Data Definition and Code Generation in Tcl

William H. Duquette
Jet Propulsion Laboratory
William.H.Duquette@jpl.nasa.gov

ABSTRACT

This paper discusses the use of Tcl-formatted data definition files processed at compile time in place of runtime-loaded text tables. This pattern has been applied many times in the development of the new Uplink Tracking and Command subsystem for Jet Propulsion Laboratory's Deep Space Network. For example, data defining the nature and capabilities of hardware controlled by the Deep Space Network's Uplink subsystem is placed in a data definition file. At compile time, the file is read by a tool which produces HTML documentation as well as C code for inclusion in the subsystem library. The hardware definitions thus become available to every aplication task in the subsystem with little run-time overhead.

1. BACKGROUND

1.1 The Deep Space Network

The Deep Space Network (DSN) is NASA's primary ground system for spacecraft telecommunications. A world-wide network of antennas and related hardware and software, it consists primarily of an operations center at the Jet Propulsion Laboratory (JPL), three Deep Space Communications Complexes (DSCC's) in California, Spain, and Australia, and the voice and data networks which connect them. The DSN is primarily used for tracking NASA spacecraft, but also supports the European Space Agency (ESA) and others.

1.2 The DSN Uplink Subsystem

Each DSN complex operates a number of "deep space stations", each of which consists of a dish antenna and related "subsystems". Each subsystem is a collection of hardware and software which supports one element of a successful spacecraft track. The Uplink subsystem is responsible for the production of the uplink signal, which is then fed to the antenna by means of the microwave subsystem. Production of the Uplink signal includes signal generation, frequency tuning, and modulation of various forms of data onto the carrier signal. The Uplink hardware includes an exciter, transmitters, command and ranging modulation boxes, and a controlling workstation.

The Uplink subsystem's software is mostly coded in C; C99 extensions are not yet available.

1.3 The Exciter

The exciter generates the carrier signal and modulates subcarriers on top of it. The DSN uses several different types of exciter, each designed for different frequency ranges; however, all are designed to be controlled by the same software. Thus, the software must know the particulars of each exciter type.

An exciter's hardware consists of a number of rack-mounted boxes called "assemblies". For example, every exciter includes an Uplink Channel Local Oscillator, or UCLO; this assembly is responsible for producing a desired sine wave given a number of reference frequencies as input. Other assemblies are specific to a particular frequency range and appear only in the exciter types which produce that frequency range. From a monitor and control point of view, each assembly consists of some number of components, which may be attenuators, phase-lock loops, sensors, and switches.

In order to monitor and control a specific exciter, then, the Uplink software must know:

- Which assemblies are included in that exciter
- For each assembly precisely which components of the four different types it contains.
- For each component, the addressing information and other details that allow the software to take readings from it, make sense of those readings, and in some cases control it.

For illustrative purposes, this paper will focus on attenuators, but the other components are handled similarly.

An attenuator is a device that controls the strength of some signal, rather like a volume control. It is monitored and controlled via a serial interface. The signal strength is controlled by setting the attenuator's value. To monitor and control a specific attenuator, the subsystem software must know the following:

- Name: A brief name used by the software to refer to the component.
- Comment: A human-readable name for the component.
- Address: Identifies the specific 64-bit serial interface; typically there are one or two per assembly.
- Monitor Line: The attenuator's value is read from the serial interface starting at this bit.
- Control Line: The attenuator's value is set by writing it to the serial interface starting at this bit.
- Max Value: The maximum attenuator setting.
- Min Value: The minimum attenuator setting.
- Bit Count: Number of bits in the attenuator value, 4 to 7.
- Resolution: The resolution of the least-significant bit, 1 or 10 (decimal).
- Bit Encoding: How the attenuator's value is encoded.
- Bit Order: Least-significant bit first, or most-significant bit first.

The other component types have similar sets of attributes.

The remainder of this paper concerns how this kind of information is made available to the application code that monitors and controls the exciter hardware, and to the developers who write that code.

2. HANDLING DEFINITION DATA

2.1 Hand Coding

The data defining the exciters' assemblies and components must be made available to the application software. The conceptually simplest approach is to hardcode all of the information directly into the application's source code. This solution is inflexible, error prone, and ugly, especially in C, where hand-editing nested structure and array initializers takes great care.

