

# Cursor-on-Target Capture from the Makito X

## Makito X v2.3

This Addendum supplements the [Makito X Encoder User's Guide](#) with the information required to use the Cursor-on-Target Metadata Capture option. Please refer to the user's guide for general information on the Makito X.

## Introduction

The Makito X supports both KLV (Key Length Value) and CoT (Cursor-on-Target) metadata capture and stream insertion. This optional feature allows the encoder to capture metadata and then incorporate the information within the metadata elementary stream of the standard MPEG Transport Stream. The metadata may be captured either from the serial port interface or from a UDP source.

This Addendum provides instructions for controlling and managing CoT metadata insertion parameters through the Makito X Web Interface.

For information on the CoT UDP implementation as well as KLV metadata capture, please refer to the [Makito X Encoder User's Guide](#) or the online help available from the Web Interface.

## Cursor-on-Target

The Makito X offers advanced capabilities for the translation of CoT (Cursor-on-Target) information (for UAV applications) into industry standard KLV format. The input format of serial or network metadata sources can be set to CoT from the Web Interface.

Cursor-on-Target is an XML-based protocol that enables Machine-to-Machine Targeting to:

- provide special forces the ability to click on a laser rangefinder designating a hostile target,
- pass precision coordinates,
- send mensurated target coordinates to an airborne strike asset, and
- download these directly into a GPS guided munition.

## CoT to KLV Metadata Mapping

This section describes the CoT to KLV metadata mapping implemented for the Makito X.

The requirement is to transmit one KLV message in the transport stream for each pair of Aircraft and SPI messages received by the Makito X. The format for these messages is provided in the following two sections:

- Aircraft Message
- Sensor Point of Interest (SPI) Message

If a message is missing a CoT element from these lists of aircraft and sensor data elements, the KLV message still transmits the information that was received. Reasons for missing information might include

a platform that does not support all of the CoT elements listed below or an incomplete message received by the Makito X.

## Aircraft Message

To distinguish between an aircraft message and associated data, the type field has the following values associated: `a-f-A-M-F-Q-r`, `a-f-A-C-F-r` and `a-f-A-M-F-M`

To simplify this to allow for future platforms or changes, do a match against `a-f-A` which maps out to be a friendly aircraft.

CoT Key (Original Requirement)	KLV Key Based on MISB EG0601.1	KLV LDS Name Based on MISB EG0601.1	16-byte Metadata Label or 16-byte Set Designator Based on MISB EG0104.5	Metadata Element or Set Name Based on MISB EG0104.5
<code>uid</code>	10	Platform Designation	06 0E 2B 34 01 01 01 01 01 01 01 20 01 00 00 00 00	Device Designation
<code>start</code>	72	Event Start Time UTC	0E 2B 34 01 01 01 01 01 02 01 02 07 01 00 00 00	Event Start Date Time - UTC
<code>time</code>	2	Unix Time Stamp	0E 2B 34 01 01 01 01 03 02 01 01 01 05 00 00 00	User Defined Time Stamp
<code>track course</code>	5	Platform Heading Angle	0E 2B 34 01 01 01 01 01 01 10 01 02 00 00 00 00	Angle to North
<code>track speed</code>	9	Platform Indicated Airspeed	Not defined in EG0104.5	Not defined in EG0104.5
<code>point lat</code>	13	Sensor Latitude	0E 2B 34 01 01 01 01 03 01 02 01 02 04 02 00 00	Device Latitude
<code>point hae</code>	15	Sensor True altitude	0E 2B 34 01 01 01 01 01 01 02 01 02 02 00 00 00	Device Altitude
<code>point lon</code>	14	Sensor Longitude	0E 2B 34 01 01 01 01 03 01 02 01 02 06 02 00 00	Device Longitude
<code>altitude roll</code>	7	Platform Roll Angle	0E 2B 34 01 01 01 01 07 01 10 01 04 00 00 00 00	(7 Platform Roll Angle) = (event/detail/spatial/altitude/roll)
<code>altitude pitch</code>	6	Platform Pitch Angle	0E 2B 34 01 01 01 01 07 01 10 01 05 00 00 00 00	(6 Platform Pitch Angle) = (event/detail/spatial/altitude/pitch)

## Sensor Point of Interest (SPI) Message

To distinguish an SPI message and associated data, the type field has the following value associated: b-m-p-s-p-i: x

