Writing a Validator for TMATS

Bryan Kelly
83 FWS/D04
Telemetry Systems Engineer
Tyndall AFB, FL
irig@bkelly.ws

Abstract

Applications that use TMATS benefit from the ability to presume that the TMATS data is well constructed. This need is met by a TMATS validator. Some classified systems need source code rather than an executable to avoid expensive testing before being allowed in. An Open Source Validator is proposed, presented and made available to the public. Major points and difficulties are discussed.

The source is available in a Visual Studio 2008 project here: www.bkelly.ws/irig/validator.html
A bulletin board for TMATS / Chapter 10 discussion is here: http://www.bkelly.ws/irig_106/

Key words

TMATS Validator "Open Source"

Purpose

This paper introduces and describes an application to validate TMATS. This application remains under continuing development and is planned to gain the ability to read the Chapter 10 data, output selected parameters. It will provide the user the ability to edit TMATS data fields and write a new TMATS or Chapter 10 file with the update.

It does not describe new or unusual concepts or techniques. Rather it provides a starting point and a collaboration point for software / telemetry engineers. A primary goal is to share the source code and help the telemetry community along the path to better applications.

Purpose Two

This paper describes a format for the TMATS definition that is easily readable by humans and applications. The format consists of two files: a) a spreadsheet version of the essential TMATS standards, and b) a CSV file created by exporting the spreadsheet version. These are described in this document and provided to the reader. This format is hereby submitted as a candidate standard for an application readable version of the TMATS standard.

Audience

This paper is written for the intermediate to advanced level programmer. In the interest of brevity routine code is not discussed. All readers are invited to register on the bulletin board to ask questions and post suggestions. The reader is expected to be familiar with TMATS and Chapter 10 but detailed knowledge is not required.

Platform

This project uses Windows 7, Visual Studio 2008/12, C++, and an MFC template. The worker code is separate from the GUI code and should be relatively easy to import into other development environments. All the worker code has been separated from the project directory and is found within the zip file in the directory COMMON_CODE. Microsoft Visual Studio has a major quirk when using **Additional Include Directories**. The reader **must read** the notes supplied with the project before the build will be successful.

In the COMMON_CODE directory are some utilities not fully described here. The first is class C_Log_Writer. This handles all the logging utilities and has some nice features. There are also some classes for TCP/IP communications. Both are out of scope for this paper, but the reader can register with the bulletin board and discuss them. All code is declared **Open Source** per the GNU license and is available for use by all, subject only to that open source license requirement.

Glossary

The following phrases are used in this document. These are the author's definition and the scope is limited to this document and to the validator project.

The Definition: Refers to any entity that contains the definition of the TMATS records. Depending on the context, it can refer to the Excel worksheet, the CSV file, or the structure within the software application containing the TMATS definition. In all cases, this is the information that specifies how the TMATS data is to be formatted. Note: The definition file presented here was derived from the TMATS standard and is my interpretation. All disagreements with the standard are to be attributed to my error and the standard prevails.

The Definition Map: A structure that contains the TMATS definition within the application. This is the data that the TMATS Data is validated against. Please note and differentiate between the terms **definition** and **data**.

The Data or the **TMATS Data** or the **Data Map**: Refers to the text based data read from the TMATS file or from the TMATS data that is the header to Chapter 10 data. That data is also stored in a map.

STD::MAP: Each of the two maps is an instance of the class more formally named: STD::MAP. This class is from the **Standard Template Library** available in most C++

compilers and development environments. The use of classes such as this can significantly reduce the programmer's burden in storing and retrieving data. Engineers not familiar with the Standard Template Library are strongly advised to research this topic.

Group: The TMATS definition is divided into groups. These are the same groups defined in the TMATS standard: G, T, R, etc.

Identifier: Each definition begins with the group letter, and eventually followed by a phrase called the identifier. The identifier phrase specifies the information found in each TMATS records. The very first definition in the TMATS document is G\FN:my name; The group is G and is token number one. The backslash is token two. The identifier is FN and is token three. The first definition of the Transmission Group is T-x\ID: The group is T and the identifier is ID.

