Textual Specifications, Tabular Languages, and PCCTS

Dr. Loring Craymer
Jet Propulsion Laboratory

Introduction:

Many software engineering problems seem to have "natural" languages associated with them. Tabular specifications are common and easily understood. Often even fairly complex systems can be described through a hierarchy of tables which mix numerical and computational descriptions with natural language text.

Tabular specifications are quite common in documentation for intelligent hardware--hardware that has its own tokenized command language. These tokenized command languages are often both a blessing and a curse. The virtue of a tokenized command language is that it makes a device easy to program and effectively modularizes a portion of the overall system. The disadvantage is that these languages is that they tend not to be tiny--100 commands/responses is not atypical--although they are usually small, and that they tend to grow (and change, although growth is more common for commercial products--backward compatibility avoids customer anguish) with new revisions of the hardware and firmware. It is often tedious to program drivers for such languages--repetitive coding is common (many commands have similar, but not identical, formats)--and maintenance can become a nightmare as the command language changes.

I have "discovered" a very useful approach for dealing with tabular specifications which minimizes development time, reduces maintenance requirements, and which can lead to improved system design because of an improved understanding of interface requirements. Tables necessarily have a logical format. That format can be viewed as providing an unambiguous syntax for a descriptive language, and the individual tables can be viewed as "programs" written in this language. These "programs" can be "compiled", with application-specific processing, into a form which fits within the context of the overall application.

The concept of application-specific languages is hardly new--text filters are a familiar example. What is new is that PCCTS makes it relatively easy to process application-specific grammars. Even for rather small problems--20-40 line specification tables--it may be more efficient to build a translator which generates code from the tabular specification than to hand-code the relevant part of an application: it is easier to debug and maintain the automatic code-generator than it is the hand-coded solution.

Text processing with PCCTS--the Cassini Spacecraft Command Language

The tradition at JPL is to command spacecraft using a "virtual assembly language" and “virtual machine code” (binary format: instruction code followed by instruction parameters) which differs for each spacecraft. For the Cassini mission, the

virtual assembly language currently consists of about 1200 commands. These commands are described in a 1300-page specification document. One of the issues associated with such a large command language is whether the implemented command language database conforms to this specification or not. The planned approach was to develop an editor which could be distributed to all of the subsystem command developers and which would generate both the specification and the command database entries. Unfortunately, the editor was not developed quickly enough and the specification document was composed with a melange of tools and integrated into a large FrameMaker document. The commands were then manually entered in the command database.

This approach left the consistency of the command database and the specification document open to question. It was decreed that the support utilities (developed for debugging purposes) not use the command database, but instead use a database to be independently constructed from the specification.

The specification document is fairly clearly laid out as a series of regular text and table entries. Appendix A illustrates the two main table types. The problem was to extract the critical information from these tables and transform it into a manipulable form.

The “obvious” approach would to construct a perl script which would do as much of the necessary processing as possible. The result would require additional editing of the processed output, but could probably be made to work. However, the specification document was temptingly regular and looked like an ideal application for PCCTS--cross-checks could be performed on the parsed data, and a PCCTS parser would be easier to maintain than an ungainly and predictably messy perl script. A PCCTS parser could also be used as the basis of other tools--once a PCCTS grammar is “tagged” with references to actions, the actions can be redefined as needed. The advantages of using PCCTS were clear, so that was the approach chosen.

As a first step, it was necessary to obtain the document in a file format which preserved layout information, preferably an ASCII format. Postscript was a possibility, but did not seem a particularly good choice since Postscript is a page layout language, not a language designed to convey document structure. Microsoft’s Rich Text Format seemed a reasonable choice, and had the advantage that the RTF syntax specification can be obtained by anonymous ftp from microsoft.com.

RTF proved to be a tractable approach. The language structure is fairly simple. All command keywords are of the form “\\ [a-z]+ [0-9]* \ ”, syntax is prefix with braces used to set off groupings of commands, and there is but a single flow-of-control. Despite the formidable appearance of RTF documents, they are actually easy to extract information from. Command structures which define features of the overall document (such structures as header and footer definitions, font declarations, and the like), but which contain no relevant information, take the form “{<command> <other commands and text>}” can be filtered out by the lexer--once one of these commands is recognized, the lexer can discard all tokens through the enclosing right brace. Most other commands

can be skipped; indeed, only the seven commands listed in Table 1 are needed for data extraction.

