Teratronik
elektronische systeme gmbh

MTD16
Protocol

Date: 2015-02-05

Table of Contents

Date	Author	Changes
2012-02-05	O. Wölfelschneider, Teratronik GmbH	Start

1. Introduction

Message Tag Data or *MTD16* is a data encapsulation used by multiple Teratronik products. It is a binary format using a length-tag-data structure.

MTD16 does not provide transmission error detection or recovery. This is typically handled by TCP/IP or, on serial ports, by an adequate serial encapsulation.

While being a fully binary message format that allows efficient use of bandwith even on slow links, the idea of the protocol is to hide as much as possible of that binary data during software development. This makes working with MTD16 feel closer to working with something like JSON or XML.

Hiding of the binary numbers is done with the aid of several tools that automatically convert binary message data to and from a human readable form. Also, during code development, the numeric message codes are used by means of symbolic names only. Code generators automate the mapping between message codes and symbolic names.

2. Typical use with TCP/IP

This is a description of a typical implementation when using TCP/IP for transport.

Clients connect to their server using the TCP protocol. Address and port of the server is configured locally at the client.

The clients trust the error detection and retransmit capabilities of TCP. There is no further checksumming in the payload. There is already a 32-bit CRC on the ethernet layer.

If a client or server detects an unrecoverable error in the data, it will disconnect and fail any pending transaction.

Stations try to reconnect to the server using a random backoff algorithm. In case of a connection failure, the first retry is after 500ms. Each further retry increases the delay by a random value between 250ms and 500ms, up to a maximum of 5000ms. This mitigates a network storm when multiple stations try to reconnect at the same time.

If encryption is required, then a standard SSL/TLS method is used. This is transparent to the *MTD16* protocol layer.

3. MTD16 Data Format

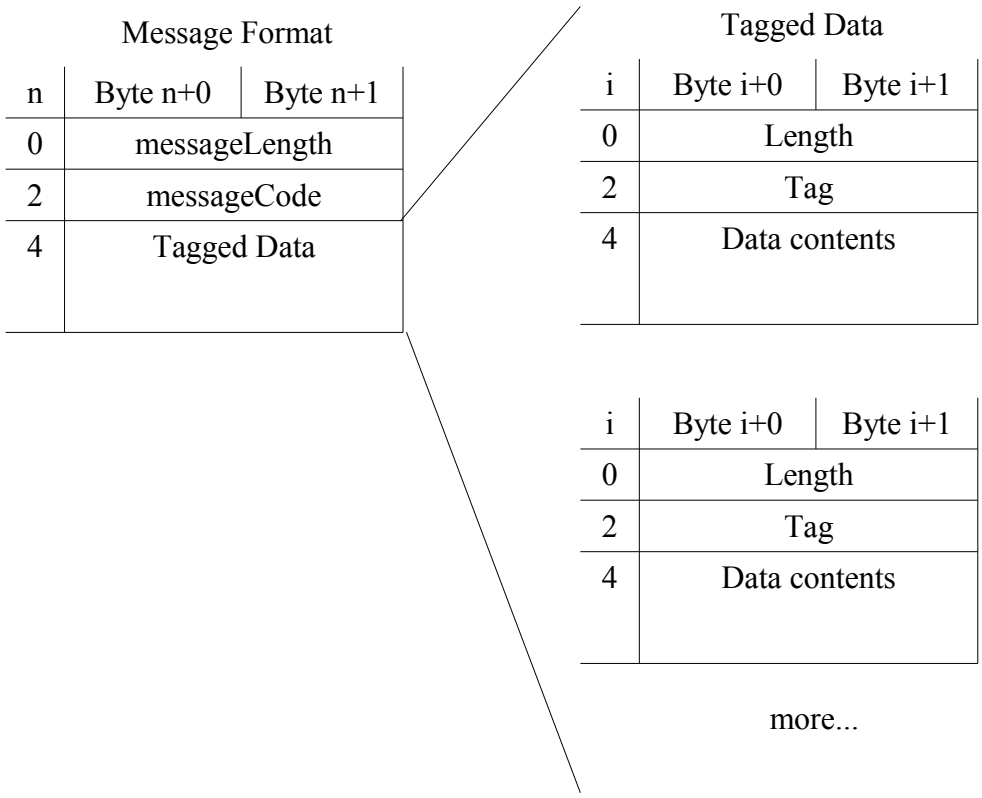
Message contents are encoded in a binary tagged data format. This allows adding more message fields at any time without much hassle.
The MTD16 format uses little endian mode (Low byte first) throughout its data.

3.1 Frame format

All tag and length fields are of 16 bit size and are sent low byte first. The length field indicates the number of bytes following after the length field.

A message always consists of at least one outer tagged block. The field messageCode indicates the kind of message, for example *StatusReport* or *PrintReceipt*.

The data variables of a message are itself encoded in the same tagged format as the outer message frame.



Important: **The protocol makes no guarantees about the order of tagged data fields inside a message unless explicitly documented for the message. The receiver of a message must be prepared to decode the fields in whatever order they arrive.**

Good practice: **Programs that are processing MTD16 messages should always expect that there are tags inside a message that they do not expect. Do not log a warning or error in that case. This allows extending the protocol without breaking older software.**

3.2 Tag structure

The fields *messageCode* and *Tag* follow a basic structure. The upper bits of the 16 bit tag code indicate the data type. This allows a debugger to print meaningful information without knowledge of the actual application.

Basic Data types

For basic data types, the upper four bits of the tag code indicate the data type. The lower twelve bits can be assigned freely.

0 - **MTD16DT_Binary**
Any data that does not fit the other data types.

1 - **MTD16DT_Integer**
Signed or unsigned integer value. The length of the data field is 1 .. 4 bytes. If the length is less than four bytes, the missing bytes are assumed to be zero. Data type supports any kind of integer between 8 and 32 bits.

2 - **MTD16DT_Bool**
One byte boolean field. Value 0x00 indicates false, anything else indicates true.

3 - **MTD16DT_String**
Text string. Usually encoded using UTF-8, unless application specifies otherwise.

4 - **MTD16DT_Date**
A calendar date, encoded as the days elapsed since 1990-01-01. Otherwise this field is encoded like a basic integer.

5 - **MTD16DT_Time**
A calendar time, number of seconds since 00:00:00. Otherwise this field is encoded like a basic integer.

6 - **MTD16DT_DateTime**
A combined calendar date/time. The field length is either eight or ten bytes. The first four bytes encode the number of days since 1990-01-01. The second four bytes encode the seconds since 00:00:00. If present, the last two bytes encode milliseconds (0...999). If the millisecond data is not included, they are assumed to be zero.