Moreover, because the C code is ugly and hard to read it's necessary to have a detailed and readable specification of the data. When changes are necessary it takes extra work to change both the spec and the code, and it is always difficult to be sure that the spec and the hardcoded data are consistent. This violates the "DRY" (Don't Repeat Yourself) principle, which states that "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."[1]

2.2 Table-Driven Execution

The traditional approach is to put the relevant data into tables which are read at run-time. This approach has been widely used in the past in DSN subsystem development, and is often the correct thing to do. It presents the data in an easy-to-read, easy-to-edit format, and allows changes to be made without recompilation. In principle, certain bugs can be fixed by redelivering the table without redelivering the application itself.

This is a useful technique, and the Uplink subsystem software uses it as appropriate. But it falls down when the data and the application logic are so linked that table changes imply code changes. For example, to properly control an attenuator the application must know not only that the component exists, but which signal it controls. This is domain knowledge which will become part of the application logic. Thus, while details might be defined in a table, the component's name and type (at a minimum) must be hard-coded. In addition, the attenuator's essential attributes are all related to its hardware design; once the definition is correctly entered, it will not be changed unless the hardware is changed—and if the hardware is changed sufficiently that the definition is invalidated, it is extremely likely (in the Uplink subsystem, anyway) that the software will need to be modified as well.

Thus, with this technique the application software bears the burden of parsing the table at startup to provide a flexibility that's of limited value. Moreover, syntax errors in the table might not be caught until the software is delivered to operations and invoked on the target platform.

Surprisingly, the table-driven method also violates the DRY principle, because there are potentially two copies of the table: the table you thought you delivered, and the table that's actually being used. Thus, for the kind of data we're discussing here it might still be necessary to have a detailed specification that's separate from the table itself.

2.3 Data-Driven Compilation

The table-driven approach does have one definite advantage: the data is available in an easily editable form. This advantage can be retained by processing the table, or data-definition file, at compile time instead of at run-time. During the build process, the following steps take place:

- The file is processed by a specially-written tool, which
  - Parses and validates the file. Thus, syntax and logical errors can be caught at compile time.
  - Stores the data in memory in an easily queried form.
  - Generates C source code which defines and populates the required data structures. The code might also contain access functions, etc.
  - Generates any other products of value, e.g., HTML documentation. The code and documentation are guaranteed to be consistent, as required by the DRY principle.
- The generated source code is compiled and linked into a library.
- Application tasks which need to use the hardware definition link with the library.

In short, a well designed data-definition file is a configuration-managed input to the build process that serves (through the medium of the translation tool) not only as program source code but also as a readable specification of the data.

3. DEFINITION FILE FORMAT

The data-driven compilation pattern has been applied many times during development of the DSN Uplink subsystem. The tools used to process the data files and generate the output have invariably been written in Tcl, for the following reasons:

- It's a text manipulation job, so the obvious tools are Tcl or Perl.
- If the file format is properly defined, the Tcl interpreter itself can be used to parse the file. This eliminates most of the syntactic processing.
- File formats defined in this way are clear and easy to edit; they don't look like "program code".
- The developer prefers Tcl to Perl.

In our first application of the pattern we devised a generic file format which proved so satisfactory that we've retained it for all subsequent definition file formats. It hinges on the fact that every definition file is defining some number of entities, each of which has attributes. That is, it's a sequence of record structures. Each entity definition looks like this:

```
<entity-type> <entity-name> {
        <attribute-name> <attribute-value>
                .
                .
                .
}
```

Normal Tcl quoting is used for the attribute values. Thus, here's a fragment from an exciter definition file. It defines one assembly,

containing one component, an attenuator. It also contains a change log entry:

```
change 6/11/2003 {
    author "Will Duquette"
    description {
        Added the SAT1 component to UCLO.
    }
}

assembly UCLO {
    comment {
        Uplink Channel Local Oscillator
    }
}

attenuator SAT1 {
    comment {S-band Attenuator 1}
    address 7
    monitorLine 1
    controlLine 13
    minValue 0
    maxValue 127
    resolution 1
    bitCount 7
    bitOrder MSB_FIRST
    bitEncoding BINARY
}
```

Note that the attribute/value list is in each case a list of pairs.

The first entity is a change log entry. Change log entries have no obvious unique name, so the name parameter was a convenient place to put the date. We could as easily have done without the name parameter altogether, and include the date of the change as another attribute. Each change log entity has two further attributes, an author (the person who made the change) and a description (what the change consisted of).

The next is an assembly entity. The entity is called UCLO, and has one attribute defined, a human-readable comment. We will define our processing code so that the component entities which follow immediately after will be included within the assembly (until the end of the file or the next assembly entity).

The last entity is an attenuator; the entity defines values for the attributes listed in Section 1.3.

