M2T Control Reference v12Jun2017

Table of Contents

M2T Control Protocol and Language

M2T Control Protocol

The M2T control protocol has several key features:

- It is a **request-response** message protocol
- An **error reporting** mechanism is provided
- The message syntax is **text based**
- Messages may carry **data** payloads

M2T receivers are controlled using a request-response pattern. A controller sends commands to the device as **request** messages, and receives in return **response** messages which contains information about success or failure of the command, and possibly a data payload. Flow control is simple - issue a request and wait for the response. Receipt of the response message signals that the device is ready for another request, and so on.

The object, or **target** of a command is a **property** of the device, or an **action** to be executed.

Two request-response modes are available, which differ in the nature of the response message:

- **Normal mode** - the response message contains an indication of success or failure, and a data payload (if required).
- **Verbose mode** - the response message contains an indication of success or failure, the name and address of the command target, and a data payload containing the current value of the command target (if required).

The verbose mode supports certain 3rd party control programming styles where the response message needs to be self-describing. Since both the name of the command target and its value are included, the response message can be processed independently of the request message.

M2T commands are divided into 3 types:

- **Action command** - these are used to execute a pre-programmed **action** on the device.
- **Query command** - these are used to **read** data from the device in order to learn the value(s) of one of its properties.
- **Update command** - these are used to **send** data to the device in order to modify the value(s) of one of its properties.

Properties are single data values, or arrays of data values, that reflect the state of the device. Many properties are multi-valued. For example, with 2 transmitters, properties such as carrier power exist as an array of 2 values, one for each transmitter. In these cases an "address" qualifier is used to specify which particular value in a property's array of values is the target of the command. In most cases a wildcard address can be used to specify "all". Some properties are

read-only, others may be both read (queried) and written (updated). Details are found in the documentation for the command set.

Device properties include:

- Hardware related control settings for things like RF carrier power, audio input trim, network setup etc. These are usually read/write properties.
- Hardware related status indications such as audio input levels and clipping status, serial number etc. These are usually read-only properties.
- "Soft" settings such as labels for audio input channels. These are usually read/write properties.

Actions change the state of the device in some way without requiring an explicit data input. An "address" qualifier may be required to specify exactly which property is affected (or referenced) by the action. Details are found in the documentation for the command set.

M2T Control Language

This section concentrates on the language as used for 3rd party control of M2T transmitters over the USB and ethernet/TCP interfaces.

The Control language is formally specified in the ABNF grammar document.

Contents:

Request Messages

Request messages are composed from a number of components, some of which are mandatory, some optional. The order in which these appear in the message is always the same. These components appear as **tokens**, which are simply characters or groups of characters defined in the grammar. The tokens contained in a message may be separated by one or more whitespace characters such as **space** or **tab.** A special meaning is reserved for the whitespace characters **carriage return** and **linefeed** when used to mark the end of a message. Otherwise whitespace has no significance and is ignored.

The building blocks for a request message are:

- **verbose mode** token (optional). An exclamation point character (!) in this position forces a verbose mode response message to be returned.
- **target** token (mandatory). This identifies the target of the command, some property or action. This may be a variable reference. The target token may contain **only** the alphabetic characters [a - z] and [A - Z]. Example: `serial`
- **address** token (optional). This modifies the target of the command. It identifies a particular member of a array of values associated with some device property. It consists of an opening "**(**" character, then one or two groups of **digits**, and finally a closing "**)**" character. Only two digit groups are allowed, so at most a two dimensional space may be addressed. In the case where two digit groups are present, they must be separated by a **comma.** The numbers represented by these groups of digits must be **decimal** (base 10) numbers **greater than zero**. Addressing and indexing schemes start with the number 1. Some commands support the use of a **range** of numbers to specify an address, using 2 numbers separated by the colon character (:). In a further twist, some commands may allow one or both of the numbers to be replaced by a **wildcard** value, the character '***'.** This has the meaning that **all** of the members addressed by that index are being referenced. Also, references to **variables containing integer values** may be used to specify an address in place of a literal number. Examples: `(7)` or `(17,6)` or `(*)` or `(3,4:10)`
- **operator** token (optional). This modifies the meaning of the command from a simple action to either a **query** or an **update.** Two tokens are allowed, **?** and **=**. The query operator (?) signals that the request is a query for data, and the update operator (=) signals that the request carries data related to the desired action. No more than one operator token may be present.
- **data format** token (optional). This modifies the data format used in the transaction. Only one token is defined, **$**, which specifies the **hexidecimal encoded** format, and it must be preceded by either the query operator or the update operator. When following the query operator it signals that the data payload in the response message is expected to be encoded in hexidecimal format. When following the update operator it signals that the data which follows in the request message is encoded in hexidecimal format. When absent the default data encoding (described below) is in force. No more than one data format token may be present.
- **argument** token (optional). This contains data related to the request and may be present only when preceded by the update operator (and optionally the data format operator). There are several options for the argument in a request message:
  - **simple** data type. For example: `7` or `"Vocalists"`
  - **array** of simple data types. For example: `{1,0,1,0}`
  - **binary data** encoded in hexidecimal format, but **only** if preceded by the data format operator **$**. For example: `02FF1D22`
- **end of message** token (mandatory). This is a **carriage return** character indicating the end of the request message.