Enumerators: Most definitions contain phrases of the general format: -x, -n, etc. They are referred to collectively and individually as enumerators. The dash is a separator and the value after the dash is the enumerator.

Payload: The ASCII characters between the colon and the semicolon.

Application Readable Definition File

An application that validates a TMATS document must be flexible. Having the definitions in an easily updatable file is essential.

The TMATS specification is distributed via PDF files. A PDF file is not easily accessible from the average software application. A format that is easy for both humans and applications is needed.

The formal TMATS definition file for this project is a CSV or Comma Separated Value file. A CSV can be a bit tedious to read and edit so Microsoft Excel is used for viewing and editing. The final product is created by exporting the worksheet to a CSV file. The filename extension is TMAD with the D on the end representing **definition**.

The definitions are, currently, hand loaded into an Excel workbook then exported to the CSV file. This process is tedious and error prone. Suggestions or utilities to automate it are solicited and will be shared with all with full credit given to the source.

Begin with a view of the first few rows of the Excel worksheet.

Figure 1 Excel Worksheet A through P

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Token 1 Group	Token 2	Token 3	Token 4	Token 5	Token 6	Token 7	Token 8	Token 9	Token 10	Token 11	Token 12	Token 13	Token 14	Token 15	Token 16
INFO	BK	Rev	1	Ver	1										
INFO															
COMMENT	:	txt	;												
G	\	PN	:	txt	;										
G	\	TA	:	txt	;										
G	\	FN	:	txt	;										

Figure 2 Excel Worksheet Q through T

Q	R	S	T
Token 17 Max Length	Token 18 Required	Token 19 Repeats	Token 20 Display comment
16			Name of Program
64			Test Item Description
256			Name of this file

Row Descriptions

Row one is the header, rows two and three are user added information and the definitions begin with row four.

Table 1 describes the columns of the worksheet. When the CSV file is read into the application each line will be broken into tokens. Referring to the above Excel view, each cell within a row becomes a separate token and each row a record. The tokens are stored in an array of CStrings. The application uses a set of constants to access that array and those constants are given the same names as in the header row. Each column header and constant contains the token number. In turn each record (an encapsulation of the array of CStrings), is stored as a single entry in the definition map.

Table 1 Column Descriptions

Column	Header	Description
A	Token 1 Group	The TMATS group.
B	Token 2	The first separator, it can be a backslash or a dash.
C … N	Token 3 through 16	After token 1the meaning of each token changes according to the identifier. Consequently, they do not have specific names. At this

Column	Header	Description
		writing sixteen tokens is sufficient. That may change as I complete the process of creating the first version of the definition file.
O	Token 17 Max Length	The maximum length of the payload section.
P	Token 18 Required	A "Y" signifies this identifier must be present, otherwise not. There are some conditions which require the presence of additional tokens. Those conditions have yet to be resolved.
Q	Token 19 Repeats	Describes how identifiers are to be repeated within a TMATS document.
R	Token 20 Display Comment	Text that can optionally be displayed by a reader application to help the users in understanding each identifier.

As the process of validating the TMATS data is discovered and the code written and tested, I anticipate that a few more columns may be added to make the validation process easier to write.

Caution: At this writing I am not comfortable with my understanding of all the ways an individual item might be repeated and how each enumeration will work out. This early version does not validate the value of the enumerator, just that they can be converted to numbers.

After the header are a few rows of interest.

Figure 3 Rows 1 ... 5

	A	B	C	D	E	F
1	Token 1 Group	Token 2	Token 3	Token 4	Token 5	Token 6
2	INFO	BK	Rev	1	Ver	1
3	INFO					
4	COMMENT	:	txt	;		

Row two indicates that this is **B**ryan **K**elly's definition file, Rev 1, Version 1. It is informational only. Future versions will check this data and advise the user if the version of the definition file does not match the version of the application. If you make significant changes you may edit the next row with your information. The validator ignores rows one through three. Row four contains the first definition, a comment.