7 - **MTD16DT_BitArray**
The data is interpreted as an array of single bits. The first byte contains bits 0..7, and so on. Any bytes not present are assumed to be zero.

8 – ***Extended***

Indicates an extended data type, where the upper eight bits indicate the data type. Discussed below in more detail.

9 - **MTD16DT_NetworkAddress**

Contains a network address in network byte order (MSB first). Field size is four bytes for IPv4 addresses, 16 bytes for IPv6 addresses and six bytes for MAC addresses.

10 – **reserved**

11 – **reserved**

12 - **MTD16DT_List**

This is used for deeply nesting messages within messages. The data content is considered an array of tagged data with multiple values having the same tag.

13 - **MTD16DT_Request**

This is used to encode the outer frame of a request message. The sender of such a request expects to gets an answer to this request encoded as type **MTD16DT_Answer**.

14 - **MTD16DT_Answer**

Encodes the outer frame of a response message. Contains the answer to a message of type **MTD16DT_Request**.

15 - **MTD16DT_Message**

This is used for deeply nesting messages within messages. The data content is again some tagged data.

Extended Data Types

Extended data types encode the data type in the upper eight bits. The lower eight bits can be assigned freely.

0x80 - **MTD16DT_Point**

Always contains four data bytes. The first two bytes encode the X coordinate, the final two bytes encode the Y coordinate.

0x81 - **MTD16DT_Rect**

Always contains eight data bytes. Two bytes each are used in consecutive order: X coordinate, Y coordinate, Width, Height.

0x82 - **MTD16DT_Size**

Always contains four data bytes. The first two bytes encode the width, the final two bytes encode the height.

4. Tag definition file

All tags and their codes are defined in an XML file that maps the numeric tag codes to human readable names. When properly implemented, a programmer using the MTD16 system will almost never have to deal with the numeric codes themselves.

This is an example definition file.

```
<?xml version="1.0" encoding="UTF-8"?> <!-- -*- nxml -*- -->
<mtd16>
  <!--Command Messages-->
  <tag name="Ping" id="0xD001"/>
  <tag name="Pong" id="0xE001"/>
  <tag name="StatusReport" id="0xD800"/>
  <tag name="StatusReportResponse" id="0xE800"/>
  <tag name="PrintReceipt" id="0xD802"/>
  <tag name="PrintReceiptResponse" id="0xE802"/>
  <!--Generic tags -->
  <tag name="StatusCode" id="0x1000" comment="Result status code" display="4x">
    <enums>
      <enum name="Success" id="0x0000" comment="Indicate success"/>
      <enum name="Error" id="0x0002" comment="Generic failure code"/>
    </enums>
  </tag>
  <tag name="MachineStatus" id="0x7620" comment="General machine status.">
    <bits>
      <bit name="Online" id="0"/>
      <bit name="Enabled" id="1" comment="Station is open to the public"/>
    </bits>
  </tag>
  <tag name="Text" id="0x3500" comment="Generic text field"/>
  <tag name="Name" id="0x3030" comment="Any name"/>
  <tag name="Index" id="0x1335"/>
  <tag name="Type" id="0x133C"/>
  <tag name="Key" id="0x3335"/>
</mtd16>
```

Commentary on the example file

The pair of tags **`Ping`**/**`Pong`** and also the pair **`StatusReport`**/**`StatusReportResponse`** are defined as tag types Request and Response. This defines these pairs as being a message exchange that belongs together.

The tag StatusCode has a nested list of enum values. This allows the programmer to use symbolic enum names instead of numbers inside the source code. Also the debugger can use the symbolic names when logging a message.

Similarily, the tag **`MachineStatus`** defines a nested list of bit values. This allows to store multiple named flags inside one field of type BitArray. Again, in source code and when debugging, the symbolic names are used.

A few generic tags, like **`Text`** or **`Name`** have been added. Their function depends on the message inside which they appear.

XML style comments in the XML file are shown by the MTD16 tag editing tool when displaying the file for editing.

The comment property seen with some tags is only used by the tag editing tool, where the comments are displayed beside the tags.

The display property, also seen with some tags, can be used by a debugging tool to help formatting the data contents of a field in a pleasing way. (TODO: Document the format of the display property)

5. Debugging tools

There exists a human readable ASCII format for MTD16 that helps debugging the binary protocol.

Applications can dump the messages in this format to logging for tracing.

The mapping between the numeric tag codes and the names for logging purposes are taken from the XML definition map file.

This is an example „Hello World!“ message.

```
12 00 02 D8 0E 00 00 35 48 65 6C 6C 6F 20 57 6F 72 6C 64 21
```

Green: Length fields, Blue: Tag fields

In debug format, a message looks like this:

```
PrintReceipt=(sText="Hello World!")
```

Features of the debug format:

- parentheses ( ) indicate nesting
- Strings are printed in double quotes
- Tag names (can) have their data type prepended in abbreviated form
- For enums and bit arrays, symbolic names are printed

6. Development tools

There are a few tools available to aid in software development.

- *MTD16 Tag Definition Editor*

  This is a GUI tool that allows editing the contents of a tag definition XML file. The tool especially helps avoiding assignment of the same numeric code to multiple tags.

  The tool can generate source code from a tag file for C, C++, C# and Lua.

- *Code generating tool*

  A commandline tool for use in build scripts, this little program can automatically convert a tag definition XML file into source code for inclusion into an application. Available languages are C, C++, C# and Lua.

- *MTD16 Testing Tool*

  This tool does a live trace of messages as they are exchanged between to applications. It does this by being configured as a proxy between two peers.

  Armed with a tag definition XML file, the tool displays the message in a human readable form for ease of debugging. Optionally it can also print the bytes as hexdump directly.