RTF command	Token Type
\sect	EndSect
\pagebb	EndPage
\par	EndPar
\line	NewLine
\cell	EndCell
\row	EndRow
\intbl	InTable
\trowd	TblStart

Table 1. List of useful RTF commands.

Once the list of recognized commands is pared down to the above set, a generic RTF document grammar is also quite simple:.

```
/* generic RTF grammar elements */
page:
       (para)+ ;

para:
       (      (textpar endLine)
              | table
       )  {EndPar | NewLine} ;

textpar:
       atext (Delimiter ((atext)? |))* ;

table:
       TblStart (row)+ ;

row:          (cell)+ InTable EndRow ;

cell:
       textParCell
       | nullCell ;

textParCell:
       InTable textpar EndCell ;

nullCell:
       InTable EndCell ;

textItem0:
       Text | Number | Punct | SpecialChar ;


alphanum:
       (      Number
              | Text
              | Dot
       )+ ;

textItem:
       Text | WhiteSpace | Number | Punct | SpecialChar ;
```

Once this generic grammar had been identified, it was possible to construct a grammar for the overall specification document. Command listings follow the rule

```
cmdDescBody:
        stem name purpose routing
        inputFields
        inputParams
        translation ;
```

and the inputParams and translation tables have rules

```
// Input parameter descriptions table
ipTable:
        ipTblHeader {TblStart} (ipEntry)* (footnote)* ;

ipTblHeader:
       TblStart textCell
       textCell
       textCell
       textCell
       textCell InTable EndRow ;

// individual input parameter descriptions--ADD CODE HERE
ipEntry:
       (ipEllipsis)? |
         (    cmdField dataType legalValues units accuracy InTable EndRow
              {EndPar TblStart}
       ) ;

ipEllipsis:
       dotCell
       (emptyCell | dotCell)
       (emptyCell | dotCell)
       (emptyCell | dotCell)
       (emptyCell | dotCell) InTable EndRow ;

dotCell:
       InTable {WhiteSpace} (Dot)+ EndCell ;

// "Translation" table
translation:
        <<!strcmp(LATEXT(1), QUOTE(Translation))>>? Text
        Colon endLine
       translationTable {endLine | EndPage} ;

translationTable:
        transTblHdr transTblBody ;


transTblBody:
       (transTblEntry)? transTblEntry transTblBody
       | ;

transTblHdr:
        TblStart textCell
       textCell
       textCell
       textCell
       textCell
       textCell InTable EndRow ;

// translation table entry
```

```
transTblEntry:
       (EndPar {EndPar} TblStart transTblEntryBody)?
       | ({TblStart} transTblEntryBody) ;

transTblEntryBody:
       ((emptyCell)? transContinuation
         | (wordNo bitRange cmdField dataType binData hexData)
       )
       InTable EndRow ;

// continuation of translation table entry
transContinuation:
       emptyCell bitRange cmdField
       (textCell | nullCell) binData hexData ;
```

This should suffice to give the feel of the overall grammar. The grammar proved somewhat more complex than originally anticipated, somewhere between the ANS C grammar and the C++ grammar in size. Many visually similar layouts proved to be implemented differently, and each revision of the specification document introduces new complexities. The grammar has to be extended for each release, and it has become quite apparent that any other approach to this parsing problem would have degenerated into manual extraction of the critical data. Interestingly enough, the development time was dominated by the grammar--it took about three weeks to recognize the essential elements of the RTF specification, about six weeks to develop the initial grammar and about two weeks for each new release of the Cassini specification document, but only three days to "tag" the grammar with actions and write and debug those actions.

It should be noted that all of the key features of ANTLR were required--the grammar is mostly LL(3), but both syntactic and semantic predicates had their place. It was also necessary to construct a custom input function for the lexer, and it proved convenient that DLG allows the return of a different token type (via assignment to NLA) than was originally identified--it proved easier to identify generic RTF commands, do a symbol table lookup to classify them, and then take appropriate action, than it would have been to identify the individual commands. It is also worth noting that the parser has very respectable performance--it takes about 2 minutes on an SGI Indigo (R3000-based) to parse the entire 13 Mb document, and this includes about ten separate invocations of the parser since the document is delivered in chunks.

