in the frame header.

Connect request frames can include data. Multiple connected links need to use addressing to distinguish separate links.

This is the same as in the original KEN protocol except for the custom control code.

```
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 = 2 to 9, F = Reserved.
```

En - Error Control, Header Byte 4, Lower Nibble

Required header element located in Header byte 4 in the lower nibble. Use this feature if error control and management is needed.

It is recommended to set this value to one when not doing error control to avoid too many consecutive zeros in the frame header.

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

Frame Number, Header Byte 5

The frame number byte uses two nibbles. The upper nibble is the current sub-frame number. The lower nibble is the total number of sub-frames expected.

This is used for data sets that require spanning multiple messages sub-frames. Sub-frame number is limited to 4-bits (15 frames). Zeros are not allowed. If the data only needs one sub-frame then both nibbles are set to 1 (i.e. 0x11).

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 of 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 of 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

The payload data can be binary or ASCII. It is up to the application to manage the meaning of any data being used. However, the size of the data payload is limited by the required data length byte in the header.

The original KEN protocol was intended for ASCII data with limited capability for binary data. A number of different flags to identify some ASCII data format variations.

PROTOCOL TYPE

There is only one protocol type as defined by the original KEN protocol. This version has a fixed header with all defined elements always present. However, the use of the features is not required. The modified protocol type below is a mean to reflect which elements are used by the system.

There are a considerable number of permutations of KEN Compact 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 of 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 Basic protocol can be expressed with a two digit hexadecimal number. See below for definitions.

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

```
msb                                lsb
1 x 1 x      x x 1 x
- - - -      - - - -
| | | |      | | | |
| | | |      | | | +-- checksum (8n)
| | | |      | | +---- sequence number (9n), always required
| | | |      | +----- from address (An)
| | | |      +----- to address (Bn)
| | | |
| | | +----- connection control (Cn)
| | +----- frame length, always required
| +----- error control (En)
+----- frame flag (Fn), always required
```

Example Types (update)

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

Type A2

- Frame length
- Sequence number
- Frame numbers

Type AE

- Frame length
- Sequence number
- From/to addressing
- Frame numbers

Type AF

- Frame length
- Checksum
- Sequence number
- From/to addressing
- Frame numbers

USAGE TIPS

This is intended for point designs, not general compatibility with other KEN Basic protocol designs (or the original KEN protocol for that matter). To get the most out of this version, it is recommended that the application need or make use of most of the capabilities. Otherwise the fixed header overhead is not efficient. Perhaps KEN-B is a better fit.

Since the header and data payload can all be binary characters, and there is no escape character, and no unique flags; any implementation must include idle breaks between frames.

All features are nibble based, and thus are limited in capabilities. As in KEN-B, the frame contents are position dependent and rely on knowing the frame length. Since the header size is fixed, the start of the payload data is easily determined.

Note that due to the nibble wise header compression, the header can contain both binary and ASCII. The header must be parsed based on frame position. The size of the data payload is a function of the maximum frame size minus the number of header bytes and checksum method, if used.

Most features that are set to '0' means that feature is not supported, and a '1' means that feature is idle and can be ignored.

This protocol is designed for ASCII or binary data. Parsing the data is completely dependent on position in the frame. This saves overhead. The original KEN protocol utilized flags to locate frame components with an overhead penalty.

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.

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 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 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.

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

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 (FL). Simplest.
2. Adding addressing.
3. Adding checksums.
4. Adding sequence numbers and/or sub-frame numbers.
5. Adding connections.
6. Adding error control. 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:

- FL Frame Length
- ++ Frame Length
- ** Required header features
- -- Flag or header ASCII component
- == Frame payload data
 - 31 = Hex 0x31
 - ==
 - 1 = ASCII "1"
 - =
- ^^ Binary data
- ~~ Data covered by checksum

Data Examples

Minimum message with one byte of data and 5-bytes of overhead.

```
86 01 11 11 11 31
++ ** ** ** ** ==
```

Twenty nine ASCII data bytes with minimum overhead and data length. 5-bytes of overhead.

```
86 01 11 11 11 KEN B Protocol - Hello World!
++ ** ** ** ** =====
```

Forty eight bytes of CSV ASCII data with from addressing and split across two frames using the frame counter with from address. Each frame is 30-bytes with 5-bytes of overhead per frame.

```
9D 01 11 11 12 ,12.41,12.03,05.01,03.33
++ ** ** ** ** =====

9D 01 11 11 22 ,02.21,01.25,05.01,03.33
++ ** ** ** ** =====
```

Addressing Examples

Sixteen ASCII data bytes with simple addressing (from node 0xA1 to 0xB2).

```
95 01 21 11 11 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
++ ** ** ** ** == == == == == == == == == == == == == == ==
```

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

```
89 01 21 11 11 40 41 42 43
++ ** ** ** ** == == == ==
```