Figure 4 First Few Definitions

4	COMMENT	:	txt	;		
5	G	\	PN	:	txt	;
6	G	\	TA	:	txt	;
7	G	\	FN	:	txt	;
8	G	\	106	:	txt	;

Figure 4 shows the first few definitions. Row 4 is too simple so begin with row 5. The definition from the TMATS document is broken out into tokens with each token in a separate cell. In this example, according to my interpretation, columns A through D and column F specify the exact contents of what is to be contained in the TMATS data.

Column E describes the payload of the TMATS data. That payload can contain any text, subject only to the maximum length, excepting the colon and semicolon.

The TMATS specification uses upper case for pre-designated phrases that can be put in the payload. When the payload does not have a pre-designated set of options, or when those options can be replaced by a described field, this definition file uses lower case letters to specify the type of data that may be contained in the TMATS data. **Figure 5** and in particular, row 296 shows an example. This TMATS payload can contain the exact characters "NO" or it may contain an integer in ASCII text format. This is author's interpretation of the standard and how it is incorporated into this definition file.

Figure 5 Selected Definitions

294	P	-	d	\	SF	\	N	-	n	:	int	;	
295	P	-	d	\	SF1	\	N	-	n	-	m	:	txt ;
296	P	-	d	\	SF2	\	N	-	n	-	m	:	NO\int ;
297	P	-	d	\	SF3	\	N	-	n	-	m	:	FI\EL\NA ;

Build The Map

Now that an application readable form of the TMATS standards has been created, the next step is to read those definitions into the application and make it available for access in validating the TMATS data. This section jumps into the code concepts and the process of building the map from the definition file. After downloading and unzipping the project the reader will find a class named: **C_tmats_definitions** in directory COMMON_CODE. This class handles all the chores related to the definitions. It reads the definition file, parses it, and stores the definitions in the map.

The main application calls a single method named: **Build_TMATS_Definition_Map()**. That method handles all the tasks necessary to build the map. This is a simple method with two worker lines of code: **Open_TMATS_Definition_File()** and **Populate_Definition_Map**(). Each is well named and does just what the name says. The open operation is skipped and the discussion proceeds to populating the map.

Important Declarations

The following structure is declared to contain each tokenized line of data read from the file and to be stored in the map.

```
typedef struct
{
  CString      token[ NUMBER_OF_TOKENS ];
}    td_tmats_definition;
```

It is a simple array of CStrings. The map and an iterator are declared as shown below. The map key is a CString that is constructed in the code.

```
std::map < CString, td_tmats_definition > m_tmats_definition_map;
std::map < CString, td_tmats_definition > ::iterator m_definition_iterator;
```

The Code

The module begins by reading the informational text at the beginning of the definition file. Following that is this loop that reads the remainder of the definition file, one line at a time:

```
while( m_definition_file.ReadString( m_one_definition ) )
{ … }
```

The first worker line in that loop is just below:

```
Tokenize_One_Definition_Line( m_one_definition, &m_one_definition_record );
```

Method Tokenize_One_Definition_Line

Here is the core code:

```
1)   m_C_Get_CString_Tokens.Load_Strings( new_definition_line, "," );

2)   for( int token_counter = 0; token_counter < NUMBER_OF_TOKENS; token_counter ++ )
     {
3)     p_definition_structure->token[ token_counter ].Empty();

4)   m_C_Get_CString_Tokens.Get_Token( &p_definition_structure->token[ token_counter ] );
     }
```

Line 1 line initializes the tokenizer class with the definition data. Line 4 fetches the tokens from the class one at a time. Notice that the argument of line 4 is the address of one CString. Specifically, one entry in the array of the structure declared earlier. The method to get each token puts the token directly into the CString array.

Class C_Get_CString_Tokens

Class **C_Get_CString_Tokens** is a very simple tokenizer. The CString class has a built in tokenizer, but it skips over empty tokens. That does not work for this application. My searches found several tokenizer classes, but none directly discussed the need to recognize all tokens. The requirements were very simple so a tokenizer class was written rather quickly. It was written specifically for CStrings. It is initialized by passing in a CString and the separator character. Each call to **Get_Token()** extracts one token and writes it into the CString passed by address. Separators with no data return an empty string and the function returns false when out of tokens.