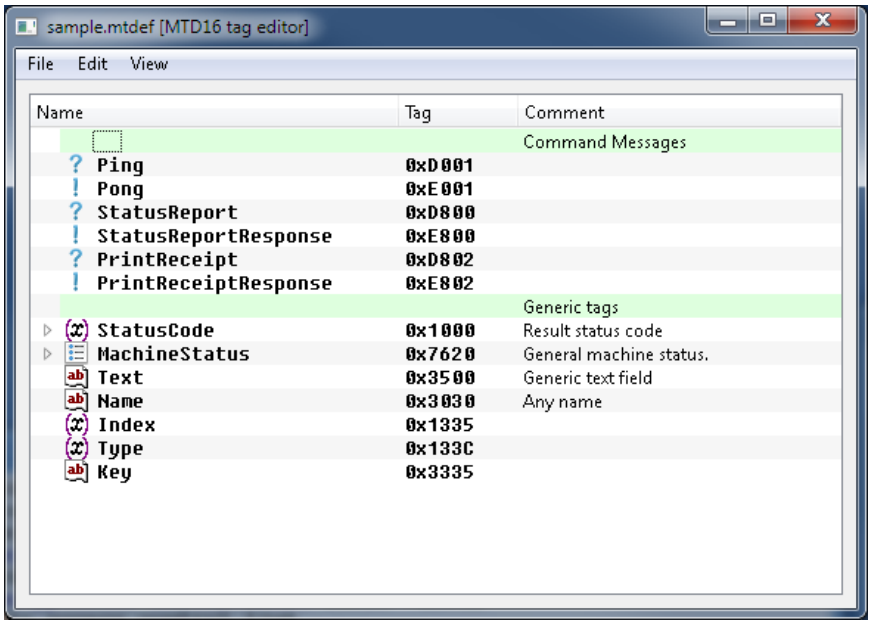
To get these development tools, download the latest Core4Manager package from this location:

http://www.teratronik.org/core4/download/Windows/

This will install the tools together with a few other development tools which are not the subject of this documentation. After installation, find the MTD16 tools in the start menu under *Core4 SDK*.

6.1 MTD16 Tag Definition Editor

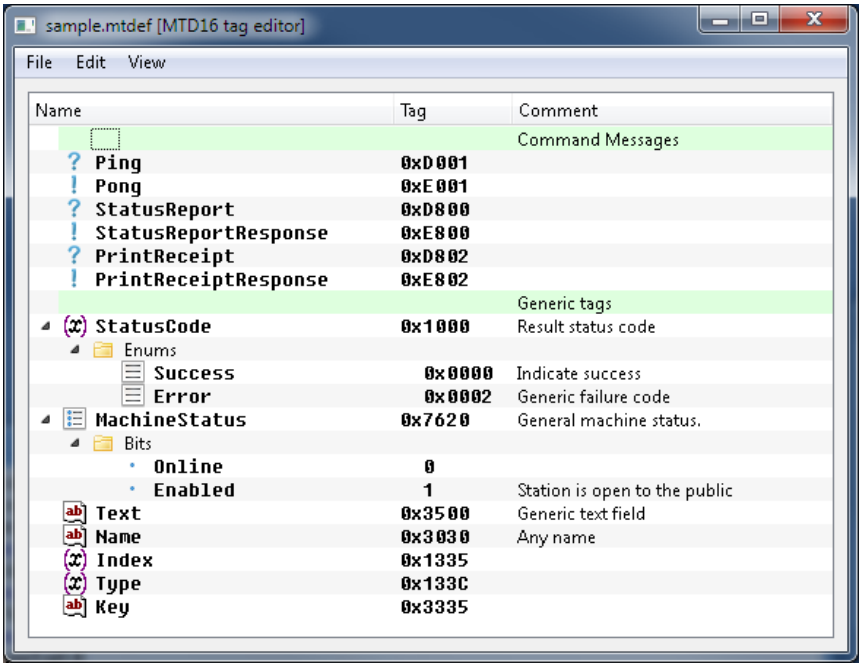
This shows the main window of the tag editor, displaying the sample definition file from page 8.



The tags are displayed in the order as they appear in the XML file. XML file comments are shown as section headings with a slight green background. Comment attributes of explicit tags are also shown.

Hovering over a tag name with the mouse points displays the data type of the tag.

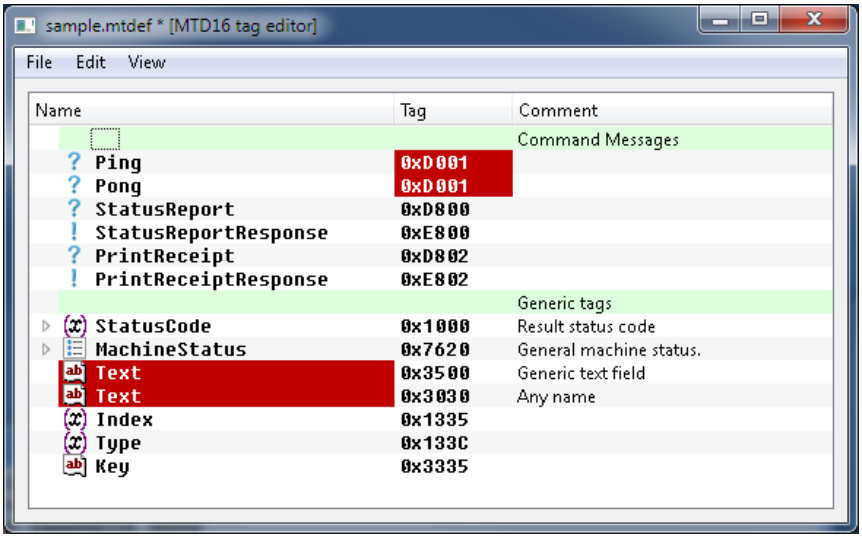
Tags that have enums or bits defined, are shown with a little arrow. Clicking the arrow displays additional information.



It is possible to navigate through the table with the cursor keys, or by clicking a cell with the mouse. To edit a single cell, press F2. To edit the full row in a popup dialog, double click a row.

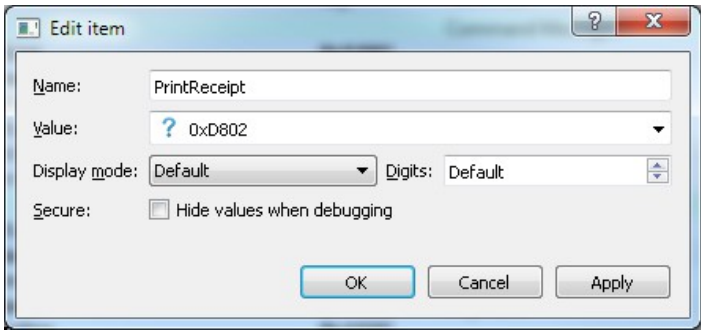
Have a look at the Edit menu, which allows insertion or deletion of lines or adding a enum or bit definition branch to a tag. All editing options have optional hotkeys, as shown in the menu.

The following image shows a ham-fisted attempt to use the same code or name at two places. The editor indicates the collision in red. The editor still allows saving of the file.



Fortunately, the editor supports Undo/Redo (Ctrl-Z, Ctrl-Y).

This is the editing tool that pops up when double clicking on a row.