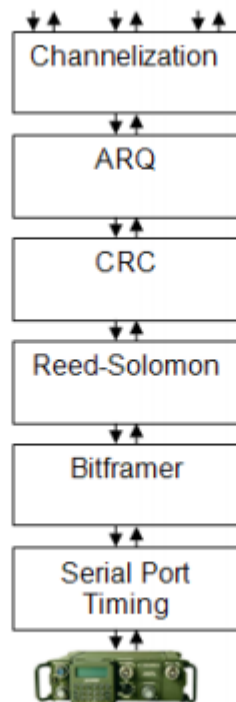
CoT Key (Original Requirement)	KLV Key Based on MISB EG0601.1	KLV LDS Name Based on MISB EG0601.1	16-byte Metadata Label or 16-byte Set Designator Based on MISB EG0104.5	Metadata Element or Set Name Based on MISB EG0104.5
uid	63	Sensor Field of View Name	06 0E 2B 34 01 01 01 01 04 20 01 02 01 01 00 00	Image Source Device
point lat	23	Frame Center Latitude	0E 2B 34 01 01 01 01 01 01 02 01 03 02 00 00	Frame Center Latitude
point hae	25	Frame Center Elevation	0E 2B 34 01 01 01 0A 01 02 01 03 16 00 00	Frame Center Elevation
point le	46	Target Error Estimate - LE90	Not defined in EG0104.5	Not defined in EG0104.5
point lon	24	Frame Center Longitude	0E 2B 34 01 01 01 01 01 01 02 01 03 02 00 00	Frame Center Longitude
point ce	45	Target Error Estimate - CE90	Not defined in EG0104.5	Not defined in EG0104.5
sensor azimuth	18	Sensor Relative Azimuth Angle	0E 2B 34 01 01 01 01 01 01 10 01 02 00 00 00	(18 Sensor Relative Azimuth Angle) = (sensor/azimuth) - (5 Platform Heading Angle)
sensor fov	16	Sensor Horizontal Field of View	06 0E 2B 34 01 01 01 02 04 20 02 01 01 08 00 00	FOV - Horizontal
sensor vfov	17	Sensor Vertical Field of View	06 0E 2B 34 01 01 01 07 04 20 02 01 01 0A 01 00	FOV - Vertical
sensor model	11	Image Source Sensor	06 0E 2B 34 01 01 01 01 04 20 01 02 01 01 00 00	Image Source Device
sensor range	21	Slant Range	06 0E 2B 34 01 01 01 01 07 01 08 01 01 00 00 00	
sensor elevation	19	Sensor Relative Elevation Angle	06 0E 2B 34 01 01 01 01 07 01 10 01 03 00 00 00	(19 Sensor Relative Elevation Angle) = (event/data/sensor/elevation) - (6 Platform Pitch Angle)
sensor roll	20	Sensor Relative Roll Angle	06 0E 2B 34 01 01 01 01 07 01 10 01 01 00 00 00	(20 Sensor Relative Roll Angle) = (event/data/sensor/roll) - (7 Platform Roll Angle)

# SerialD Version 1 Over-the-Air Protocol

This section describes the SerialD protocol used to transport CoT XML data on serial ports. For network sources, the XML data is directly carried over UDP.

## SerialD Overview

SerialD makes use of a layered communications architecture, as can be seen in Figure 1 below. In this arrangement, data to be sent over the wire enters via the top of the stack, and works its way down. Each layer processes the data, possibly adding, removing, or modifying bytes, and passes the data down to the next layer. The bottom most layer is connected to the communications hardware itself.



Each individual layer operates independently from the others; the layered design allows individual layers to be added, removed, reordered, or modified without requiring changes outside of the code for an individual layer.

The layer stack shown in the figure above is typical; however, it is not uncommon to encounter situations in which certain layers (such as the Reed-Solomon FEC layer) would be missing, or additional layers (i.e., for channel access) would be added. But the stack shown is the most common setup. This document describes the over-the-air format for version 1 of SerialD.

## Channelization

### Channelized Message Format

CHAN	DATA
------	------

Messages from one Serial to another are sent over virtual channels, allowing multiple logical data streams to be carried over a single serial connection. The channel is identified by a single byte prepended to the message, as seen in Figure 2 above. The channel with the identifier '0' (ASCII character 0x30) is used as the management channel for inter-node control communications. The channel with the identifier '1' (ASCII character 0x31) is designated as the CoT channel, and is expected to carry Cursor-on-Target messages exclusively. Other channels are treated as carrying opaque data, with no expectation as to their content. There are typically 5 user channels, designated as "CoT", "Chan2", "Chan3", "Chan4", and "Chan5", which use the channel tags "1", "2", "3", "4", and "5", respectively.

For example, if a station wishes to send the message "hello" on channel "3", it would send:

```
3hello
```

With the corresponding hex dump:

```
33 68 65 6c 6c 66
```

There are currently two messages defined for the management channel. The first one is the "Comm Check" request, and the other is the corresponding "Comm Check" response. These messages are used to verify basic communication over the end-to-end communication system. The format of the Comm Check message is:

```
?seq#time#host
```

where:

- `seq` is a unique (usually monotonically increasing) sequence number. It can be any alphanumeric string not containing the character '#'.
- `time` is the current time at the sending station. It is usually expressed as an ASCII string representing the current time, in seconds since the epoch (1970-01-01T00:00:00.00Z); however, it may be expressed in any format desired by the sending station, as long as the representation does not contain the character '#'. This time value is not to be interpreted by other stations; it will be returned to the sending station for round trip time (RTT) calculation.
- `host` is an identifier for the sending station. It is commonly the ASCII string returned by the `gethostname(2)` system call, but it may be any sequence of bytes that does not contain the character '#'.

