IEEE 1451.4 Smart Transducers Template Description Language

Charles H. Jones, PhD
412TW/ENTI, Edwards Air Force Base

Abstract
The IEEE 1451.4 smart transducer interface standard [1] provides a mechanism for both analog and digital, or mixed mode, interfacing to sensors and actuators, which are collectively referred to as transducers. The analog mode allows normal interfacing to the transducer. The digital mode is intended to provide the 'smarts' by allowing the transducer to provide basic information to the application system. This information is referred to as the transducer electronic data sheet (TEDS) and contains information ranging from serial number to calibration data and electrical characteristics. A major driving force behind the development of the standard was to minimize the amount of memory required to store a TEDS; with a stated objective of only needing 256 bits, although more are allowed. This requires a method of mapping the bits in a precise fashion. This bit mapping is accomplished through templates which are text based files written in the template description language (TDL). The TDL is a formal language similar to programming languages, but with considerably less looping and conditional control. This is because the entire purpose of the language is to map bits and not to implement general processing or mathematical capabilities. This paper outlines the functionality and syntax of the TDL.

Introduction
The use of smart transducers will greatly increase the plug-and-play nature of data acquisition systems. This will increase overall flexibility by allowing easier and potentially dynamic configuration. The ability to easily attach sensors and to query them for basic information will also make installation, pretest setup, and maintenance faster and more efficient.

Perhaps the most fundamental ability that makes a smart transducer 'smart' is its ability to provide basic information about itself. This implies an ability to store this information. The IEEE 1451.4 standard[1] addresses making a transducer smart while still maintaining the ability to receive its analog signal directly. This is the reason for the phrase 'Mixed-Mode Communication' in its title. This mixed mode ability usually implies the transducers are being used in applications where it is not desirable to place a large digital or computerized component collocated with the transducer. As such, a primary tenet of the IEEE 1451.4 working group has always been to minimize the impact of adding intelligence. Specifically, the working group has attempted to require no more than the addition of a 256-bit (yes, bit, not byte) memory and necessary communications electronics; although more memory can be

added if the manufacturer wants to. The information kept in this memory is, as with the entire IEEE 1451 family, referred to as a TEDS.

In order to accommodate such a small memory and still be able to contain significant information in the TEDS, it is necessary to provide a very precise definition of what every bit means. Since every transducer will need to store different types of information (e.g., a temperature sensor is described with different properties than an accelerometer), a mechanism for describing the TEDS for each individual transducer must be provided. The adopted mechanism is referred to as a *template*. A template is a text file that provides a series of commands describing a TEDS bit by bit. The language the templates are written in is the TDL.

The TDL encapsulates a large portion of the software needed in any IEEE 1451.4 implementation. In order to be able to read data from a transducer, a system must be able to parse a template written in TDL. This paper describes the TDL, although it does not cover every command. The TDL is a language in the formal sense. However, it is important to realize that the TDL is not a programming language in the way C++ or FORTRAN is. For example, it does not have an If-Then-Else ability. This is because the TDL has one purpose: to define bits in a TEDS. It is not intended to provide the ability to calculate or implement general logical control.

Basic Structure

A template begins with a Template command and ends with an EndTemplate command. More than one template can be in a single text file. The file as a whole ends with a ValidationKeyCode command that provides a basic checksum for the file. A template command has the following syntax:

Template *<manufacturer code>, <ID number of bits>, <Template ID>, <title>*

The *<manufacturer code>* is an integer that defines who the manufacturer is that wrote the template. Manufacturer codes are maintained and allocated by the working group. The manufacturer code of 0 indicates an IEEE defined template. Every template has an identification unique within the manufacturer code. That is, the combination of a manufacturer code and template ID uniquely defines a template. The manufacturer code is read from the TEDS prior to starting the parse of a template. (The full bootstrap process is outside the scope of this paper but is described in Clause 6 of the standard.) The *<ID number of bits>* identifies how many bits in the TEDS make up the *<Template ID>* and represents the first bits of the TEDS mapped by the template. The *<title>* is a quoted string describing the template.