Name and Value are already known. The settings for *Display mode*, *Digits* and *Secure* are only used by debugging tools, like the MTD16 testing tool. The *Display mode* is applied to numeric values and can be used to enforce display as *Signed Decimal*, *Unsigned Decimal*, *Hex*, *Octal* or *Binary*. When forcing a number mode, the number of digits to display can also be selected. By setting the Secure bit, the debugger will not show the contents of the tag when tracing a message. Use this for e.g. credit card numbers.

The *View* menu allows displaying the result of the source code generator for the current definition file. It is possible to just cut+paste from the view into your source code, but there also exists a scriptable command line tool to automate this. (Explained later in this document.)

See the appendix for examples of generated code.

6.2 Code generating tool

The commandline tool c4mkmtd converts a tag definition XML file into source code. The tool can be integrated into build scripts to automatically include changes to the tag definitions into application source code. When installed on windows, the tool typically resides in

```
C:\Program Files (x86)\Teratronik\Core4Manager\c4mkmtd.exe
```

When called with no arguments, the tool displays a short help notice:

```
Usage: c4mkmtd [<options>] <inputfile>

    -T,--trace=<n>          Enable tracing level <n>
    -H,--header=<filename>  Write C-Style header
    -L,--lua=<filename>     Write Lua-Style code
    --c++-header=<filename> Write C++-Style header
    --c++-lut=<filename>    Write C++-Style lookup table
    -#,--csharp=<filename>  Write C#-Style code
    --prepend=<filename>    (C++,C#) Prefix output with contents from file
    --append=<filename>     (C++,C#) Add contents from file to end of output
    --suffix=<text>         (C++,C#) Append text at end of each identifier
```

For example, this call generates both Lua and a C header at the same time:

```
c4mkmtd --header=example.h --lua=example.lua example.mtdef
```

See the appendix for examples of generated code.

6.3 MTD16 Testing Tool

The MTD16 Testing Tool does a live trace of messages. It can be used in two ways:

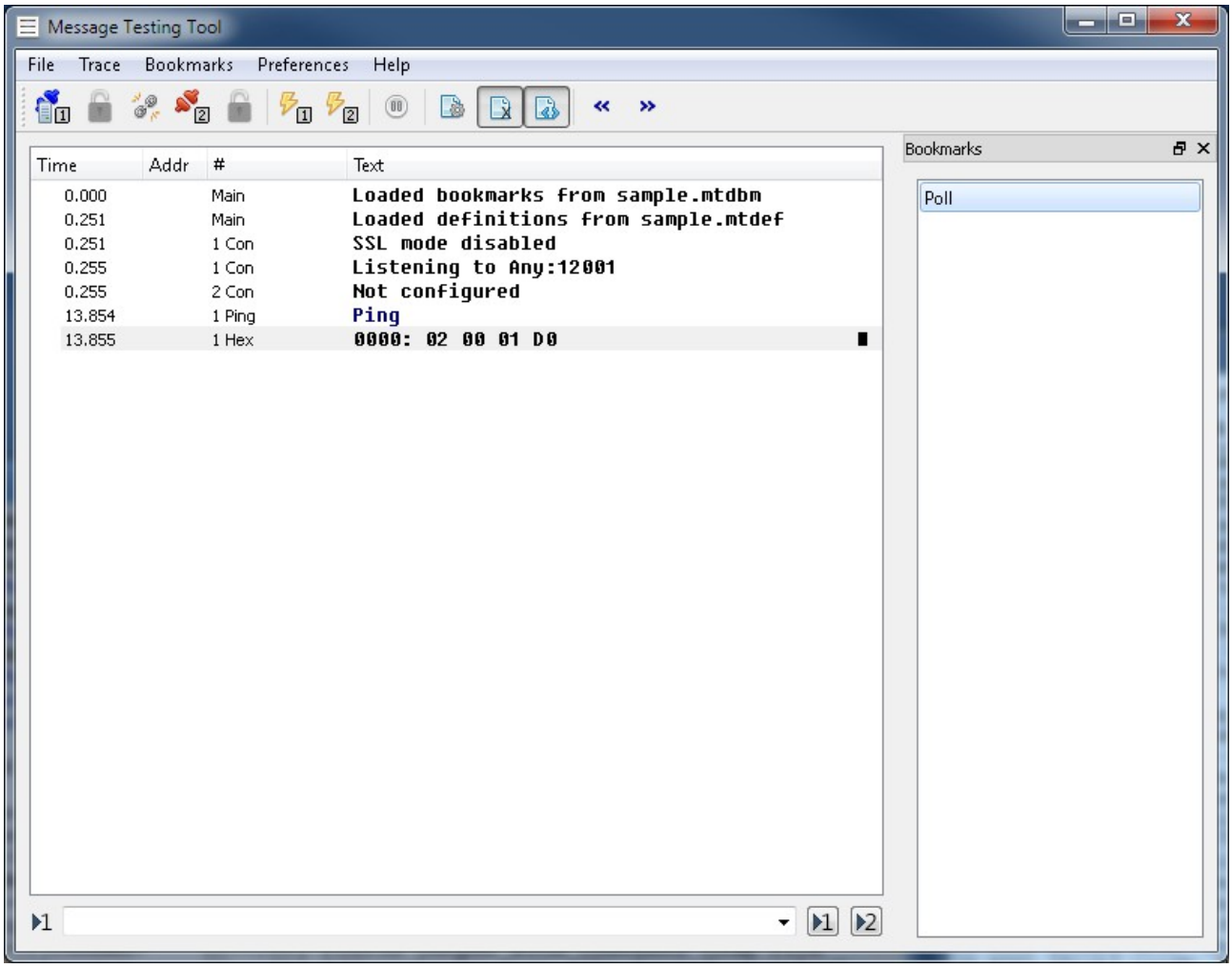
- Direct control

  With direct control, the testing tool communicates over a single channel with a peer. The tool displays all messages from the peer, and in turn can send messages to the peer.

- Man-in-the-Middle mode

  The testing tool is configured as a proxy between two other applications that are communicating via MTD16. Messages received from one connection are forwarded to the other, both ways. The tool displays all messages that it forwards.

6.3.1 Main window of the testing tool



After starting, the tool automatically loads the tag definition file it was using last time. On first run, or when switching projects, you will need to open a definition file using the File menu. The example screenshot has loaded the example definition file from page 8.

The tool has already been configured to listen for incoming connections on port 12001. Connections do not use encryption.

On the right hand side of the window is a list of Bookmarks. These bookmarks can be freely configured to send a specific message. When clicking the bookmark, the message is sent through the channel. In this example, a single bookmark has been defined that sends a simple Ping message. The tool has been configured to display the message text and also a hexdump of the message raw data.