Upon receipt of a Comm Check request message, a station should change only the first character from a '?' to a '!' (to indicate a change from a request to a reply), and send the new message back out.

Upon receipt of a Comm Check reply, the host should use the contents of the time field to calculate a round trip time for the message.

If a station with a hostname of "foo" wishes to send its 12th Comm. Check at 2006-0316T17:03:38.00Z, the string passed to the layer below (including the leading "0" to indicate that it is a message on the management channel) would be:

```
0?12#1132528618.00#foo
```

with hex dump:

```
30 3f 31 32 23 31 31 33 32 35 32 38 36 31 38 2e 30 30 23 66 6f 6f
```

Upon receipt, any (and all) other station(s) would respond with:

```
0!12#1132528618.00#foo
```

with hex dump:

```
30 21 31 32 23 31 31 33 32 35 32 38 36 31 38 2e 30 30 23 66 6f 6f0
```

## Automatic Repeat Request

ARQ Message Format

ARQHDR	DATA
--------	------

Reliable delivery of a message is accomplished by an automatic repeat request (ARQ) scheme. When a message is to be encoded for transmit, it is broken up into smaller chunks, typically of not more than 1000 bytes. Each chunk has a header applied to it, as seen in the table above, and is passed down to the next layer for further processing. If an ACK for an individual chunk is not received within a time-out period, typically 120 seconds, then the chunk is retransmitted. Chunks are retransmitted until the maximum retransmission limit is reached. This limit is typically 4, after which the message is considered to have failed.

The header for each chunk of an ARQ message is given by:

```
R#from#to#seq:part:total>
```

The leading " R " indicates that the message is requesting reliable delivery.

- **from** is the unique station identifier of the sending node (also referred to as its MAC address).
- **to** is the unique station identifier of the recipient node. The station identifiers can be any alphanumeric sequence made up of characters other than ' # '. There is no preset requirement on the length of station identifiers. The only requirement is that they be unique across the network.
- **seq** is a unique message identifier. It must be an ASCII string of a numeric values, but there is no requirement that its value be related to the value used on any other messages. (i.e., it does not need to be sequential in relation to previous messages, despite its name). It must be unique across all messages sent from a source node to a destination node, but it is the same for all chunks of a given message. Note that in order to support the reinitialization of one node without requiring the restart of all other nodes, the sequence number should attempt to be unique for all time, not just unique to a given invocation of the software (i.e., initialize with a sufficiently diverse random value, or save state across restarts).

- `part` indicates which chunk of the message follows.
- `total` indicates the total number of chunks. Both are ASCII strings representing decimal numbers.
- The "`>`" character indicates that it is an outgoing chunk.

Immediately following the header will be one chunk of message data. Message chunks are typically 1000 bytes long, but there is no requirement that the chunks be any particular size, or all the same size. The final chunk will typically be shorter than the previous ones, as the message is not padded to fill out a partial chunk.

When a reliable message addressed to a particular station is received from the radio by that station, the receiving station should indicate successful reception with the ACK message:

```
R#from#to#seq:part:total<
```

where `seq`, `part`, and `total` are exactly as taken from the received message. The "`<`" character indicates that the message is an ACK. No data should follow the ACK. (However, there may be multiple independent messages concatenated and sent, as detailed in the "Bit Framing" section.) The values for `from` and `to` are relative to the station sending the ACK. (i.e., they would be swapped from the message being ACKed.)

When a chunk of a reliable message is received, it should be buffered until all parts have been received. When all parts have been received, the entire message should be reassembled (without any ARQ headers), and passed up the stack for delivery. Message parts may arrive out of sequence. Individual parts may be received more than once. In either of these cases, the message should only be passed up exactly once, regardless of how many times various parts arrive.

In order to meet this requirement, it is necessary that a receiving station continue to keep a record of messages that have been received after the complete message has arrived and been passed up the stack. In the case where the ACKs are not properly received at the sending node, the sender may re-send the entire message multiple times, and despite multiple complete sets of chunks arriving, only one copy of the message should be passed up the stack.

If a message is not to receive reliable transport (i.e., SA message that are to be routinely resent), then the ARQ header should be '`U#`', to indicate unreliable delivery. When a message with a leading '`U#`' is received, the '`U#`' should be stripped off, and the message body should be immediately passed up the stack.

For example, if the message "hello" wishes to be sent reliably from station "a" to station "b", the sending station ("a") would transmit:

```
R#a#b#12:1:1>hello
```

with the corresponding hex dump:

```
52 23 61 23 62 23 31 32 3a 31 3a 31 3e 68 65 6c 6c 6f
```

In this case, the sequence number for this message is 12.