The reader will note that this for loop makes a major presumption: There will be NUMBER_OF_TOKENS in the CString. (That constant is defined in the project.) When Excel exports the definition worksheet to a CSV, there will always be a separator and a place for each column used in the spreadsheet.

Method Populate_Definition_Map()

Returning to the while loop in this method, there is some code to detect the definition data defining the group and the location of the key. Skipping that we find these worker lines:

```
status = Build_Map_Key_Using_Backslashes(  );
if( status )
{
  m_tmats_definition_map[ m_map_key ] =  m_one_definition_record;
}
```

The method is called to build the map key in member variable m_map_key. Within that method is a check for a duplicate key. If a duplicate is found, the erroneous definition is logged and is not loaded into the map. The entire record is stored in the map using that key. Remember that this is the STD::MAP declared earlier and one definition record boils down to a simple array of CStrings.

Method Build_Map_Key_Using_Backslashes( );

The writing and testing of this method revealed some difficulties in preparing the TMATS definitions.

The key is constructed by concatenating two components: The group letter and the identifier. There is a problem in that the identifier is not always in the same token. Group G has the identifier in token 3 while Group T has it in token 5.

After writing code to locate the moving identifier, there remained many duplicated keys. A unique key for each definition is essential. Going directly to the solution, I discovered that a unique key can be produced by concatenating three phrases from the definition: the group, the first phrase after a backslash, and if there is a second backslash before the colon, include that phrase.

After constructing the key in this manner, then logging all the records to a text file to see how the code performed, the key was discovered to be rather difficult to read. That was resolved by including the backslashes in the key. This provides a significant improvement in readability and aids in code development.

Standard Change

The standard has many definitions. An automated validator is essential to full acceptance in the telemetry world. Automated tools such as a validator are much easier to write, verify, and maintain when the standard is consistent within itself. The TMATS standard is lacking in this category. The identifier starts in column three, then shifts to column five, and in many definitions, requires the addition of a third token to be complete. My suggestion is that the standard should incorporate an explicit rule to the effect:

The first character of each definition shall be a single character. (Two would be okay.) That will be followed by a backslash. The next field will be an identifier phrase that is unique within each group. The identifier should always be presented before any enumerators. Recognizing that a second field used as a modifier makes for easier to read definition, keep the option of a modifier, but have it immediately follow the group in a standard location rather than bouncing about within the definitions.

This means that the dash x and the dash d should be moved to follow the identifier and modifier rather than precede it. There are additional comments on the standard on my web site, but they are out of scope for this paper.

All said, the 2013 standard has been published and it is what it is. The validator must deal with it.