6.3.2 Setting up a connection

The testing tool can handle two connections at the same time if required. This is necessary for the proxy mode, where the tool can forward data between two other parties. Both connections are handled identically. To configure a connection, Chose *Preferences -> Connection 1* or *Preferences -> Connection 2* respectively.

Each connection can be configured to one of four modes.

- Disabled

  The connection is not used.

- Serial

  The connection uses a serial port. You will need to chose a port and speed settings.

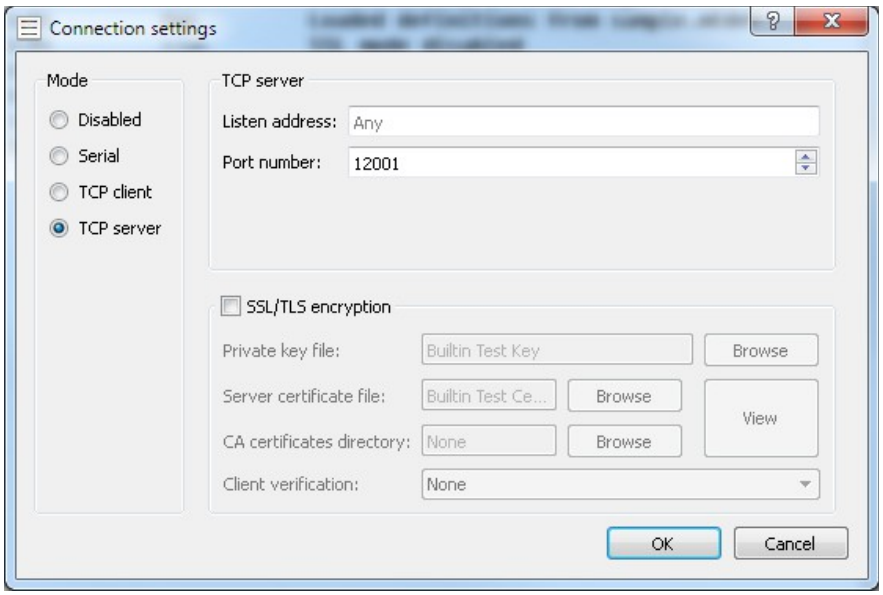
- TCP client

  The tool will open a TCP connection to a remote server. Parameters are server name or IP address and the port number.

- TCP server

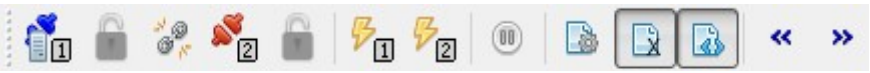
  The tool listens for an incoming TCP connection from a remote client. Typically the listen address is kept empty and a port number is chosen depending on the application.

For the two TCP modes, it is optionally possible to use SSL/TLS encryption. To use this, the tool needs to know the proper keys. The standard certificate files can be selected via the provided browse buttons.



6.3.3 Connection Control

The connections are controlled by using the toolbar buttons on the top of the main window.



Each connection has a control button and an indicator for the encryption state. Click the control button to cycle between the connection states.

- Connection closed or disabled

- TCP server mode is waiting for a connection

- Connection established

When connected:

- Unencrypted connection

- Encrypted connection

If two connections are used, the link button between the connections controls data forwarding:

- Data is **not** forwarded

- Data is being forwarded between both connections

Error injecting

- Only functional with serial connections. With a special error injection device, these buttons can cause transmission errors on a serial connection to test error recovery procedures.

Connection control (continued)

Beware of the pause button. The pause button stops updating the display. If you are not seeing any data being logged, check that the pause button is not in pause state.

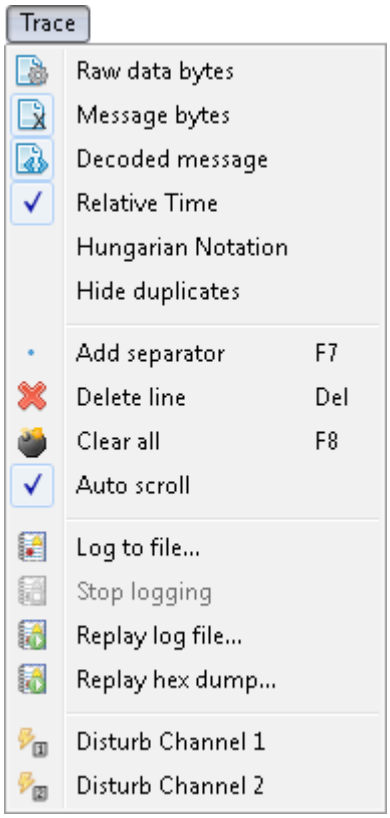
- Normal mode. Clicking the button turns on pause.
- Pause mode. Clicking the button resumes logging.

Trace selection

- Enables hexdump of the raw data bytes.
- Enables hexdump of the message data bytes.
- Decode and trace a human readable form of the messages.

Note that for TCP/IP, there is no difference between the two hexdump variants. Only when using serial port mode, these hexdumps display data before and after the serial protocol encoding.

6.3.4 Trace menu



The trace menu controls how the data logging is handled. The first three items reflect the trace selection also available via the toolbar, explained on the toolbar page.

- Relative time
  When off, timestamps display the time when the message was received in millisecond resolution.
  When on, timestamps log the number of milliseconds since the start of trace.

- Hungarian notation
  When on, each tag name gets a short prefix indicating its type. (Examples: s for string, i for integers, ...)

- Hide duplicates
  Useful when a protocol with a lot of polling is traced. When the same message is received multiple times in a row, then it is not logged multiple times. Instead the log window indicates a counter with the number of repeats.

- Add separator
  Inserts a dashed line into the log. This is meant as a visual aid and has no other function.

- Delete line
  Single lines can be removed from the log

- Clear all
  Erases the log window contents

- Auto scroll
  When on, automatically scrolls the window to the last entry whenever a new entry is added.

- Log to file...
  A file can be selected, where the received data is logged to.

- Stop logging
  Stops the logging to a file.

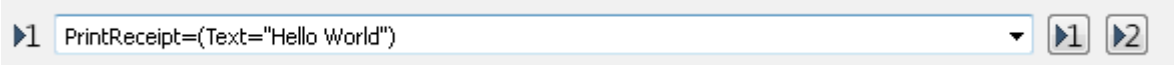
- Replay log file
  A previously generated log file is replayed into the log window.

- Replay hex dump
  This can replay a log file that was generated with hex dumps on.