This is a small fragment of an exciter definition file. A real exciter definition file would have many assembly entities, along with "pll", "sensor", and "switch" entities (the other three kinds of component), and would also have exciter entities listing the assemblies that comprise each exciter.

This basic format has proven quite flexible; it handles all kinds of attribute values, ranging from numbers to enumerated constants to blocks of HTML text. Most attribute values are simple atomic data but that's not a requirement; an attribute value can have any additional syntax we care to define.

It's easy to write a parser for this format in Tcl:
just define a Tcl command for each entity type and use "foreach" or even "array set" to parse the attribute list.

4. A SIMPLE IMPLEMENTATION

This section describes a simple Tcl solution for parsing definition files. The solution is similar to our initial implementations, but has disadvantages which will be described in Section 5.

4.1 A Simple Data Store

Out initial implementation stored all of the parsed data in a single Tcl array using complex array indices. For the example shown in Section 3, the data would be layed out in the array as follows:

Array Index	Description
`data(types)`	List of entity type names
`data(entities)`	List of entity IDs
`data(assemblies)`	List of assembly entity IDs
`data(attenuators)`	List of attenuator entity IDs
`data(changes)`	List of change entity IDs
`data(attrs-$type)`	List of attribute names for entity type $type
`data(types-$id)`	List of entity types to which entity $id belongs.
`data(value-$id-$attr)`	Value of attribute $attr for entity $id

The first five elements are not strictly necessary, but their values are easily constructed as the input is parsed, and they make it easier to iterate over the data later on. Retaining a list of the attribute names for each entity type is also not necessary, but simplifies error checking.

This simple data store can be elaborated in a variety of ways; for example, the single array could store multiple databases, possibly with different entity types, simply by inserting "`-$db`" into all of the above indices.

Note that entity types will often have attributes which never appear in the parsed input. Each assembly entity, for example, will have an "attenuators" attribute which at the end of parsing contains a list of the IDs of the attenuator entities which belong to the assembly. Similarly, each attenuator entity will have an "assembly" attribute naming the assembly to which it belongs.

4.2 A Simple Parser

The following script is a simple parser for the example file shown in Section 3. The name of the file to parse is in `argv`.

```
set currentAssembly ""
set counter 0

proc assembly {id attrlist} {
    global currentAssembly
    global data

    set currentAssembly $id
    lappend data(entities) $id
    lappend data(assemblies) $id

    foreach {attr value} $attrlist {
        set data(value-$id-$attr) $value
    }
}
```

```
proc attenuator {id attrlist} {
    global currentAssembly
    global data

    lappend data(entities) $id
    lappend data(attenuators) $id

    foreach {attr value} $attrlist {
        set data(value-$id-$attr) $value
    }

    set data(value-$id-assembly) \
        $currentAssembly
}

proc change {date attrlist} {
    global data

    set id "chg[incr $counter]"

    lappend data(entities) $id
    lappend data(changes) $id

    foreach {attr value} $attrlist {
        set data(value-$id-$attr) $value
    }

    set data(value-$id-timestamp) \
        [clock scan $date]
}

source [lindex $argv 0]

# Next, query the data and generate
# the desired outputs
    .
    .
    .
```

A Tcl command is written to parse each entity type. Each command simply takes the input data and stuffs it into the data array. Finally, the definition file is simply sourced as a Tcl script.

5. A REUSEABLE IMPLEMENTATION

The code presented in Section 4 shows how simple the parsing and data storage problem is when the possibility of input errors is ignored. But definition files can be many thousands of lines long (our largest contains over 17,000 lines), and are edited by multiple programmers, so input errors are likely, and it's best to catch them early. Our initial implementations quickly grew to do the following things:

- Verify that each new entity ID is unique.
- Verify that each new entity's attribute list contains all of the required attributes.
- Verify that each new entity's attribute list contains no unknown attributes (e.g. "coment" or "maxValu").
- Verify that all attribute values are valid.
- Verify that any other entity preconditions are met (e.g., the first attenuator entity must be preceded by at least one assembly entity.

When these had been added, the simple commands shown in Section 4 had grown much less simple. When the second file format was defined, we simply copied the code for the first and made the necessary changes. When the third was defined, it was clear that a certain amount of infrastructure was needed. This section describes our current software, after several cycles of abstraction and refactoring done over a period of several years. The implementation is not discussed in detail; instead, we discuss the pieces, how they fit together, and the lessons learned.

At base, however, the final implementation is really just an elaboration of the code shown in Section 4: the data is still parsed by Tcl commands named after the entity types, and stored in a Tcl array using the same technique.