Upon receipt, station "b" would respond with:

```
R#b#a#12:1:1<
```

with the corresponding hex dump: 52 23 62 23 61 23 31 32 3a 31 3a 31 3c

and station "b" would pass the message up the stack, as it is complete.

If a reliable message arrives at any station other than the one to which it is addressed, it is discarded, no ACK is generated, and no message is passed up to higher layers.

If station "a" wanted to send the message "hello" unreliably, it would send:

U#hello

With the corresponding hex dump:

55 23 68 65 6c 6c 6f

As unreliable messages are all implicitly broadcast, there is no address specified. Upon receipt of the unreliable message, the receiving station(s) should pass the message up immediately, and no ACK is generated or sent. It should be noted that unreliable messages cannot be fragmented.

## Frame Check Sequence

Message with Appended Frame Check Sequence

DATA	CRC16
------	-------

Verification of successful reception of a message is ensured via a CRC16 frame check sequence (FCS). The FCS is appended to the message data as shown in the table above, and is generated using the IBM BISYNC CRC16 algorithm, with a generator polynomial of  $x^{16}+x^{15}+x^2+1$ . C code for a compatible implementation of the algorithm is:

```
unsigned short int crc16(unsigned char *d, int n) {
    unsigned short int crc=0;
    int i, bit;

    for (i = 0; i < n; i++) {
        crc ^= d[i];

        for (bit = 0; bit < 8; bit++)
            crc = crc & 1 ? (crc >> 1) ^ 0xa001 : crc >> 1;
    }
    return crc;
}
```

The 16 bit CRC code is stored in big endian format. When decoding received data, if the received CRC16 does not match the computed CRC16 for the data (up to, but not including the 16 bit FCS), then the entire message is considered to be in error and should be discarded.

For example, if the input message is "hello", then the resulting sequence would be:

68 65 6c 6c 6f 34 d2



## Forward Error Correction

### Reed-Solomon Encoded Message Format



Forward Error Correction (FEC) is accomplished by applying a (31,21) Reed-Solomon code to the data. The Reed-Solomon algorithm uses 5 bit symbols, with 31 total symbols per codeword, comprised of 21 data symbols followed by 10 parity symbols. Note that the Makito X implementation does not check parity symbols.

The code over  $GF(2^5)$  is defined by the primitive polynomial  $x^5+x^2+1$ . The power of the first of the 10 consecutive roots of the generator polynomial is 120, and the primitive of the field is given as  $x$ . Each codeword is 155 bits long. The encoded sequence is formatted as sets of 105 bits of message data, followed by 50 bits of parity information as shown in the previous figure. Symbols are packed into bytes MSb first. (i.e., the symbol 10111 would get packed into a byte as 10111000, with the next symbol starting in the 3 least significant bits of that byte.)

As the input data cannot necessarily be broken into an integer number of codewords, prior to the Reed-Solomon encoding process, a 16 bit little-endian length field (`len`) is prepended to the data. This length field is a count of the number of original data bytes in the message (not including the 2 bytes of `len`). The message data is then padded by appending sufficient zero bits as to result in an integer number of codewords. When a block of received symbols is being decoded, if the total decoded length is less than `len+2`, then the whole block is considered to be in error and is discarded. If it is longer than `len+2`, then the two bytes of `len` are removed, and the next `len` bytes are extracted and passed up to the next layer.

For example, if the input message is "hello", then the string passed to the Reed-Solomon encoder would be `05 00 68 65 6c 6c 6f .`, and the resulting coded sequence would be (including 49 padding bits appended to the source data to make it an integer multiple of 105 bits in length, and the five pad bits appended to the encoded sequence to fill out the last byte):

```
05 00 68 65 6c 6c 6f 00 00 00 00 00 00 00 3b e3 8b e5 c7 ac 20.
```

## Bit Framing

### Bit Framed Message Format

SYNC	LEN	LEN	LEN	DATA
------	-----	-----	-----	------

Bit framing is accomplished by marking the start of a message with the byte sequence (represented as hex values)

```
6f 48 65 59 21
```

Due to the requirement that these bytes be shifted out LSB first, the bit sync sequence seen on the wire would be:

```
1111011000010010101001101001101010000100
```

Immediately following that sequence, three copies of the length of the following data block (not including the bit framing header) are appended, as shown in the figure above. Each length is composed of a 16 bit little endian value indicating the number of bytes, followed by a length check, which is 16 bits calculated as  $((2^{16} - (2 * len)) \& 0xffff)$ . The data to be sent follows the three length plus check repetitions. For example, if the input sequence is "hello", the hex dump of the sequence of bytes sent out would be:

```
6f 48 65 59 21 05 00 f6 ff 05 00 f6 ff 05 00 f6 ff 68 65 6c 6c 6f
```