- Disturb channel
  Only functional with serial connections. With a special error injection device, these buttons can cause transmission errors on a serial connection to test error recovery procedures.

6.3.5 Direct message injection

At the bottom of the main window is an edit field that allows the entry of a message in human readable format.



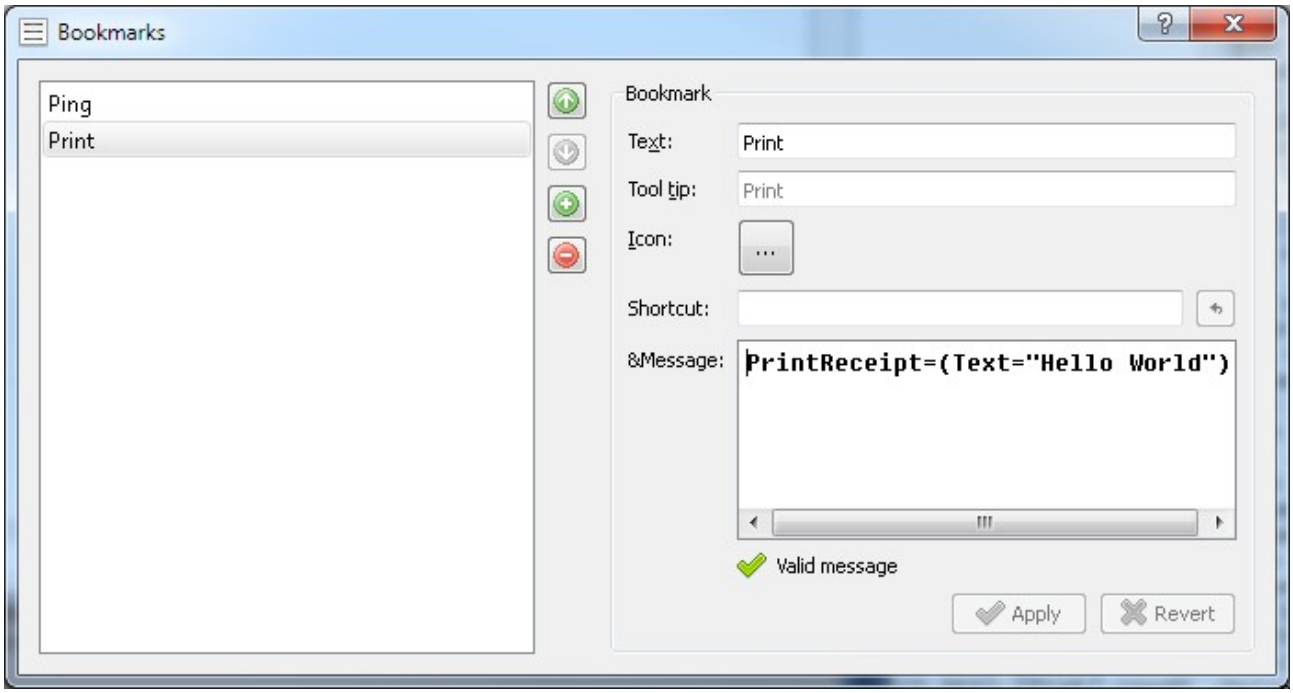
Clicking one of the two buttons on the right hand side will encode the message and send it to Connection 1 or Connection 2, respectively.

The icon to the left of the message indicates where the last message has been sent to. Hitting enter after typing in a message will always send to this connection.

6.3.6 Bookmark editor

The testing tool can keep a list of quick-access bookmarks. When clicking a bookmark, the message that was previously stored is sent through the connection. The bookmarks are typically shown to the side of the main menu. If they are hidden, get them back with *Bookmarks -> Show Bookmarks*.

To edit the bookmarks, chose the *Edit Bookmarks* from the *Bookmarks* menu.



The example on the previous page shows two bookmarks being defined. The Print bookmark is selected for editing.

Each bookmark has a name (*Text*). You can optionally enter a *Tool tip* text and chose an *Icon* from a graphics file.

A keyboard *shortcut* can be entered. When that shortcut is pressed in the main window, the bookmark is activated.

The message field takes the message formatted as human readable text. Of course, this can only work if a proper tag definition XML file has been loaded previously. As the message is being typed in, the tool checks the message interactively for correctness. Any error it finds is marked with a red squiggly line.

Once you're done entering a new bookmark, save it with *Apply*. The button *Revert* will undo the edits.

The arrow buttons in the middle can be used to change the order of bookmarks. Each arrow moves the selected bookmark in its direction.

The + and - buttons add and delete bookmarks, respectively.

Bookmarks must be saved to a bookmark file after editing using the *Bookmarks* menu and *Save Bookmarks* or Save Bookmarks As.

When the testing tool ist started, it will always try to open the last bookmark file that was used.

A bookmark that is being triggered is either sent to Connection 1 or Connection 2. The connection number is indicated by the icon at the bottom left of the window. This icon changes when using Direct Message Injection, see page 21.

6.3.7 Using the testing tool as a proxy

An interesting use during software development is to use the tool as a proxy between two applications. The tool logs the exchanged messages and helps finding bugs and gettings things done.

Lets assume a testing scenario with one client and one server. The client knows it must reach its server at IP 10.0.0.1, port 12001.

Client
IP: 10.0.0.50
Server IP: 10.0.0.1 Server Port: 12001

Server
IP: 10.0.0.1 Port 12001

Now lets put the testing tool, running on a PC with IP 10.0.0.20, inbetween.

Client
IP: 10.0.0.50
Server IP: 10.0.0.20 Server Port: 12001

Testing tool	
IP: 10.0.0.20	
Connection 1: TCP Server Port 12001	Connection 2: TCP Client Server IP: 10.0.0.1 Server Port: 12001

Server
IP: 10.0.0.1 Port 12001

Necessary steps:

- The actual client must be reconfigured to use the testing tool as its server.
- The testing tool's Connection 1 is set up as a TCP server where the client can reach it.
- The Connection 2 is set up as a TCP client that connects to the actual server.
- Make sure both connections are enabled (Their toolbar buttons are **not** red.)
- Also make sure the forwarding is enabled via the link button on the toolbar.