After the Template command is the TDL_Version_Number command, which is self-explanatory. There are no other required commands. Between the TDL_Version_Number and EndTemplate commands are all other commands used to define the bits in the TEDS. The most important commands are the property commands, which define the information describing the transducer. There are also

control commands and a variety of other commands that allow additional functionality. Figure 1 shows a simple template written in TDL.

```
TEMPLATE 0,8,25,"Example template"
   // Comments are preceded by two back slashes.
   TDL_VERSION_NUMBER 2
   PHYSICAL_UNIT "Hz",(0,0,0,0,0,-1,0,0,0,0,1,0)
   // Frequency: Hertz = 1/second
   %Reffreq, "Reference frequency", CAL, 6, ConRelRes,
   7.9, 3.26, "0p","Hz"
   ENUMERATE DirectionEnum,"x","y","z"
   %Direction, "Sensitivity direction (x,y,z)", CAL, 2,
   DirectionEnum, "e", ""
EndTemplate
```

Figure 1 – A simple template.

Property Commands

Property commands are the core of templates and everything else simply provides a support structure to specify which bits define the properties describing the transducer. Although there are other forms described in the standard, the basic syntax of a property command is:

%<property_tag>,<"description">,<access_level>,<data_type>,<format>,<physical_unit>

All property commands are prefixed with a percent sign, '%.' Figure 1 contains two property commands with *<property_tag>*s of Reffreq and Direction. As of the time of writing, Annex B of the standard identifies and defines 88 reserved properties. These characteristics are broken into four major categories: sensitivity and mapping properties, electrical signal properties, calibration properties, and miscellaneous properties. For the purpose of this paper it is enough to understand that these properties define characteristics of or information about transducers. For example, the %CalDate property indicates the date of the last calibration of the transducer. Each property command defines how many bits to read from the TEDS and the data type of the bits read. Working through the fields in the command: The *<"description">* is a quoted string describing the property. The *<access_level>* identifies whom has permission to change the data in the TEDS related to this property. The *<data_type>* defines how many bits to read and the data type of the data for this property. The *<format>* is a suggested way of displaying the data read from the TEDS for this property. The *<physical_unit>* defines the physical unit of the data associated with this property. The only field required other than the **<property_tag>**, is the *<data_type>*. Omitted fields must still be delimited by commas.

Part of the *<data_type>* is a *<number_of_bits>* field, which defines how many bits are read from the TEDS. There are twelve data types identified. Many of them are standard enough not to need much explanation. For example, UnInt is an unsigned

integer. However, an interesting twist is that it can take up any number of bits – as defined in the *<number_of_bits>* field – which helps illustrate what levels of effort were implemented to minimize the number of bits in the TEDS. For example, if you only need the integers from 0 to 3, then you only need to take up 2 bits in the TEDS. Two of the floating-point data types illustrate the attempt to minimize bits even further and need some thought and consideration to understand.

The floating point types of ConRes and ConRelRes require two parameter fields: *<start_value>* and *<tolerance>*. These data types map a set of integers onto a real interval. The *<start_value>* is the lower bound of this interval. The *<tolerance>* can be thought of as the granularity. The <number_of_bits> field determines the upper bound of the interval. At this high level of description, this is no different than standard floating point or integer types. For example, a standard 8-bit integer has a start value of -128 and a tolerance of 1. In fact, the ConRes type can be used to define such an integer data type. Similarly, although most people don't think about it, there is some smallest increment possible between the discrete values of a 32-bit floating-point number. This is the tolerance, and the start value is some very large negative number. The unusual thing here is that the ConRes and ConRelRes allow you to specify arbitrary start values and tolerances. For example, ConRes allows you to say that the difference between every binary integer represents a difference of, say, 0.13 in the actual value. The difference between these two types is that ConRes specifies a linear mapping, whereas ConRelRes specifies a logarithmic mapping. That is, for ConRes, the numbers mapped are equally spaced, whereas, for ConRelRes, the differences between the mapped numbers increase logarithmically. To illustrate this further, consider the definition of the value for a property with data type ConRelRes:

property = *<start_value>* * [1 + 2 * *<tolerance>*] ^ *<teds_value>*

For the example in Figure 1 for property %RefFreq, the parser would read 6 bits from the TEDS. These 6 bits are interpreted as an unsigned integer and are the *<teds_value>* in the equation. So, if the 6 bits are 000010, then we have:

Reference Frequency = 7.9 * [1 + 2 * 3.26] ^ 2 = 446.74816

The following table shows the calculated property values for some potential values read from the TEDS with *<start_value>* = 7.9 and *<tolerance>* = 3.26.