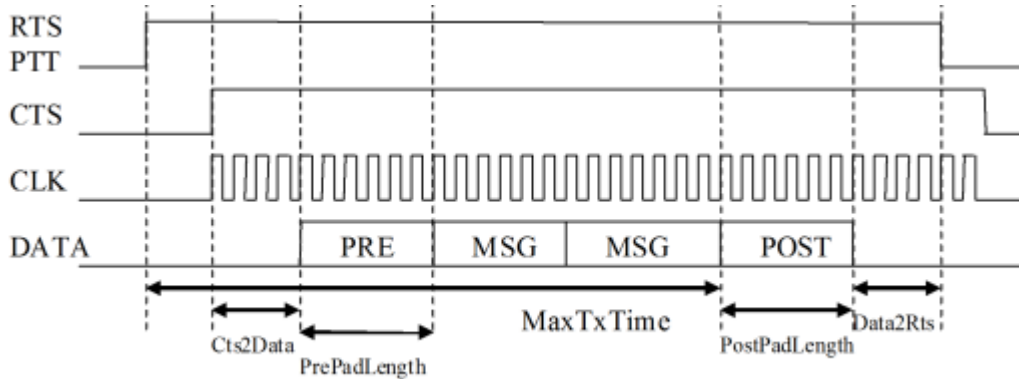
When data is received from the radio, it is compared to the sync pattern by successive bit shifts, until the detection threshold has been met. A valid sync sequence is received when the number of bit differences between the target sequence and the received sequence is 4 or less (90% match). If the complement of the target sequence and the received sequence differ by 4 or less bits, then an inverted sync is detected, and all subsequent data read from the port following the sync sequence is inverted.

Once a good packet has been detected, which occurs if at least one length is received correctly (i.e., one in which the computed checksum agrees with the received checksum), the len bytes immediately following the three encoded lengths are received and passed up to the next layer.

One consequence of the way in which the message length is encoded is that the maximum length of an individual message is  $2^{16} - 1$ , or 65535 bytes. In order to send messages longer than that, the message may be sent using the reliable delivery method (see the "Automatic Repeat Request" section), which will break the message up into smaller pieces prior to transmit. Note that there is no way to send a broadcast or best-effort delivery message with total size greater than 65535 bytes. Also note that this is the maximum size of the message body after all transforms performed by higher layers in the stack have been applied. This results in a maximum size for an unreliable message at approximately 43K bytes, assuming the FEC described in the section "Forward Error Correction" is in use.

## Serial Port Timing

Data to be sent out over the serial port to an attached radio must observe various timing constraints in order to be successfully received at a remote node. These relations can be seen in the following figure:



When a station has data to transmit, it must assert PTT (via RTS). The station waits for acknowledgment that the radio is ready to accept data, which may be indicated by the radio asserting CTS, the radio providing CLK, or there may be no indication. The sending station must wait for the `Cts2Data` time (specified in milliseconds) after CTS is received before attempting to send any data. If the radio does not provide CTS in response to RTS, CTS is assumed to occur simultaneously with the asserting of RTS, and therefore, the `Cts2Data` period begins when RTS is asserted. A typical value for `Cts2data` is 500 msec, but it varies depending on the communications hardware in use. Once the `Cts2Data` period has elapsed, the station sends out a sequence of zero or more pre-pad characters. These characters may be repeated occurrences of a single character, a rotating set of characters, or any other characters as necessary, as long as the sequence does not include the sync sequence discussed in the section "Bit Framing". Once the pre-pad characters have been sent, one or more messages may be concatenated and sent. Zero or more bits of filler data may be inserted between the messages, with message boundaries being demarcated as discussed in the "Bit Framing" section. Multiple messages may be concatenated until the total transmission time reaches `MaxTxTime`, which is specified in milliseconds. `MaxTxTime` is measured from when PTT is asserted. `MaxTxTime` is configurable, with a typical value of 60000 msec.

If RTS is asserted, and no CTS (or clock) is returned within a reasonable time (typically 10000 msec), then RTS is de-asserted, and the cycle begins again.

When there are no more messages to send, or `MaxTxTime` has elapsed, a sequence of zero or more post-pad characters are sent, subject to the same constraints as the pre-pad characters. Once the post-pad characters have been sent, PTT is held asserted for the time specified in `Data2Rts` (in milliseconds). Once `Data2Rts` has elapsed, PTT is released. `Data2Rts` often has the same value as `Cts2Data`.

After PTT is released, a sending station may not initiate a new transmission for a period of `IdleChannel`, which is specified in milliseconds. In addition, a station may not initiate a transmission until `TurnAround` (specified in milliseconds) has elapsed following any indication of channel activity. This includes receiving any bytes from the serial device, or any other indication provided, such as asserting of the CD line. Typical values for `IdleChannel` and `TurnAround` are 3000 msec and 1000 msec, respectively.

Data to be sent over the radio is transmitted least significant bit first. When driving a radio via a synchronous connection, there are no character framing bits. Each byte is transmitted as exactly 8 bits. As a consequence, when a transmission begins, there must be a continuous, uninterrupted flow of bytes out the port; any pauses will be interpreted as extra data bits inserted in the middle of the message, which will corrupt the data.