Verbose request

A request message can be prefixed with an exclamation point (!) character to force a "verbose" response message. The verbose response contains the name and address of the target (action or property being addressed), along with the current value of that target if it's a property. This supports certain 3rd party control programming styles where the response message resulting from a request needs to be self-describing so that it can be processed independently of the request message. For example, this query request (verbose mode):

```
!serial?<CR>
```

results in a response like this:

```
OK serial="6300101"<CRLF>
```

rather than a response like this (non-verbose mode)

```
OK "6300101"<CRLF>
```

Regarding verbose update requests, it's important to note that property values returned in the verbose response reflect the actual state of the property *after* the update has been attempted. This makes the verbose response reliable for refreshing 3rd party controller internal state or displays.

Request message types

- **Action request**

  Action commands may or may not have an address operator, and trigger an action which doesn't need input data and doesn't return any output data. Some examples:

  ```
  txafmutetog(1,1)<CR>
  ```

  This is a normal mode request to toggle the mute state on audio input channel 1 of transmitter 1.

  ```
  !txafmutetog(1,1)<CR>
  ```

  This is a verbose mode request to toggle the mute state on audio channel 1 of transmitter 1.

- **Query request**

  Queries may or may not have an address operator. They specify the query operator **'?'** and trigger the return of the requested data in the response message. They are the means by which a controller can read the value of some property of the device or poll the device to get status information, perhaps to drive indicators. Some examples:

  ```
  id?<CR>
  ```

This is a normal mode request for the value of the property "id".

```
txrffreq(2)?<CR>
```

This is a normal mode request for the value of the property "txrffreq" at address "(2)".

```
txrfpwr(*)?<CR>
```

This is a normal mode request for the value of the property "txrfpwr" at all addresses (wildcarded).

```
txafclip(*,*)?<CR>
```

This is a normal mode request for the value of the 2-dimensional property "txafclip" at all addresses (wildcarded).

```
!txrfenable(1)?<CR>
```

This is a verbose mode request for the value of the property "txrfenable" at address "(1)". A verbose mode response will be received by the controller.

- **Update request**

Updates may or may not have an address operator. They specify the update operator '=' and carry within them a data payload (the argument) for the device to process. They are the means by which a controller can change the value of device properties such as the gain of an audio channel or control a feature like a pink noise generator. Some examples:

```
fplcdbl=1<CR>
```

This is a normal mode request to update the property named "fplcdbl" to the value 1.

```
txrffreq(2)=470100<CR>
```

This is a normal mode request to update the property named "txrffreq", at address "(2)" to the value 470100.

```
txrfenable(*,*)={0,1}<CR>
```

This is a normal mode request to update the property named "txrfenable", at address "(*)" (wildcarded). The new values are supplied in the array of integers in the data payload. Property "txrfenable(1)" will be updated with the 1st integer in the array (0) and "txrfenable(2)" with the 2nd integer in the array (1). *A complete set of data must be provided when an update is directed at a wildcarded address.* If the "txrfenable" property is an array of 4 items, then an ERROR response will be received if an array of data is sent to the device whose size is not exactly 4.

```
txaftrim(*,*)={-6,-6,0,0}<CR>
```