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

```
89 01 21 11 11 40 41 42 43
++ ** ** ** ** == == == ==
```

Data Length Examples

Two ASCII data bytes basic data length, data flag.

```
87 01 21 11 11 7A 7B
++ ** ** ** ** ==
```

Fifteen ASCII data bytes with basic data length.

```
93 01 21 11 11 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
++ ++ ** ** ** == == == == == == == == == == ==
```

Checksum Examples

Twelve ASCII data bytes with CRC-16 checksum. Data length is specified. 5-bytes of overhead.

```
93 A1 21 11 11 4B 45 4E 20 50 52 4F 54 4F 43 4F 4C 31 22
++ ** ** ** ** == == == == == == == == == == - - - - -
~~~~~
```

Two ASCII data bytes with addressing, checksum, length, data flag. 6-bytes of overhead.

```
88 81 AB 11 11 7A 7B 11
++ ** ** ** ** == == - -
~~~~~
```

Two ASCII data bytes with addressing, length, ack request, and CRC-16 checksum. 8-bytes of overhead.

```
09 A1 AB 11 11 7A 7B 31 22
++ ** ** ** ** == == - - - - -
~~~~~
```

Error Control Examples

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

```
86 01 AB 11 11 7A
++ ** ** ** ** ==
```

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

```
86 01 AB 15 11 7A
++ ** ** ** ** ==
```

Response with ack.

```
85 01 BA 1A 11
++ ** ** **
```

Response with nack.

```
85 01 BA 1E 11
++ ** ** **
```

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

```
85 01 AB 0C 11
++ ** ** **
```

Connection Examples

Connect request with addressing (node 0xA1 asking to connect to 0xB2).

```
85 01 AB A1 11
++ ** ** ** **
```

Connect granted with addressing (node 0xA2 granting connect to 0xB1).

```
85 01 BA C1 11
++ ** ** ** **
```

Connected message with sixteen ASCII data bytes with simple addressing (from node 0xA1 to 0xB2).

```
85 01 AB C1 11 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
++ ** ** ** ** == == == == == == == == == == == == == == ==
```

Connection error or rejected (node 0xA2 rejecting 0xB1).

```
85 01 BA E1 11
++ ** ** ** *
```

Connect break request with addressing (node 0xA1 asking to connect to 0xB2).

```
85 01 AB B1 11
++ ** ** ** *
```

Connection disconnected (node 0xA2 disconnecting from 0xB1).

```
85 01 BA D1 11
++ ** ** ** *
```

Sequence Number Examples

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

```
88 11 AB 15 11 7A 7B 11
++ ** ** ** ** == == --
~~~~~
```

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

```
86 11 BA 1A 11 11
++ ** ** ** ** --
~~~~~
```

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

```
05 01 BA 1A 11
++ ** ** ** *
```

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

```
05 01 BA 1A 11
++ ** ** ** *
```

Compound Examples

Checksum, simple addressing. 7-bytes of overhead.

```
95 81 AB 11 11 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 13 04
++ ** ** ** * == == == == == == == == == == == == == --
~~~~~
```

Simple addressing, extended data length, 16 ASCII data bytes. 5-bytes of overhead.

```
95 01 AB 11 11 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
++ ** ** ** * == == == == == == == == == == == == == ==
```


Fully Loaded Examples

CRC-8 checksum, sequence number 1, addressing from 1 to 2, connect request, data length, ack requested, frame 1 of 2, and checksum. 6-bytes of overhead.

```
88 11 AB A5 12 31 32 13
++ ** ** ** ** == == --
~~~~~
```

Telemetry Examples

Message that fits within the single packet size of 32-bytes for nRF24L01+ applications. Five 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.

```
FL 89 AB CD Ff <Label...>, +##.##, UUU
++ ** ** ** ** =====
```

Example Garage temperature with 17-bytes payload data, 22-byte frame.

```
96 01 A0 11 33 Garage T, +25.00, C
++ ** ** ** ** =====
```

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=VGVzdA](https://gchq.github.io/CyberChef/#recipe=Fletcher-16_Checksum()&input=VGVzdA)

Older Ideas

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

- Thought about making sure all header elements had msb=1 (binary). This would expand the header size or further reduce variation.
- Considered dropping some of the capabilities to further reduce the header size. The main candidates are the connection and error control. These both require some complex code and rule to be effective. That may run counter to the needs of a small system. Elimination would save one byte in the header. Maybe there should be a "C-" version that drops these two.
- Considered having the sub-frame and frame counters be one byte each. Seems more like header bloat and not in arraignment with being compact.
- Considered keeping the use of "0" for feature not used. However this could leader to serval consecutive header bytes with that are all `0x00` . Having some more regular transitions is desirable to know data is coming through.
- Considered having the frame length byte use all eight bits and allow for a 255 byte frame. Determined that being the same as KEN-B was a better choice.

END OF DOCUMENT