5.1 The Pattern

Each definition file format is defined by three things:

- A man page documenting the input format, e.g., excdef(5).
- A Tcl package used to parse excdef(5) files into memory and query the result, e.g., excdef(n). This is colloquially called the file format's API.
- A tool script which uses the Tcl package to parse excdef(5) files and generate desired outputs, e.g., upl_excdef(1).

In some cases the Tcl package contains code to generate the typical output files, but more usually such code is in the tool script.

5.2 The Object System

After implementing several of the API packages, two things became clear: 1) all the APIs contained much the same set of commands for loading, parsing, and querying data, and 2) it is sometimes convenient to have several distinct sets of input loaded at the same time. Switching to an object interface style was thus a natural step: one program could then create multiple instances of each API, and in use all of the APIs looked more or less alike.

At first the interfaces were coded in pure Tcl[2]; later, a simple SNIT-like[3] package called otype(n) was implemented. Classes defined using otype(n) have constructors, destructors, instance variables, and instance methods. There is no inheritance, no class methods or variables, no automatic delegation, and no megawidget support. Instance variables are available in all method bodies without declaration, as is the special "self" variable which contains the name of the instance command.

5.3 The Data Store

All the file definition APIs are based ultimately on an in-memory data store called a "recordspace". Given a recordspace(n) object *::rs*, the available operations include:

`:rs typedef` *`typeId attrNames`*

: Defines a new entity type given its type name and attribute names. Recordspace does not allow the specification of default values or value constraints.

`::rs new` *`typeIdList recordId attrList`*

Adds a new record (entity) to the recordspace. The entity belongs to the types listed in the *typeIdList*. That is, its valid attribute list is the union of the attribute lists for all of the types, and it can be queried as a member of each of the types.

The *attrList* is a list of attribute names and values; the attribute names are validated. Unless an attribute's value is specified in the attribute list, it will default to the empty string.

`::rs types ?`*`recordId`*`?`

If no *recordId* is specified, this command returns a list of all defined entity types. Otherwise it returns a list of the entity types to which the specified entity belongs.

`::rs list ?`*`typeId`*`?`

If no *typeId* is specified, returns a list of all defined entities. Otherwise, returns a list of all entities belonging to the specified type.

An entity will be returned by *::rs list* for each type to which it belongs. It is sometimes useful to define entity types which define no new attributes, simply for use as categories.

`::rs get` *`recordId`* `?`*`attr`*`?`

If an *attr* name is specified, retrieves the value of that attribute for the given entity. Otherwise, retrieves all attributes of the entity as a list suitable for use with Tcl's *array set* command.

`::rs set` *`recordId attrList`*

Sets one or more values for the specified entity using an attribute/value list.

`::rs exists` *`recordId`*

Queries whether the entity exists or not.

`::rs hastype` *`recordId typeId`*

Queries whether the entity record is a member of the specified type or not.

`::rs delete` *`recordId`*

Deletes the specified entity.

The recordspace(n) package was written during the first wave of infrastructure development; subsequent file format APIs were developed directly on top of it.

The implementation of this data store is left as an exercise for the reader; it's based on the same Tcl array indexing scheme shown in Section 4.1. The important lesson here is that this simple API has proven sufficient to create and query a database of records of various types.

5.4 The Parser

The recordspace(n) data store abstracts the data storage problem quite nicely; it does nothing for the data parsing and validation problem. Rather than adding these features to recordspace(n) (and possibly breaking code that depends on it), we defined a reuseable definition file parser, def(n), on top of it using the otype(n) object system. def(n) wraps recordspace(n); APIs for specific definition file formats are then implemented by wrapping def(n) and defining an entity schema. This section describes the features of the def(n) parser; Section 5.5 shows how it is used to implement a file format API.

The def(n) parser has these features, over and above those provided by the recordspace(n) data store:

- Safe parsing, via a slave interpreter
- Symbol definition, symbol substition, and conditional processing, similar to that of the C preprocessor
- Improved entity definition, with attribute constraints.
- Standardized error handling

5.4.1 Safe Parsing and Symbol Substitution

Every def(n) object owns a slave Tcl interpreter; definition files are parsed by sourcing them into the slave interpreter rather than into the master interpreter, thus protecting the application code. Commands are aliased into the slave as needed to provide the necessary parsing capabilities. A def(n) object *::p* has the following methods for managing the slave interpreter:

`::p alias` *`alias target`*

Aliases a *target* command into the slave interpreter under the name *alias*.

`::p eval` *`command`*

Evaluates the *command* (which might be an entire script) in the context of the slave.