This is a normal mode request to update the 2-dimensional property named "txaftrim", at address "(*,*)" (wildcarded). The new values are supplied in the array of integers in the data payload. Property "txaftrim(1,1)" will be updated with the 1st integer in the array (-6), "txaftrim(1,2)" with the 2nd integer in the array (-6), and so on. *A complete set of data must be provided when an update is directed at a wildcarded address.* If the "txaftrim" property is an array of 4 items, then an ERROR response will be received if an array of data is sent to the device whose size is not exactly 4. In the case of two dimensional arrays, the notation (*,*) used in addressing a property implies a responsibility for the controller to send the entire 2D array if data is transferred. Likewise, the notations (2,*) and (*,2) imply a responsibility for the controller to send the appropriate row or column slice of the 2D array.

```
!txrffreq(2)=520500<CR>
```

This is a verbose mode request to update the property named "txrffreq", at address "(2)" to the value 520500. A verbose mode response will be received by the controller.

Response Messages

Response messages are composed from a number of components, some of which are mandatory, some optional. The order in which these appear in the message is always the same. These components are tokens as described in the discussion of request messages above, and the same whitespace considerations apply.

The building blocks for a response message are:

- **status** token (mandatory). This indicates whether the request was successful or resulted in failure. The status token has two possible values, **OK** and **ERROR** . No more than one status token may be present in a message. Example: `OK`
- **target** token (optional). This is present only in verbose mode response messages, and identifies the target of the command, some property or action. This may be a variable reference. The target token may contain **only** the alphabetic characters [a - z] and [A - Z]. No more than one target token may be present in a message. Example: `serial`
- **address** token (optional). This is present only in verbose mode response messages, and modifies the target of the command. It identifies a particular member of a array of values associated with some device property. It consists of an opening "**(**" character, then one or two groups of **digits**, and finally a closing "**)**" character. Only two digit groups are allowed, so at most a two dimensional space may be addressed. In the case where two digit groups are present, they must be separated by a **comma.** The numbers represented by these groups of digits must be **decimal** (base 10) numbers **greater than zero**. Addressing and indexing schemes start with the number 1. Some commands support the use of a **range** of numbers to specify an address, using 2 numbers separated by the colon character (:). In a further twist, some commands may allow one or both of the numbers to be replaced by a **wildcard** value, the character '***'.** This has the meaning that **all** of the

members addressed by that index are being referenced. No more than one address token may be present in a message. Examples: `(7)` or `(17,6)` or `(*)` or `(3,4:10)`

- **operator** token (optional). This is present only in verbose mode response messages, where the update operator (=) signals that the response message carries data representing the value of the target of the command. Only the update operator may appear in a verbose mode response message.
- **data format** token (optional). This modifies the data format used in the transaction. Only one token is defined, **$**, which specifies the **hexadecimal encoded** format. When present it signals that the data payload in the response message is encoded in hexadecimal format. No more than one data format token may be present.
- **data** token (optional). This contains data requested by the controller in a query message or being returned in response to a verbose mode request message. In a verbose mode response the data is preceded by the update operator (and optionally the data format operator). There are only 3 options for the data payload in a response message:
  - **simple** data type. For example: `7` or `"Jury Box"`
  - **array** of simple data types. For example: `{1,0,1,0}`
  - **binary data** encoded in hexidecimal format, but **only** if preceded by the data format operator **$**. For example: `02FF1D22` No more than one data token may be present in a message.
- **end of message** token (mandatory). This is a **carriage return / linefeed** character pair indicating the end of the message.

Response message types

- **Action response**

  The response contains a status code indicating success or failure of the request processing. No data token is present. An example:

  ```
  OK<CRLF>
  ```

  Normal mode request processing succeeded. The action was executed.

  ```
  OK rxafmute(1,1)=1<CRLF>
  ```

  Verbose mode request processing succeeded. Either the name of the action executed or the name and value of the property affected by the action is returned. In this example the name, address and value of the property affected by the "txafmutetog(1,1)" action is returned.

  ```
  ERROR<CRLF>
  ```

  Request processing failed, the action was not executed. The status code ERROR is returned to the controller, which **should** recognize this status code and take appropriate action. The most common reasons why a action command request might fail are:

  - the target of the command is misspelled or nonexistent
  - the address is out of range

- **Query response**

  The response contains a status code indicating success or failure of the request processing. A data payload is present. Examples:

  ```
  OK 22<CRLF>
  ```

  Normal mode request succeeded. The requested data (22) is returned in the message.

  ```
  OK txrfenable(2)=0<CRLF>
  ```

  Verbose mode request processing succeeded. The name/address of the property queried is returned along with its current value of 0.

  ```
  ERROR<CRLF>
  ```

  Request processing failed. The status code ERROR is returned to the controller, which **should** recognize this status code and take appropriate action. The most common reasons why a query request might fail are:

  - the target of the command is misspelled or nonexistent
  - the address is out of range

- **Update response**

  The response contains a status code indicating success or failure of the request processing. A data payload is present only it the request was made in verbose mode. Examples:

  ```
  OK<CRLF>
  ```

  Normal mode request processing succeeded.

  ```
  OK txrfenable(1)=1<CRLF>
  ```

  Verbose mode request processing succeeded. The name/address of the property updated is returned along with its new value of 1.

  ```
  ERROR<CRLF>
  ```

  Request processing failed. The status code ERROR is returned to the controller, which **should** recognize this status code and take appropriate action. The most common reasons why an update request might fail are:

  - the target of the command is misspelled or nonexistent

- the address is out of range
- the data sent in the update request message is ill formed or out of range

Data types

The following data types are supported:

- **quoted string**
- **integer**
- **array of integer**
- **floating point**
- **array of floating point**
- **binary**

It is worth noting that many properties are naturally thought of as **boolean** types. These items use the **integer** type and but are limited to values **0** ("false" or "disabled") and **1** ("true" or "enabled"). Likewise, logical and comparison expressions evaluate to integer values, either **0** or **1**.

Data type formats

- **Quoted string type**

  A quoted string token consists of an opening **double quote** character ("), followed by zero or more characters, and finally a closing double quote character. Quoted strings may contain any printable ASCII character **except** the double quote character (") used to enclose them, and the backslash character (\) used as an "escape" character. To embed double quote characters or backslashes within a string they must be "escaped" by preceding them with a **backslash** character: `\"` or `\\`. The special escaped forms `\r` (carriage return), `\n` (newline) and `\t` (tab) are also recognized. Non-printable ASCII characters may be expressed using the hexadecimal escaped form `\xHH` where `HH` is any 2-digit hexadecimal number. Whitespace is ok within a quoted string, and is preserved. Quoted string tokens may not exceed 127 characters in length, exclusive of the enclosing double quote characters. Strings may be empty. Some examples:

  ```
  "Chairman"

  ""

  "The \"Lost\" Sheep"

  "serial?\r"
  ```

```
"id?\x0D"
```

- **Integer type**

  An integer token consists of an optional **sign** character (either "+" or "-") and a series of one or more decimal **digit** characters (0,1,2,3,4,5,6,7,8,9 and 0). No other characters are permitted. The maximum number of characters in an integer token, including any sign character, is 15. This limitation is general and has nothing to do with the range of **values** allowed for a particular property. Some examples:

  ```
  999
  ```

  ```
  -22
  ```

- **Array of integer**

  An array of integer token consists of a series of one or more **comma delimited** integer tokens enclosed in matching **braces.** Whitespace separating braces, integer tokens, and commas is ignored. The maximum size of an integer array is 64 items. Arrays may **not** be empty. Some examples:

  ```
  {0,-10,22,0,0,50}
  ```

  ```
  {1750}
  ```

- **Floating point**

  A floating point token consists of an optional **sign** character (either "+" or "-"), a series of zero or more decimal **digit** characters (0,1,2,3,4,5,6,7,8,9 and 0), a **mandatory decimal point** character ("."), and finally another a series of zero or more decimal **digit** characters. Exponential notation is not supported. The maximum number of characters in a floating point token, including any sign character and the mandatory decimal point character, is 15. This limitation is general and has nothing to do with the range of **values** allowed for a particular property. Some examples:

  ```
  0.6242
  ```

  ```
  -172.0
  ```