<teds_value> Binary	<teds_value> Decimal	Calculated Property Value
000000	0	7.9
000001	1	59.408
000010	2	446.74816
000011	3	3359.5461632
111110	62	1.6716e+55

The last entry with a *<teds_value>* of 62 represents the maximum possible value that can be stored in the TEDS using these ConRelRes parameters. That is, this floating-point structure allows 63 discrete values over the interval [7.9, 1.6716e+55]. There are only 63 values instead of 64 because the standard defines a floating-point value of all ones as NaN (Not a Number).

The point here is that these data types allow scaling of floating type points to the exact scale and accuracy required for any specific application while minimizing the number of bits needed. This example only requires 5 bits for a very large range in contrast to the 32 or 64 bits normally used for floating point numbers.

There is also an enumeration data type. As illustrated in figure 1, an Enumerate command allows a discrete enumeration of values. The %Direction property command uses an enumeration type. The number of bits defined in this command is 2, because it takes exactly 2 bits to specify 1 of 3 enumerated values.

One other useful ability is to be able to define physical units. Anyone who has ever looked at the issue of standardized units knows that this is an incredibly difficult problem. The International System of Units (SI)[2] are as close as anything to a standard set of units, but not everybody likes or uses these metric based units. The Physical_Unit command allows a mapping between SI units and a text based description. It is outside the scope of this paper to describe the approach to defining the SI units. The important thing here is that the SI part of the mapping (the twelve element array) allows computers to determine unit equivalences. While the text based side allows users to define units in symbols they understand. As indicated in the comment, the Physical_Unit command maps 'Hz' to the unit '1/second.'

Control Commands

There are several commands lumped together under the heading of "Control Commands" in the standard. However, this control is always in the context of how to interpret bits in the TEDS. There is only the smallest amount of branching or looping control. Some of the commands under this heading do not map any bits in the TEDS at all. The two most significant control commands are the SelectCase and StructArray commands.

The SelectCase is the only command that allows any type of branching in a template. The command looks much like case commands in programming languages in that there is an outer delimitation using SelectCase and EndSelect. There are also Case and EndCase statements. Within the syntax of the Selectcase statement, there is a *<number_of_bits>* field. Once again, this indicates the number of bits to be read from the TEDS. The integer value of the bits read is used to determine which of the various cases to implement. A main benefit of the ability to specify cases is that it reduces the number of templates. Without a case selection ability there would be large numbers of templates with large identical sections. That is, the SelectCase command allows templates to be usable over a fairly general class of transducers.

The StructArray command is the only command that allows any looping, while at the same time it allows a way of grouping a set of properties. Once again there is a field in the StructArray syntax that indicates the number of bits to be read from the TEDS. The value of these bits determines how many times the elements in the structure are to be read. It thus acts much like a dimension variable in an array declaration of a programming language.

Some Miscellaneous Aspects of TDL
One more feature of the TDL is the ability to describe extended functionality of transducers through the use of subproperties. Syntactically, these are identified by putting a subproperty name inside square brackets immediately following a property tag. Although there are probably other uses, the main use of subproperties is to describe switches on the transducer. These may be actual, physical switches (perhaps dip switches or other toggles) or programmable switches. This functionality starts to overlap with user interfaces and actual control of the transducer. That is, this allows a description of how to modify the settings of a transducer rather than just getting information about its static transducing ability. Examples of subproperties include SwitchType, which indicates how many settings the switch has, and SwitchValue, which allows the description of what value each setting of the switch has. A specific example might be a gain switch that has four settings with different gains.

A couple of functionalities in the TDL are described here to emphasize to users that they should only be used as a last resort. Most standards have 'back doors' that allow a user to do pretty much anything they want just in case there is something that was not adequately covered in the main body of the standard. One such is the %MDEF property. By appending a string to %MDEF, it is legal to create a manufacturer defined property. This allows a manufacturer to introduce a property if there is no reserved property that fits the bill. The danger here is that a generic TDL parser will not know what to do with the property. The naming convention is provided so that a parser can identify it as legal and pass it on to a higher level application. Other back doors are the free form properties. The %user property allows the user to put information in to the TEDS in any form desired. Once again, the danger is that this format will not be understood by anybody that hasn't been given a description of the format. These properties should only be used as a last resort and avoided if at all possible!