For example if the two bytes 0x12 and 0x34 are to be sent out, the bit sequence seen on the wire will be:

```
0100100000101100
```

where the bit on the left represents the first bit sent, and time increases to the right.

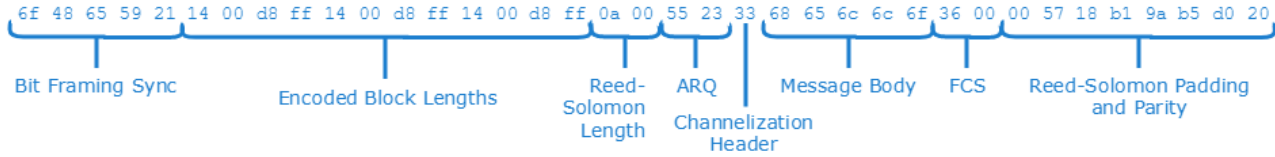
Data to be received should be shifted in LSb first, and delivered repacked into bytes such that the most significant bit of the  $i^{\text{th}}$  byte was received immediately before the least significant bit of the  $(i+1)^{\text{th}}$  byte.

All example byte sequences in this document assume this data packing convention.

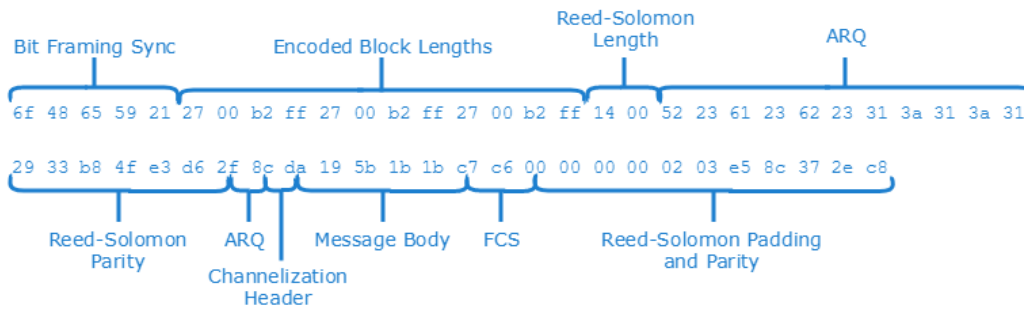
Data that is sent via an asynchronous connection (typically to a radio, null modem, or other asynchronous device) is sent with the appropriate number of start and stop bits, as required by the asynchronous device in use.

### Conclusion

Given the layer stack in the figure in [Serial Overview](#), if the message "hello" is to be sent on channel 3, and it is to be carried unreliably, the resulting byte sequence sent to the serial port is as follows:



The same message ("hello"), if sent reliably, with the same coding:




Note that due to the fact that the Reed-Solomon block size is 155 bits, which is 19.375 bytes, the coloring of individual bytes beyond the first block is approximate, as individual message bytes may contain bits from two separate layers. Individual codewords are not padded out to be an integer number of bytes; only the last codeword is padded so that the overall message occupies an integer number of bytes. This is why the actual message data in the above example is not recognizable as the ASCII string "hello".

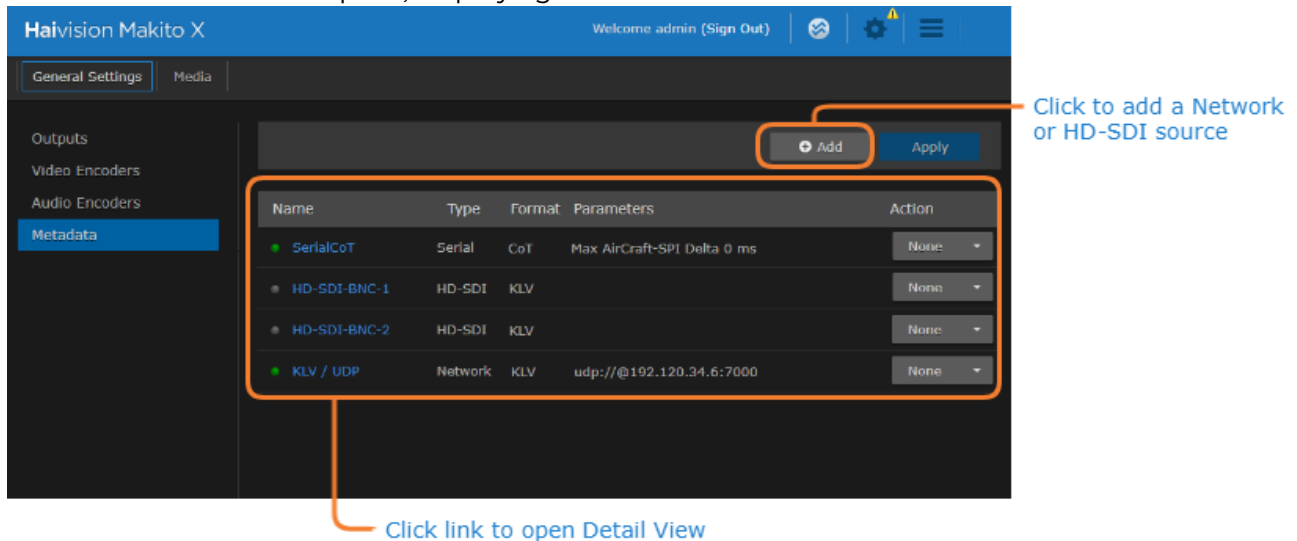
# Configuring CoT Metadata Capture from the Web Interface