`::p define` *`name value`*

Defines a symbol, like a C preprocessor symbol. Symbols are used to control conditional processing, and may also be substituted into parsed text. Symbols are implemented as global variables in the slave interpreter. *define* is aliased into the slave, and thus can be used in input to be parsed.

`::p undef` *`name`*

Undefines a symbol. *undef* is aliased into the slave.

`::p subst` *`text`*

Substitutes symbols into *text*, returning the result. Symbols appear in the *text* using standard Tcl "$" notation. Literal "$" characters must be backslashed; command substitution is not allowed.

`::p ifdef` *`name thenclause`* `?else` *`elseclause`*`?`
`::p ifndef` *`name thenclause`* `?else` *`elseclause`*`?`

Evaluates the *thenclause* or the *elseclause* based on whether symbol *name* is defined or not. These are aliased into the slave to support conditional processing.

`::p parsefile` *`filename`*

Parses the named file by sourcing it into the slave interpreter. The file may contain standard Tcl commands, along with any commands aliased in by the *alias* or *entitytype* methods. If an input error is detected, *parsefile* will throw an error stating the file name and the nature of the error.

5.4.2 Entity Definition and Management

A def(n) object provides a superset of the recordspace(n) operations. Given a def(n) object *::p*,

`::p entitytype` *`name command attrDefs`*

Defines an entity type. The new type has the given *name*; the specified *command* will be aliased into the parser's slave interpreter to parse entities of this type. The *attrDefs* parameter is a block that defines the entity's attributes and their properties.

The attribute definition consists of comments and attribute definition lines. Each attribute definition line has this syntax:

`attr` *`name`* `?`*`options`*`?`

The following options are available:

`-required 0|1`

If 1, the attribute is required, i.e., must not be the empty string. The default is 0.

`-symbolsubst 0|1`

If 1, def(n) will attempt to substitute symbols into the attribute's value at entity creation and whenever the attribute's value is changed. The default is 0.

`-strippedline 0|1`

If 1, the attribute's value is constrained to be a single line of text. All leading and trailing whitespace is trimmed, and any internal whitespace is normalized to single space characters. The default is 0.

`-regexp` *`expression`*

The attribute's value is constrained to match the regular expression.

`::p new` *`typeList id`* `?`*`prettyName`*`? ?`*`attrList`*`?`

This method extends the recordspace(n) method in two ways: it handles the attribute options listed above, and it allows the specification of a "pretty name", which will be used to identify the entity in error messages if an error is detected. For example, the pretty name of an assembly might be:

```
assembly UCLO {...}
```

Since an attenuator is part of a component, the pretty name for the attenuator might be:

```
assembly UCLO, attenuator SAT1 {...}
```

Since it's difficult to pinpoint the line number of an input error, a properly defined pretty name helps the developer find the error.

`::p types ?`*`id`*`?`
`::p list ?`*`type`*`?`
`::p get` *`id`* `?`*`attr`*`?`
`::p exists` *`id`*
`::p hastype` *`id type`*
`::p delete` *`id`*

These are equivalent to the recordspace(n) commands of the same name.

`::p set` *`id attr value`*

Sets one attribute value for the specified entity, taking into account the attribute options.

`::p setlist` *`id attrList`*

Sets one or more attribute values for the specified entity using an attribute/value list, taking into account the attribute options.

`::p gensym` *`code`*

Generates a unique entity ID of the form "*_code:nnn*", where *nnn* is an integer.

5.5 Wrapping The Parser

A new file definition format API is defined by wrapping the def(n) API; usually the otype(n) object system is used, though that isn't required. The following methods are typically delegated to the def(n) object: *types*, *list*, *get*, *exists*, *hastype*, *define*, *undef*, *delete*, and *parsefile*. In addition, it will do at least these two things:

- Define an entity creation method for each entity type.
- Define each entity using the def(n) *entitytype* method, which will alias the entity's creation method into the def(n) parser's slave interpreter.

5.5.1 Entity Creation Methods

The exciter hardware definition format API is called excdef(n). Here is the entity creation method for the assembly entity:

```
method assembly {name attrList} {
    $self.p new assembly $name \
        "assembly $name {...}" $attrList

    set currentAssembly $name

    return $name
}
```

In the otype(n) object system, instance methods are defined using the *method* keyword. Every instance method has immediate access, without declaration, to the *self* variable and to all defined instance variables. By convention, the underlying def(n) object is always created as *$self.p*. So this method is calling the def(n) object's *new* method to create an *assembly* entity called *$name* with the specified *$attrList*, and saving the *currentAssembly* name. If an error is detected, the entity will be identified in the error message as "*assembly $name {...}*".