There is also a %XML property. There is recognition by the working group that the extended Markup Language (XML) is growing in use. Thus, this property indicates that a portion of the TEDS is in XML. This is considered a free form property and the same danger applies that no one will know how to parse it. However, another aspect of using this property is that XML takes a lot of space. This is in direct contrast to the objective of needing only a minimal memory on the transducer.

It is worth mentioning that the use of XML instead of TDL was considered. A primary reason for not\ using XML is a legacy issue. The initial version of TDL was implemented quite a few years ago when the working group was first established and XML was not yet mature. A secondary reason is that XML takes a lot of space.

Because of the use of tags and end tags for everything in XML, the same information contained in a TDL file would take up considerably more space in an XML file. The working group not only had concerns about the amount of memory needed on a transducer, but also the amount of memory needed for support software on a small embedded system.

This paper has emphasized that a template maps bits in a TEDS. It is not true, however, that a single template maps every bit in a TEDS. There are two reasons for this. First, it is possible to concatenate templates. One suggested scenario is that a manufacturer might use a standard IEEE template along with one of their own manufacturer templates. These might then be added to with a template defined by the end user. A benefit of this approach is that a generic parser may have access to the IEEE standard template, but not the other two. This would allow at least some information to be obtained. (Although ideally, of course, all required templates should be available to the application parser.) Second, both in order to bootstrap and to concatenate templates, there is a control loop outside the template structure. In particular, the manufacturing code is not mapped in templates since this information is needed in order to find the appropriate template. This control loop is described in Clause 6 of the standard. Finally, the working group recognizes that, although they did their best to identify all properties, users of the standard are going to want to add more. The immediate out is the %MDEF property as mentioned above. However, the standard establishes a mechanism for adding properties to the standard (without having to revise the standard through a formal reballoting). In essence, desired properties can be submitted and the working group (or designee) will periodically review the submissions.

The Formal Grammar
Annex C of the standard contains a formal grammar for the TDL. This is written in syntax compatible with flex and bison (which are variants of lex and yacc[3]). These software packages are compiler compilers. That is, they use formal grammars as input and produce compilers for the specified language. The inputs to these packages have their own grammars, which are variants of regular expressions and Backus-Naur Form (BNF). In its raw form, this formal grammar is simply a method of verifying syntax: Is a given instance of TDL syntactically correct. However, also included in the annex are semantic routines (written in C++) that provide a basis for an application program. Mostly, the semantic routines just spit out the components of each command. This provides the core set of routines a programmer could use to develop whatever level of application is necessary. However, the semantic routines do a couple of other things such as verifying that property tags are valid and calculating the minimum and maximum number of bits possible in a TEDS for the given template. At some point, it is intended for the syntax-checking program based on the code in the annex to be readily available to anyone who wants it. This is to aid any would be user to bootstrap their development.

Summary
As a quick summary, let us walk through the example in figure 1. After the initial bootstrap process, the Template command forces the reading of 8 bits from the TEDS to verify the Template ID. The TDL version is verified. The text value of 'Hz' is

associated with the SI unit of 1/second. Then 6 bits are read from the TEDS to calculate the value of the property %RefFreq. Finally, an enumerated type is defined and used to interpret the 2 bits read from the TEDS to determine the value of %Direction. The outer control loop then checks to see if there is another template to parse further bits in the TEDS. This illustrates that the primary function of templates and the TDL is to map bits in a TEDS. Although there is a formal syntax and quite a few support structures in the TDL, the key to understanding templates are the property commands. These property commands provide the core functionality of templates. They are what define the information associated with a transducer, and obtaining that information is the primary reason for the IEEE 1451 family of standards.

---

[1] IEEE Std 1451.4, "Draft Standard for a Smart Transducer Interface for Sensors and Actuators – Mixed-Mode Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats." IEEE
[2] Le Systeme International d'Unites (SI), The International System of Units (SI), 7'th Edition (Bur. Intl. Poids et Mesures, Sevre, France, 1998) with Supplement 2000
[3] Levine, John R., Mason, T., Brouwn, D*.,"lex & yacc", A Nutshell Handbook*, O'Reilly & Associates, Inc., Sebastopol, CA 95472, 1995