- **Array of floating point**

  An array of floating point token consists of a series of one or more **comma delimited** floating point tokens enclosed in matching **braces.** Whitespace separating braces, floating point tokens, and commas is ignored. The maximum size of an floating point array is 64 items. Arrays may **not** be empty. Some examples:

  ```
  {0.0,-10.6,-22.651,0.0,0.0}
  ```

```
{-1.000175}
```

- **Binary**

  A binary data token consists of a series of **hexadecimal** digit character **pairs** (0...9 and A...F). Each pair encodes one byte of binary data. The maximum size of an encoded binary block is 96 bytes in a request message to a device (which requires 192 hex digits total when encoded). Binary data tokens may be used **only** when the data format for the message has been set to **hexadecimal encoded** by the presence of the data format token **$.** An example

  ```
  $00A7C2990014
  ```

  The hexadecimal encoded data format and the binary data type are used by certain software programs for special purposes. Third party remote control applications don't need to use the binary type, it is described here for completeness.

ABNF Grammar for M2T Control Language

```
OCTET          = %x00-FF   ; any 8-bit data
CHAR           = %x01-7F   ; any US-ASCII character except NUL (1 - 127)
UPALPHA        = %x41-5A   ; any US-ASCII uppercase letter "A".."Z"
LOALPHA        = %x61-7A   ; any US-ASCII lowercase letter "a".."z"
DIGIT          = %x31-39   ; any US-ASCII digit "0".."9"
LF             = %x0A      ; US-ASCII LF, linefeed (10)
CR             = %x0D      ; US-ASCII CR, carriage return (13)
SP             = %x20      ; US-ASCII SP, space (32)
HT             = %x09      ; US-ASCII HT, horizontal tab (9)
DQUOTE         = %x22      ; US-ASCII double-quote mark (34)
BSLASH         = %5C       ; US-ASCII backslash (92)
QCHAR          = %x01-21 / %x23-5B / %x5D-7F; any CHAR except DQUOTE and BSLASH
WS             = SP / HT
SIGN           = "-" / "+"
OFFSET         = 1*DIGIT / "*"
RANGE          = 1*DIGIT ":" 1*DIGIT
ALPHA          = UPALPHA / LOALPHA
ALPHANUM       = ALPHA / DIGIT
HEX            = "A" / "B" / "C" / "D" / "E" / "F" / "a" / "b" / "c" / "d" / "e" / "f" / DIGIT
HEXNUM         = "x" 2HEX
ESCSEQ         = BSLASH ("n" / "r" / "t" / BSLASH / DQUOTE / HEXNUM)
STR_TOK        = ALPHA *(ALPHANUM)
INT_TOK        = *1SIGN 1*DIGIT
FLT_TOK        = *1SIGN *DIGIT "." *DIGIT ; note that bare "." is valid
QSTR_TOK       = DQUOTE *(QCHAR / ESCSEQ) DQUOTE
CRLF           = CR LF
OK_TOK         = %x4F %x4B ; uppercase string "OK"
ERROR_TOK      = %x45 %x52 %x52 %x4F %x52 ; uppercase string "ERROR"
input          = request / verb_request
output         = (response / verb_response) *WS CRLF
request        = (query / hquery / update / target) *WS CR
verb_request   = "!" (query / hquery / update / target) *WS CR
response       = status *WS (argument / hargument)
verb_response  = status *WS target "=" *WS (argument / hargument)
status         = OK_TOK / ERROR_TOK
query          = target *WS "?"
hquery         = target *WS "?" *WS "$"
update         = target *WS "=" *WS (argument / hargument)
argument       = INT_TOK / FLT_TOK / QSTR_TOK / intarray / fltarray
hargument      = "$" 1*(*WS 2HEX) ; note that size of data must be > 0
target         = STR_TOK [*WS arraydims]
arraydims      = "(" *WS arrayoffsets *WS ")"
arrayoffsets   = arraydim [*WS "," *WS arraydim]
arraydim       = RANGE / OFFSET
intarray       = "{" *WS intsequence *WS "}"
intsequence    = INT_TOK *( *WS "," *WS INT_TOK)
fltarray       = "{" *WS fltsequence *WS "}"
fltsequence    = FLT_TOK *(*WS "," *WS FLT_TOK)
```