**Note**

CoT Metadata Capture is an optional feature and must be installed at the factory or via a field upgrade by installing a license file. For metadata captured from the serial port, before you can configure the Metadata settings, the COM Port **Mode** must be set to **Metadata**. Please refer to [Managing the COM Port](#) (in the [Makito X Encoder User's Guide](#)).

**To configure CoT Metadata Capture:**

1. Click the  **Streaming** icon on the toolbar. On the Streaming page, click **General Settings** on the navigation bar and **Metadata** on the sidebar. The Metadata List View opens, displaying the list of defined Metadata sources for the encoder.

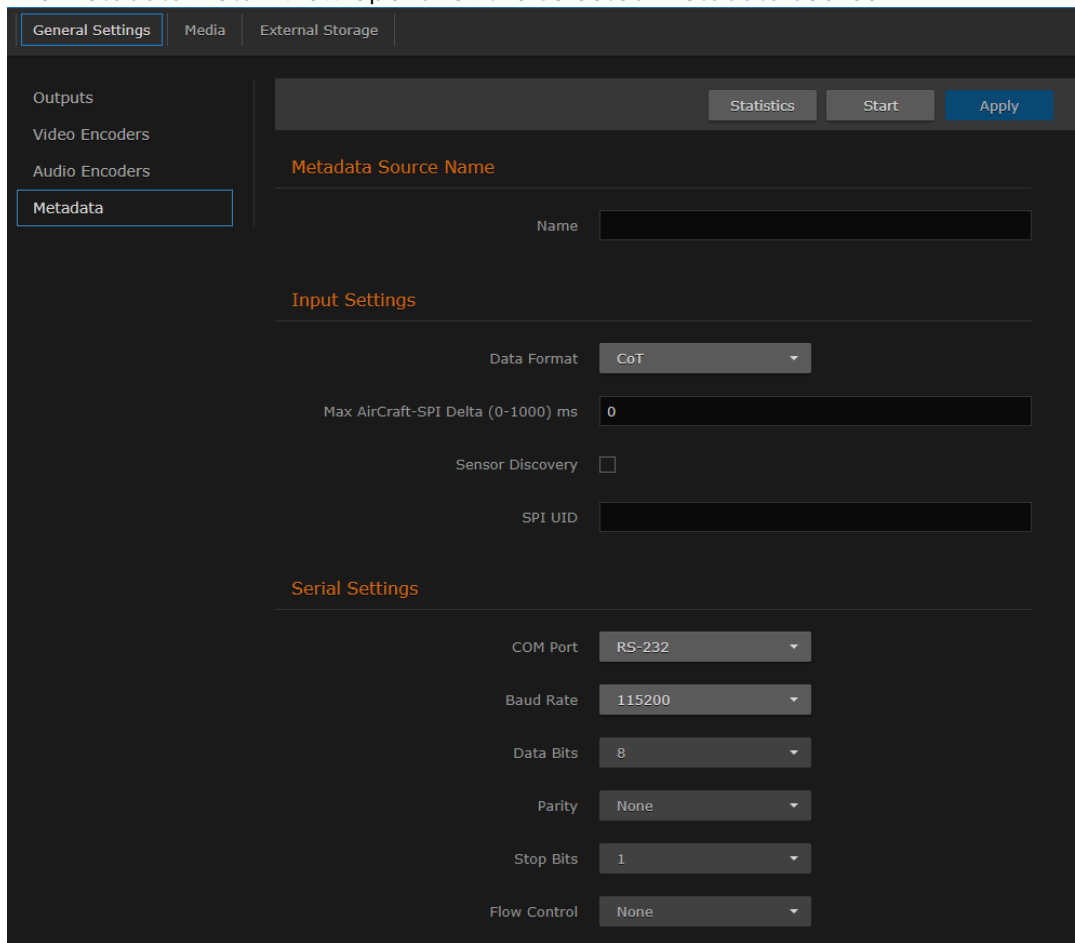


**Note**

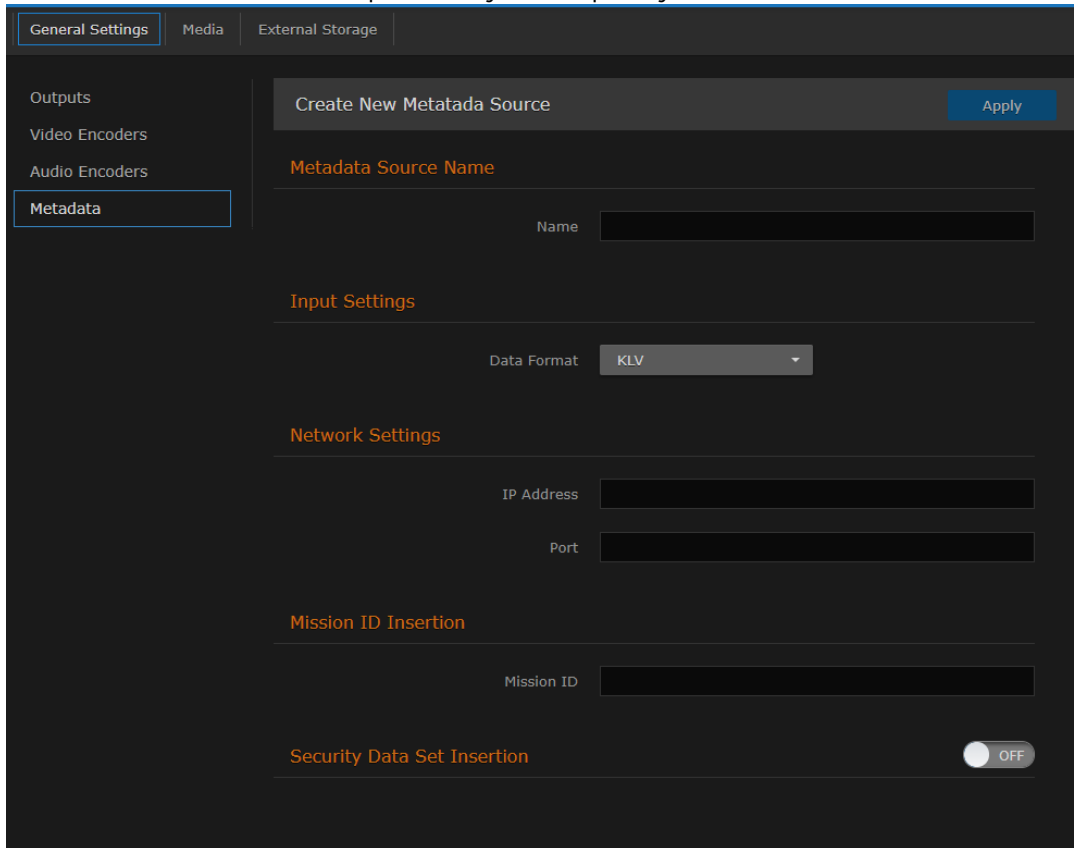
The Makito X auto-detects the hardware setup of the encoder, so when CoT Metadata Capture has been installed and Metadata has been enabled on the COM Port, the first input on the Metadata List View is filled in as shown above.

2. **(Serial CoT Input)** To view or modify CoT input details, click the link for the Serial source (i.e., the first line in the table).

The Metadata Detail View opens for the selected metadata source.



3. **(UDP Source)** From the Metadata List View, Click **Add**.  
The Metadata Detail View opens for you to specify a new metadata source.



4. Type in the Name for the input.
5. Select CoT (Cursor-on-Target) for the Data Format.