Note that this method is simpler even than the naïve implementation shown in Section 4.2. That's because def(n) and recordspace(n) are doing all of the necessary error checking.

The attenuator entity's creation method is more complicated than the assembly creation method shown above, because it has many additional error conditions to check:

```
method attenuator {name attrList} {
    # FIRST, create the entity
    set pretty \
        "assembly $currentAssembly,
attenuator $name {...}"

    $self component $name $attrList \
        attenuator $pretty attenuators

    # NEXT, validate unique fields
    $self CheckRange $name $pretty \
        controlLine 0  64
    $self CheckRange $name $pretty \
        maxValue    1 127
    $self CheckRange $name $pretty \
        minValue    0 127

    if {[$self.p get $name minValue] >
        [$self.p get $name maxValue]} {
        error "$pretty: minValue > maxValue"
    }
}
```

The *CheckRange* method simply verifies that the named attribute's value is within the desired range, and throws a pretty error message otherwise.

Note that the attenuator isn't created using the def(n) object's *new* method. Much of the needed creation code can be shared by all four of the component types, and this is implemented by the *component* method, which in turn will call *new*.

The most important thing to note in the above code is that all of the explicit error checks are truly part of the definition of the attenuator entity, and can't easily be abstracted away.

```
method change {date attrList} {
    ::Cu::try {
        set clockTime [clock scan $date]
    } catch msg {
        error "change $date {...}: \
invalid date '$date'."
    }

    set id [$self.p gensym "CHG$clockTime"]

    lappend attrList dateseconds $clockTime

    $self.p new change $id \
        "change $date {...}" $attrList

    return $id
}
```

This *change* entity creation method is rather more complicated than the naïve implementation in Section 4, but it does more. It's validating the date string; it's also generating a unique ID such that sorting a list of change entity IDs will put them in chronological order.

5.5.2 Schema Definition

The following is a portion of the schema definition for the exciter hardware definition file; it is executed in the excdef(n) object's constructor. The most important thing to note is the "component" entity type definition. It doesn't include a command to alias into the slave interpreter, because it's an abstract entity type. However, it does define all of the attributes shared by the four component types. Each attenuator component will be created first as an attenuator entity, and second as a component entity.

```
$self.p entitytype assembly \
    [list $self assembly] {
    # Input attributes
    attr name -regexp {^[A-Za-z0-9_]+$}
    attr comment -required 1 \
        -symbolsubst 1 -strippedline 1

    # Computed attributes:
    # lists of included entities.
    attr components
    attr attenuators
        .
        .
        .
}

$self.p entitytype component "" {
    # Input Attributes
    attr name -regexp {^[A-Za-z0-9_]+$}
    attr comment -required 1 \
        -symbolsubst 1 -strippedline 1
    attr address -regexp {^[0-9]+$}
    attr monitorLine -regexp {^[0-9]+$}

    # Output Attributes
    attr assembly -required 1
}

$self.p entitytype attenuator \
    [list $self attenuator] {
    # Input Attributes
    attr controlLine -regexp {^[0-9]+$}
    attr maxValue -regexp {^[0-9]+$}
    attr minValue -regexp {^[0-9]+$}
    attr resolution -regexp {^(1|10)$}
    attr bitCount -regexp {^[4-7]$}
    attr bitOrder \
        -regexp {^(LSB_FIRST|MSB_FIRST)$}
    attr bitEncoding \
        -regexp {^(BINARY|KAT1|KAT2)$}
}

$self.p entitytype change \
    [list $self change] {
    # Input Attributes
    attr author -required 1 \
        -symbolsubst 1 -strippedline 1
    attr description -required 1 \
        -symbolsubst 1

    # Output Attributes
    attr dateseconds
}
```

6. PRODUCING OUTPUT

Once the data is loaded into memory, it's necessary to generate the desired output. Generation of text is a normal Tcl activity, and would ordinarily be too trivial to discuss in a paper like this. But we've put a few spins on it that may be of interest.