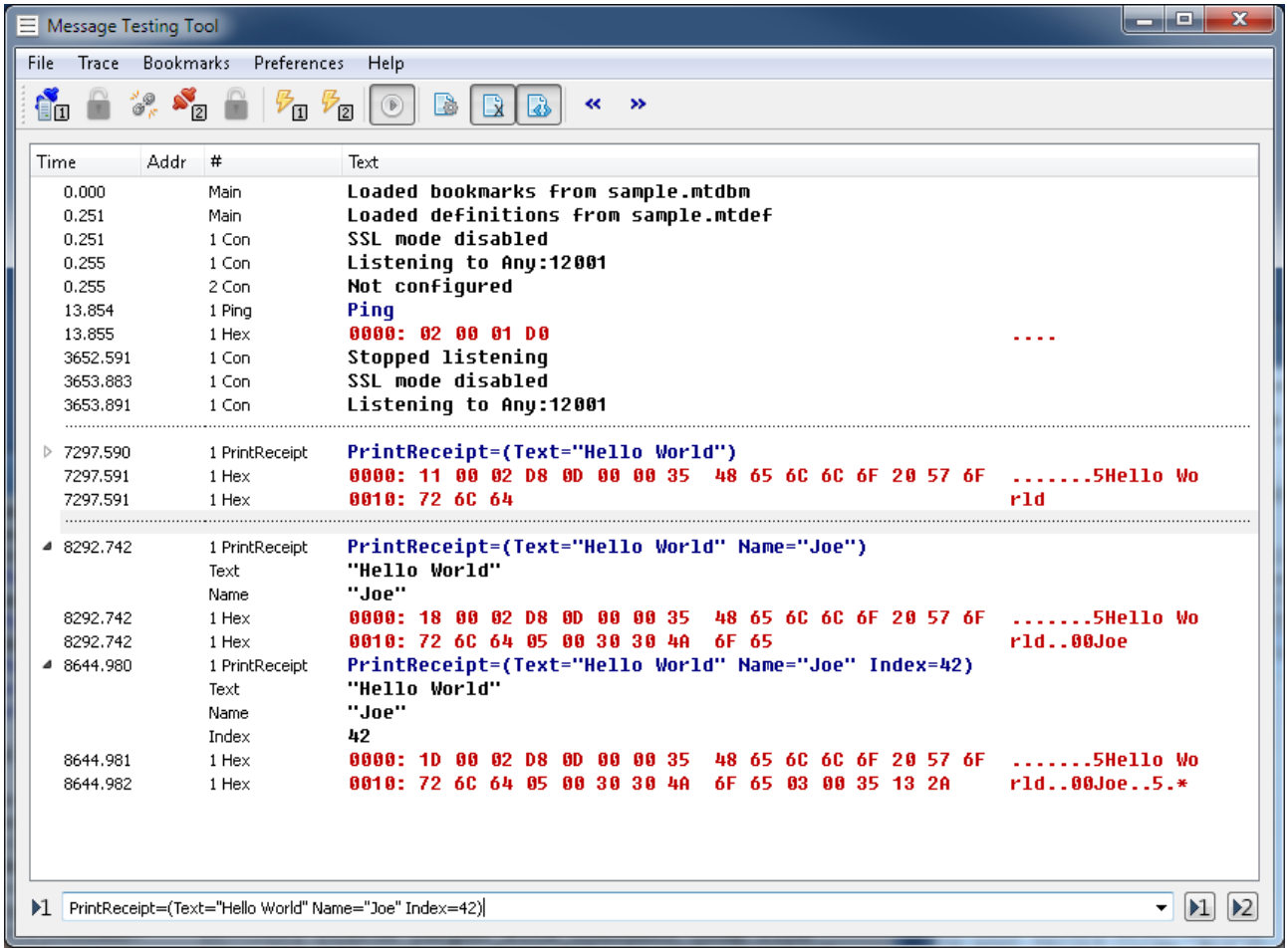
Once this has been set up, the testing tool will forward messages between both peers, while displaying the message contents.

It is even possible to inject messages in the data stream via the testing tool, using bookmarks or the direct entry at the bottom of the window.

Of course, it is often possible to run the testing tool on the same computer as one or both of the peers being traced. Simply chose different port numbers for both connections.

6.3.8 Tracing example

Here is another screenshot with a little more complexity.



The direct entry tool at the bottom has been used to enter a few different *Hello World* messages.

Hexdump has been turned on to display message contents. In a real world debugging sessions, the hexdump is most likely always turned off once the message encoding/decoding layers have been finished in the application. This follows the idea that the programmer has the least possible need to actually touch the numbers.

The second and third *Hello World* samples have been selected for detailed view with the little arrow to the left of the line. Detailed view displays each tag field on a separate line for clarity.

7. Appendix - Examples

This is the example definition file, repeated from page 8. On the following pages, the output of the code generator for several languages is shown.

```
<?xml version="1.0" encoding="UTF-8"?> <!-- -*- nxml -*- -->
<mtd16>
  <!--Command Messages-->
  <tag name="Ping" id="0xD001"/>
  <tag name="Pong" id="0xE001"/>
  <tag name="StatusReport" id="0xD800"/>
  <tag name="StatusReportResponse" id="0xE800"/>
  <tag name="PrintReceipt" id="0xD802"/>
  <tag name="PrintReceiptResponse" id="0xE802"/>
  <!--Generic tags -->
  <tag name="StatusCode" id="0x1000" comment="Result status code" display="4x">
    <enums>
      <enum name="Success" id="0x0000" comment="Indicate success"/>
      <enum name="Error" id="0x0002" comment="Generic failure code"/>
    </enums>
  </tag>
  <tag name="MachineStatus" id="0x7620" comment="General machine status.">
    <bits>
      <bit name="Online" id="0"/>
      <bit name="Enabled" id="1" comment="Station is open to the public"/>
    </bits>
  </tag>
  <tag name="Text" id="0x3500" comment="Generic text field"/>
  <tag name="Name" id="0x3030" comment="Any name"/>
  <tag name="Index" id="0x1335"/>
  <tag name="Type" id="0x133C"/>
  <tag name="Key" id="0x3335"/>
</mtd16>
```