6. Enter the Max Aircraft SPI Delta. For details, see [Metadata Settings](#) (in the [Makito X Encoder User's Guide](#))
7. To apply your changes, click **Apply**.
8. To start or stop the stream, click **Start** or **Stop** (as applicable).

**Tip**

You can also change the status for a source from the List View by clicking the drop-down menu under **Actions** and selecting either Start or Stop (as applicable).

9. To view Metadata statistics, click **Statistics**. For details, see [Metadata Statistics](#) (in the [Makito X Encoder User's Guide](#))
10. To return to List View, click **Metadata** from the sidebar.

## Obtaining Documentation

This document was generated from the Haivision InfoCenter. To ensure you are reading the most up-to-date version of this content, access the documentation online at <https://doc.haivision.com>. You may generate a PDF at any time of the current content. See the footer of the page for the date it was generated.



## Getting Help

<b>General Support</b>	<p>North America (Toll-Free)  <b>1 (877) 224-5445</b></p> <p>International  <b>1 (514) 334-5445</b></p> <p><i>and choose from the following:</i>          Sales - 1, Cloud Services - 3, Support - 4</p>
<b>Managed Services</b>	U.S. and International 1 (512) 220-3463
<b>Fax</b>	1 (514) 334-0088
<b>Support Portal</b>	<a href="https://support.haivision.com">https://support.haivision.com</a>
<b>Product Information</b>	<a href="mailto:info@haivision.com">info@haivision.com</a>