In the case of the exciter hardware definition, this consists of C data structures and query functions, and of HTML documentation. The nature of the generated C code is complex, and though of great interest to the exciter software developers has little appeal for anyone else. Consequently, I'll use a simplified problem: creating an HTML list of assemblies. The output should look something like this:

```
<html>
<head>
<title>Exciter Assemblies</title>
</head>

<body>
<h1>Exciter Assemblies</h1>

<ul>
    <li>UCLO: Uplink Channel
        Local Oscillator</li>
</ul>
</body>
</html>
```

6.1 Text Generation with Append

The usual implementation would involve creating the HTML output as a string using the *append* command, and then writing it to a file or to standard output. This code assumes that we've already parsed the hardware definition file into an excdef(n) object called "*::ed*".

```
proc assemblydoc {} {
    set title "Exciter Assemblies"
    set text "<html>\n"
    append text "<head>\n"
    append text "<title>$title</title>\n"
    append text "</head>\n\n"

    append text "<body>\n"
    append text "$<h1>$title</h1>\n\n"

    append text "<ul>\n"
    foreach id [ed list assembly] {
        append text "<li>$id:"
        append text [ed get assembly comment]
        append text "</li>\n"
    }
    append text "</ul>\n"
    append text "</body>\n"
    append text "</html>\n"

    return $text
}

puts [assemblydoc]
```

Now, this code has a number of shortcomings, the most notable of which is that it's hard to picture the expected output. This is hardly unusual; most text generation code shares it. But in Tcl we should be able to do quite a bit better.

There are any number of things one might do to beautify this code; for example, one might use an HTML generation package to create all of the HTML tags. Most such efforts, however, result in code that's not much easier to read than that above, and which obscures the expected output at least as badly—one has simply substituted Tcl markup for HTML markup. The code shown above at least has the virtue of simplicity.

6.2 Text Generation with Subst

One appealing change is to use the Tcl *subst* command to format the text. This allows us to create a kind of picture of the desired output:

```
proc assemblydoc {} {
    set title "Exciter Assemblies"

    subst {
        <html>
        <head>
        <title>$title</title>
        </head>

        <body>
        <h1>$title</h1>

        <ul>
        [set text ""
         foreach id [ed list assembly] {
             set comment \
                 [ed get assembly comment]
             append text "<li>$id:"
             append text "$comment</li>"
             append text "\n"
         }
         set text]
        </ul>
        </body>
        </html>
    }
}

puts [assemblydoc]
```

In this version, it's easy to see what the output should look like; yet there are problems. First, it's difficult to control the output of whitespace; every line will be indented eight spaces. This isn't a big deal for HTML, where excess whitespace is automatically removed, but it can be a serious problem for other kinds of output. One could simply move the template text over to the left margin, but this just obscures the Tcl code (as well as confusing the auto-indenter in the text editor of one's choice).

The second problem is the loop in the middle, which after all is the most significant part: as soon as we need to loop over a number of items, we're stuck using append again.

Fortunately, both of these problems can be solved with the use of a little sugar.

6.3 Text Generation with Templates

The following example shows how to format the same HTML document using a *template*:

```
template assemblydoc {} {
    set title "Exciter Assemblies"
} {
    |<--
    <html>
    <head>
    <title>$title</title>
    </head>

    <body>
    <h1>$title</h1>

    <ul>
    [tforeach id [ed list assembly] {
        <li>$id: [ed get $id comment]</li>
    }]
    </ul>
    </body>
    </html>
}

puts [assemblydoc]
```

A *template* is a command that returns a formatted string based on a template string, possibly given some arguments. In this case, all of the necessary information is in the *::ed* object, so no arguments are needed. In this version, the template string closely resembles the expected output from start to finish, and the control logic is both clear and concise.

The *template* command defines a proc that returns a string. The *template* has two bodies. The first is an optional initialization body; it contains standard Tcl code, and is used to define variables for substitution into the second body, which is the template string proper. A modified version of *subst* is used to substitute local variables, template arguments, and commands into the template string, and the result is returned.

The "|<--" at the beginning of the template string is a special notation; it indicates the left margin. All whitespace in the template to the left of that is automatically deleted before the substitution is done. (The line containing "|<--" is also removed, of course.)

The *tforeach* command is an extended *foreach* whose body is a template string. It iterates over a list, assigning values to the index variable (or variables) just as *foreach* does; in each iteration, it substitutes its index variable(s), any other local variables or arguments, and commands into the template string using the same modified *subst* command. The results are concatenated and returned. Like *template*, *tforeach* can optionally include an initialization body; the loop shown above could also be written as follows:

```
    <ul>
    [tforeach id [ed list assembly] {
        array set a [ed get $id]
    } {
        <li>$id: $a(comment)</li>
    }]
    </ul>
```