7.1 C-Header

```
#ifndef MTD16_TAGS_H  /* Automagically generated -- do not edit */
#define MTD16_TAGS_H

#define MTD16DT_Binary           0
#define MTD16DT_Integer          1
#define MTD16DT_Bool             2
#define MTD16DT_String           3
#define MTD16DT_Date             4
#define MTD16DT_Time             5
#define MTD16DT_DateTime         6
#define MTD16DT_BitArray         7
#define MTD16DT_NetworkAddress   9

#define MTD16DT_List             12
#define MTD16DT_Request          13
#define MTD16DT_Answer           14
#define MTD16DT_Message          15

#define MTD16DT_ExtPoint         0x80
#define MTD16DT_ExtRect          0x81
#define MTD16DT_ExtSize          0x82

#define MSG_Ping                                    0xD001  /* Request  */
#define MSG_Pong                                    0xE001  /* Answer   */
#define MSG_StatusReport                            0xD800  /* Request  */
#define MSG_StatusReportResponse                    0xE800  /* Answer   */
#define MSG_PrintReceipt                            0xD802  /* Request  */
#define MSG_PrintReceiptResponse                    0xE802  /* Answer   */

#define TAG_StatusCode                              0x1000  /* Integer  */
#define TAG_MachineStatus                           0x7620  /* BitArray */
#define TAG_Text                                    0x3500  /* String   */
#define TAG_Name                                    0x3030  /* String   */
#define TAG_Index                                   0x1335  /* Integer  */
#define TAG_Type                                    0x133C  /* Integer  */
#define TAG_Key                                     0x3335  /* String   */

#define ENU_StatusCode_Success                      0x0000
#define ENU_StatusCode_Error                        0x0002

#define BIT_MachineStatus_Online                    0
#define BIT_MachineStatus_Enabled                   1
#define MSK_MachineStatus_Online                    0x01
#define MSK_MachineStatus_Enabled                   0x02

#endif
```

7.2 Lua

```
-- Automagically generated -- do not edit

mtd16.lutTag = {
  Ping                                                = 0xD001,    -- Request
  Pong                                                = 0xE001,    -- Answer
  StatusReport                                        = 0xD800,    -- Request
  StatusReportResponse                                = 0xE800,    -- Answer
  PrintReceipt                                        = 0xD802,    -- Request
  PrintReceiptResponse                                = 0xE802,    -- Answer

  StatusCode                                          = 0x1000,    -- Integer
  MachineStatus                                       = 0x7620,    -- BitArray
  Text                                                = 0x3500,    -- String
  Name                                                = 0x3030,    -- String
  Index                                               = 0x1335,    -- Integer
  Type                                                = 0x133C,    -- Integer
  Key                                                 = 0x3335,    -- String
}

mtd16.lutStatusCode = {
  Success                                             = 0x0000,
  Error                                               = 0x0002,
}

mtd16.lutBitMachineStatus = {
  Online                                              = 1,
  Enabled                                             = 2,
}
```

7.3 C++ Header

```
#ifndef MTD16_ENUMS_H  /* Automagically generated -- do not edit */
#define MTD16_ENUMS_H

namespace MTD16 {

  namespace Type {
    enum T {
      Binary          = 0,
      Integer         = 1,
      Bool            = 2,
      String          = 3,
      Date            = 4,
      Time            = 5,
      DateTime        = 6,
      BitArray        = 7,
      NetworkAddress = 9,

      List            = 12,
      Request         = 13,
      Answer          = 14,
      Message         = 15,

      ExtPoint        = 0x80,
      ExtRect         = 0x81,
      ExtSize         = 0x82,
    };
  };

  namespace Cmd {
    enum T {
      Ping                                          = 0xD001,
      Pong                                          = 0xE001,
      StatusReport                                  = 0xD800,
      StatusReportResponse                          = 0xE800,
      PrintReceipt                                  = 0xD802,
      PrintReceiptResponse                          = 0xE802,
    };
  };

  namespace Tag {
    enum T {
      StatusCode                                    = 0x1000,
      MachineStatus                                 = 0x7620,
      Text                                          = 0x3500,
      Name                                          = 0x3030,
      Index                                         = 0x1335,
      Type                                          = 0x133C,
      Key                                           = 0x3335,
    };
  };

  namespace StatusCode {
    enum T {
      Success                                       = 0x0000,
      Error                                         = 0x0002,
    };
  };

  namespace MachineStatusBits {
    enum T {
      Online                                        = 0,
      Enabled                                       = 1,
    };
  };

  namespace MachineStatus {
    enum T {
      Online                                        = 0x01,
      Enabled                                       = 0x02,
    };
  };

};

#endif
```

7.4 C++ lookup table

```
#ifndef MTD16_LOOKUP_H  /* Automagically generated -- do not edit */
#define MTD16_LOOKUP_H

namespace MTD16 {

  static const NameLookupTable nameLookupToplevelTags[] = {
    { "ToplevelTags", 0 }, // Name and type of the lookup table
    { "Ping",                                          0xD001 }, // Request
    { "Pong",                                          0xE001 }, // Answer
    { "StatusReport",                                  0xD800 }, // Request
    { "StatusReportResponse",                          0xE800 }, // Answer
    { "PrintReceipt",                                  0xD802 }, // Request
    { "PrintReceiptResponse",                          0xE802 }, // Answer

    { "StatusCode",                                    0x1000 }, // Integer
    { "MachineStatus",                                 0x7620 }, // BitArray
    { "Text",                                          0x3500 }, // String
    { "Name",                                          0x3030 }, // String
    { "Index",                                         0x1335 }, // Integer
    { "Type",                                          0x133C }, // Integer
    { "Key",                                           0x3335 }, // String
    { 0, 0 },
  };

  static const NameLookupTable nameLookupStatusCode[] = {
    { "StatusCode", 1 }, // Name and type of the lookup table
    { "Success",                                       0x0000 },
    { "Error",                                         0x0002 },
    { 0, 0 },
  };

  static const NameLookupTable nameLookupMachineStatus[] = {
    { "MachineStatus", 2 }, // Name and type of the lookup table
    { "Online",                                        0 },
    { "Enabled",                                       1 },
    { 0, 0 },
  };

  static const TagLookupTable tagLookup[] = {
    {    ~0U, nameLookupToplevelTags                     },
    { 0x1000, nameLookupStatusCode                       },
    { 0x7620, nameLookupMachineStatus                    },
    { 0x7620, nameLookupMachineStatus                    },
    { 0, 0 },
  };

};

#endif
```

7.5 C#

```
public enum TagID
{
  Ping                                          = 0xD001,
  Pong                                          = 0xE001,
  StatusReport                                  = 0xD800,
  StatusReportResponse                          = 0xE800,
  PrintReceipt                                  = 0xD802,
  PrintReceiptResponse                          = 0xE802,

  StatusCode                                    = 0x1000,
  MachineStatus                                 = 0x7620,
  Text                                          = 0x3500,
  Name                                          = 0x3030,
  Index                                         = 0x1335,
  Type                                          = 0x133C,
  Key                                           = 0x3335,
}

public enum StatusCodeEnums
{
  Success                                       = 0x0000,
  Error                                         = 0x0002,
}

public enum MachineStatusBits
{
  Online                                        = 0,
  Enabled                                       = 1,
}

public enum MachineStatusMasks
{
  Online                                        = 0x01,
  Enabled                                       = 0x02,
}
```