Definition Summary

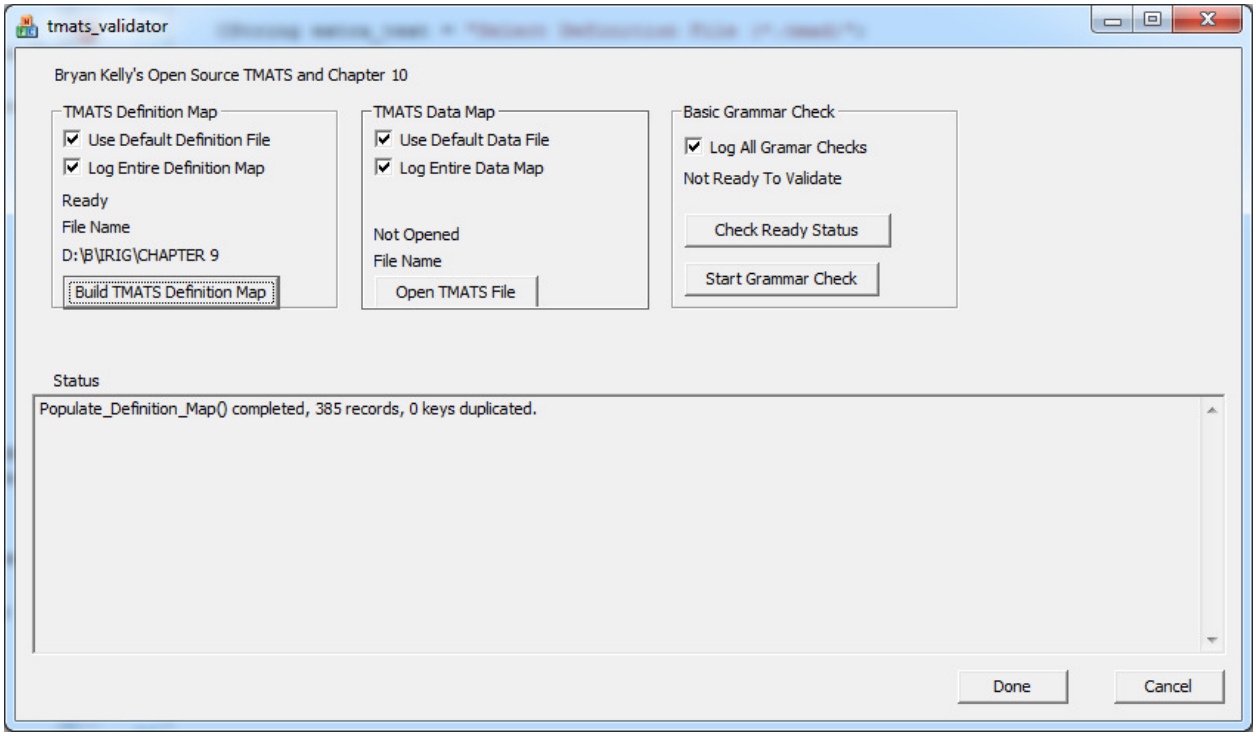
In summary, the class to read the definitions does the following:

1. Select and open the definition file
2. Read each token
    a. Each cell in the Excel workbook, within a row, is one token.
    b. Each row is a new definition.
    c. Blank cells are significant and are retained.
3. Put each token into the proper location within an array of CStrings.
4. Build a unique key for the map consisting of:
    a. The group.
    b. First backslash. (Must be present)
    c. The phrase after the backslash.
    d. The second backslash, before the colon, if it exists.
    e. The phrase after that second backslash.

5. Load the CString Array into the map using the key.

Here is an image of the validator dialog after the definition file has been read.

Figure 6 Dialog After Building Definition



Note the two checkboxes in section TMATS Definition Map. The first, **Use Default TMAD file**, provides the ability to use a hard coded file name. This significantly speeds up operations during development. The file name is currently declared as: const CString DEFAULT_DEFINITION_FILE = "E:\\BRYAN\\CHAPTER 9 TMATS\\tmats_definition_workbook.tmad";

Each developer is expected to change this as desired. When the checkbox is disabled a standard open dialog will be presented to select your favorite file.

Note that the log file defaults to C:\LOG_FILES. Create that directory before running the application or edit the constant and create it in the location of your choosing.

The second check box, Log Entire Definition Map, causes the class to start a new log file, log each definition, then start a new log file. The definition log file will be named to indicate its contents.

Looking ahead, the same two check boxes are defined in the group to read in the TMATS data.

Note that at this writing, many of the definitions have yet to be entered into the definition file. This is done by hand and is a rather tedious process that should be complete soon. Also note that this project is under development and that the very nature of **Open Source** implies that updates will be on going.

Build The Data Map

The next step in this validator project is to read the TMATS data and load it into a map. This map will be similar to that of the definitions.

Please note that I have endeavored to differentiate and appropriately use the terms **definition** and **data**. Please reference the glossary above.

The TMATS data is handled by class: **C_Tmats_data**. It uses a very similar process to read the TMATS data and load it into a map.

Begin with a look at header file **C_Tmats_Data_Declarations.h** The constants used to access the array of CString have been extracted out to an include file along with the typedef of the record that holds each token and is loaded into the map.

Reading the data and building the map is done with a single call to method: **Build_TMATS_Data_Map()**. There it opens the data file and calls **Populate_Data_Map()**.