6.3.1 The *tsubst* Command

The *tsubst* command is the basis for *template* and *tforeach*. It's equivalent to *subst* except that it has a simplified syntax (it always does all three kinds of substitution), and it looks for and handles the "|<--" indent marker, if present. It's implemented as follows:

```
proc tsubst {tstring} {
    # If the string begins with the indent
    # mark, process it.
    if {[regexp {^(\s*)\|<--[^\n]*\n(.*)$} \
        $tstring dummy leader body]} {

        # Determine the indent from
        # the position of the indent mark.
        if {![regexp {\n([^\n]*)$} \
            $leader dummy indent]} {
            set indent $leader
        }

        # Remove the indent spaces from the
        # beginning of each indented
        # line, and update the template
        # string.
        regsub -all -line "^$indent" \
            $body "" tstring
    }

    # Process and return the
    # template string.
    return [uplevel 1 [list subst $tstring]]
}
```

6.3.2 The *template* Command

The *template* command implementation is simple in concept; it's complicated by only two things: the desire to make the initialization body optional, and the need to do the substitution in the proper context. It is implemented as follows:

```
proc template {
    name arglist initbody {template ""}
} {
    # FIRST, have we an initbody?
    if {"" == $template} {
        set template $initbody
        set initbody ""
    }

    # NEXT, define the body of the new
    # proc so that the initbody, if any,
    # is executed and then the
    # substitution is.
    set body "$initbody\n"
    append body "tsubst [list $template]\n"

    # NEXT, define
    uplevel 1 \
        [list proc $name $arglist $body]
}
```

6.3.3 The *tforeach* Command

The *tforeach* command implements a subset of the normal *foreach* functionality in that it iterates over a single list. It suffers from the same complications as the *template* command. It is implemented as follows:

```
proc tforeach {
    vars items initbody {template ""}
} {
    # FIRST, have we an initbody?
    if {"" == $template} {
        set template $initbody
        set initbody ""
    }

    # NEXT, define the index variables.
    foreach var $vars {
        upvar $var $var
    }

    set results ""

    foreach $vars $items {
        if {"" != $initbody} {
            uplevel $initbody
        }
        set result [uplevel \
            [list tsubst $template]]
        append results $result
    }

    return $results
}
```

6.3.4 The *tif* Command

The *tif* command is a templated *if*, just as *tforeach* is a templated *foreach*—it has a *thenbody* and an optional *elsebody*, each of which is a template string. Either the one or the other is substituted and returned based on the value of a *condition*. For example, the following code supplies a default title:

```
template assemblydoc {title} {
    |<--
    <html>
    <head>
    <title>
    [tif {"" != $title} {
        $title
    } else {
        Exciter Assemblies
    }]
    </title>
    </head>
    .
    .
    .
}

puts [assemblydoc]
```

The *tif* command is less often needed than *tforeach*; in the example shown above, it would be more natural to set the value of *title* in an initialization body. It is implemented as follows; note that *elseif* clauses are not supported.

```
proc tif {
    condition thenbody
    {"else" ""} {elsebody ""}
} {
    # FIRST, evaluate the condition
    set flag [uplevel 1 \
        [list expr $condition]]

    # NEXT, evaluate one or the other and
    # return the result.
    if {$flag} {
        uplevel 1 [list tsubst $thenbody]
    } else {
        uplevel 1 [list tsubst $elsebody]
    }
}
```

7. CONCLUSIONS

During the course of development of the DSN Uplink subsystem software to date, we've found compile-time data definition files to be extremely useful; at present we are using four different definition file formats; all but one use the new infrastructure.

We've found that while conceptually simple Tcl implementations exist, more sophisticated infrastructure provides better error handling, easier coding, improved maintainability, and reduced code size. The excdef(5) format and tools described in this paper were written using the new infrastructure to begin with, but in another case a definition file API and tool originally written using a "naïve" implementation were rewritten to use our latest infrastructure. After the rewrite the API code shrank by 25% from 923 lines to 703 lines of code, and the tool script shrank by 20% from 1975 lines to 1589 lines of code.—at the same time as additional features were added.

In the case of the tool script, much of the improvement resulted from the deletion of proc header comments. Using the old-style "append"-based text generation, the code to produce a single document was typically broken for clarity into many routines. There was usually at least one for each section, with many helper procs as well. Using templates, it's feasible to put an entire document in one template with perhaps one or two helper routines—and no loss of clarity. Thus, the new version contains fewer procs, and thus fewer proc header comments. As our coding standard calls for fairly large header comments, the effect on code size—and on code readability—is substantial.

8. REFERENCES

[1] Hunt, Andrew, and David Thomas, "The Pragmatic Programmer", pg. 27, Addison-Wesley, 2000.

[2] Duquette, William H., "Guide to Creating Object Commands", http://www.wjduquette.com/tcl/objects.html.

[3] Duquette, William H., "Snit's Not Incr Tcl", http://www.wjduquette.com/snit.

9. ACKNOWLEDGEMENTS

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.