An immediate difficulty is discovered. While the standard states that TMATS is to contain only seven bit ASCII characters, the example chapter 10 files I have perused contain some amount of prefix data that is not seven bit ASCII. Since this application is destined to deal with chapter 10 files as well as pure TMATS files, this data must be managed.

At this time, lacking information about the contents and format of that lead-in data, the data class simply looks for valid TMATS data and ignores all the non TMATS data. So far, with very limited exposure to data files from multiple vendors, this is satisfactory.

The majority of the code in method Populate_Data_Map() is dedicated to finding the beginning and the end of the TMATS data. Look for the comment:

// Begin the section to process each record of TMATS data.

This marks the beginning of the section that actually ingests the TMATS data. There are essentially two lines of code that do all the work. Obviously these are methods containing much code. The first is:

```
m_C_Get_CString_Tokens_And_Separators.Tokenize_The_Data(
  mp_C_Log_Writer,
```

```
m_one_line_of_data,          // the data read from the TMATS data file
TMATS_DATA_SEPARATORS,       // The separators found in a TMATS record
&m_one_data_record );        // The tokenized data
```

This method parses each line of TMATS data, tokenizes it, and loads it into the structure: **m_one_data_record**, an array of tokens (CString) very similar to that used for the definitions. The first 16 tokens are functionally identical to the **definition** structure with a one to one match. Each is in an array of strings and **data.token[N]** for data is validated with **definition.token[N].**

The next worker is just one line of code:

m_tmats_data_map[data_key] = m_one_data_record;

Several things about these worker lines of code bear explanation.

Note that the first is a class and a method within that class. The TMATS **definitions** as formatted for this application are somewhat precise and easy to parse. Each token is separated by a comma and those separators are discarded. The separators of the TMATS **data** are an essential part of each record and each is saved as a separate token. The differences are sufficiently different to require a unique tokenizer.

Go back to header file C_Tmats_Data_Declarations.h and note constant: TOKEN_DATA_24_DEFINITION_KEY. This tokenizer builds the same key as used by the definition map and stores it along with the record of data. As the tokenizer processes each character of the data, it arrives at this line of code:

```
if( add_to_key )
        {
          tokenized_data->token[ TOKEN_DATA_24_DEFINITION_KEY ] += current_char;
        }
```

Referring back to the discussion of the definition and its map, each character that looks like definition key material is copied into a token reserved for that key. After the record is stored and the process moves on to validation, all the information is present within each data record to find the appropriate definition.

The map for the data uses an integer for the key. It increments 0 to N and provides the ability to sequence through the data map in the same order as the TMATS data in the file. The order of the **definitions** is not critical while the order of the **data** is.

Tokenizing the data records requires multiple if statements, but is relatively straightforward. The two major classes presented so far each perform one simple task: tokenize the **definition** and the **data**. Then they store that information in their own map.

During this tokenization process it would be easy to perform some amount of basic validation of the data. However, that would cause the code that tokenizes the data to begin to serve two masters. It is better to stay with a single purpose and keep it conceptually and technically as simple as possible. With that, we continue on to the next phase, validation.

Basic Validation

I have reached this point in development of the application as the deadline for this paper approaches. The validator presented here is decidedly not comprehensive. This is partly due to time and partly due to no previous experience writing validator code. Work will continue and the latest code may be found via the bulletin board and support pages previously noted.

The validator is encapsulated in class **C_TMATS_Basic_Grammar_Checker**. The method **Perform_Grammar_Check()** initiates the process.

After the definitions and the data have both been loaded into their respective maps, the data can be validated against the definitions. By way of quick review, the definitions have been loaded into a definition class. That class can provide the definitions with random selection using a CString key. The data has been loaded into a similar class that has sequential access rather than random access. It can also provide the data records one at a time.

Since all the information has been loaded into maps, the top level of the validator is rather simple. A while loop iterates through the data records one at time in the same order they were read from the data file. After acquiring each data record, it uses the pre-stored key to fetch the definition record. It then calls a method to conduct the validations:

```
Compare_Definition_And_Data(
        &one_definition_record,
        &one_data_record  );
```

This method logs the results and prepares a notice that can be displayed in a dialog so return values are not necessary.

Method **Compare_Definition_And_Data()** uses a simple FSM (Finite State Machine) and processes the characters of the data record one at a time. The states are declared locally and on entry we are guaranteed that the first character is the group. We are also guaranteed that the group and all the parts of the identifier match the definition; otherwise the key prepared earlier would not have found the definition record. Nonetheless, for symmetry and simplicity, all fields of the TMATS data are verified using the definition.

The data record, an array of CStrings organized as one string per token, is validated one token at a time. Many of the tokens are validated with a simple equality compare against the definition record. The one-to-one match between the two records for the token entries makes this trivial.

This type comparison is valid for all the tokens except the enumerators themselves and the payload. Those get a bit more complicated.

By taking careful note of the contents of each token the FSM can determine the type of token being evaluated. Once the state is set a simple switch selects the appropriate code section to validate each token.

In this first validator the enumerators are simply scanned into an integer. When the conversion is successful the field is declared valid. In later versions the various enumerators will have their values tracked and checked against other enumerators for proper order. Check the web site for the latest version.

The payload is significantly more difficult and requires its own method: Check_Value_Field(). This method is currently being tested but here are the fundamental concepts.

There are two distinct types of payload values. First is a defined set of options. For example look at the stand for: P-d\SYNC1. This description is found in the standard:

THIS SPECIFIES THE DESIRED CRITERIA FOR DECLARING THE SYSTEM TO BE IN SYNC:
FIRST GOOD SYNC – 0
CHECK - NUMBER OF AGREES
(1 OR GREATER)
NOT SPECIFIED - 'NS'

The allowed values appears to be "0" for first good sync, "NS" for not specified, or a counting integer. The payload may contain one of two predefined options, or may contain an integer, in text format of course. In the definition file each option is separated by a backslash. The token in the definition record contains: **0\NS\dec**. The result is that the grammar checker needs its own tokenizer for the payload. As noted earlier, the **dec** in lower case, describes the allowed payload rather than prescribing the exact characters, and is recognized as such by that characteristic of being presented in lower case.

This definition token is parsed and each phrase is compared against the data token for an exact match. If the lower case phrase is found before a match is recognized, the format of the allowed values is determined. If "text" then there are really no restrictions. If a number, the type of number is determine, int, decimal, float, etc, and the payload is converted to that type. If no error is found, the payload is validated.

The deadline has made its approach and the paper must be finalized. The validation process stops with this set of basic checks. Some advancements may be incorporated by conference time so please register to the bulletin board and check the web site.

Conclusion and Summary

At this point a format for an application and human readable version of the TMATS definitions has been created and proposed as a candidate for a standard document.

A validator application has been created and is functional through the first level of validation.

Current abilities:

1) Read the new definition document.
2) Store each record in a definition map.
3) Read the TMATS data.
4) Store each TMATS record in a data map.
5) Sequence through the TMATS record one at a time.
6) Fetch the appropriate definition record.
7) Perform a basic validation for each data record.

Some of the features:

1) The ability to log each definition record and data record for evaluation.
2) The ability to log details of each validation step.
3) A separate logging utility in a class that may be used by your projects. There are a couple of nice features in this class.
4) Some asynchronous TCP/IP classes, not used in this application, but available in the COMMON_CODE directory and written for Windows and Visual Studio.
5) The worker code is encapsulated in classes that are outside of the Visual Studio GUI classes and hopefully will be easy to port to other platforms.

Desired updates include:

1) A more exhaustive validation.
2) The ability to edit fields within this application and write a new TMATS or chapter 10 files.
3) A new set of TCP/IP classes that use the Windows API directly. The current version uses Microsoft wrapper classes and is too slow for the high data rates needed for telemetry data. I found that out the hard way and had to revert to blocking TCP/IP calls. They are significantly faster.
4) This list is already a bit ambitious and therefore ends here.

This application is declared Open Source in accordance with the GNU open source license. It is available via the bulletin board and the web site noted earlier. It is being written primarily on my own time and made available via my personal web site. This application and paper are being created as a learning tool, as a utility needed by the engineering team where I work, and hopefully, will be found useful at other sites.