Driving intelligent hardware--identifying application-specific languages

The "virtual machine code" approach used for the Cassini spacecraft is common in command languages for intelligent hardware. It is natural to document these codings in tabular form, and such tables are common in documentation for intelligent commercial hardware. The usual approach to writing drivers for such hardware is to repetitiously code driver functions for each command and then go through the tedious process of debugging each function written. This is a labor-intensive approach, and the resulting code is neither elegant nor particularly maintainable.

I place a high value on maintainable code, and prefer to avoid tediously repetitive tasks at all cost. When faced with the task of driving a TANS Vector GPS receiver from

Trimble Navigation, Ltd. for use in a prototypical spacecraft attitude determination system, I looked for an alternative implementation approach.

My experiences with parsing the Cassini specification document suggested that something similar could be done to drive the receiver. The command language specification could be extracted from the documentation for the receiver (manually, unfortunately: the document was not received in machine-readable form), and processed with a custom translator written with PCCTS. The language describing the tables would be small, and development time should be correspondingly rapid. Debugging time could also be expected to be reduced, since repetitive coding would be replaced by repetitive calls to code generating functions.

Extracting the data from the TANS Vector documentation into tabular form was straightforward, and required minimal thought. Table 2 shows a sample of the extracted data.

**USER PACKET DESCRIPTIONS

<id>	type	value	field name (optional)
1D	BYTE	0x43	
1E	BYTE	0x4B	
1F	none	none	
20	BYTE	PRN	
21	none	none	
22	BYTE	0 | 1 | 3 | 4	
23	SINGLE	any	X (meters)
	SINGLE	any	Y (meters)
	SINGLE	any	Z (meters)
24	none	none	
25	none	none	
26	none	none	
27	none	none	
28	none	none	
29	none	none	
2A	SINGLE	any	
2B	SINGLE	any	lat (radians, north)
	SINGLE	any	long (radians, east)
	SINGLE	any	alt (meters)
2C	BYTE	0-4 (3)	dynamics
	SINGLE	any (0.1745)	elevMask (radians)
	SINGLE	any (6.)	signalMask
	SINGLE	any (12.)	PDOPmask
	SINGLE	any (8.)	PDOPswitch

Table 2. Portion of table describing command language for Trimble's TANS Vector GPS receiver.

Table 2 is reasonably self-explanatory. Each row of the table describes a parameter; a command id is listed for the first parameter of each command. Each parameter has a data type, a legal range of values (default values in parentheses), and may have a field name (comments in parentheses). The corresponding grammar is fairly simple:

```
/* grammar for processing TRIMBLE packets */

#token LParen "\("
#token RParen "\)"
#token OR "\|"
#token Comma ","
#token LBracket "\["
#token RBracket "\]"
#token NewLine "\n" <<zzline++;>>
#token Continuation "\n [\t]+"   <<zzline++;>>
#token Dashes "\-\-"
#token Colon ":"
#token NONE "none"
#token ANY "any"
#token BYTE "BYTE"
#token SINGLE "SINGLE"
#token INT "INTEGER"
#token FLOAT "FLOAT)"
#token DOUBLE "DOUBLE"
#token STRING "STRING"
#token USER_TITLE "\*\*USER \ PACKET \ DESCRIPTIONS"
#token TANS_TITLE "\*\*TANS \ PACKET \ DESCRIPTIONS"
#token TextItem "[a-zA-Z_ 0-9 \- / \. \?]+"
#token ShiftLeft "\<\<"
#token Char "\' {\\} [\ -\~] \'"
#token Tab "[\t]+"
#token "\ "  <<zzskip();>>


trimbleTables:
       USER_TITLE NewLine table
       TANS_TITLE  NewLine table
       ;

table:
       tableHeader (NewLine)+
       (tableEntry (NewLine)+)+
       ;

tableTitle:
       (TextItem)+ ;

tableHeader:
       "\< id \>" Tab TextItem Tab TextItem Tab
       TextItem TextItem LParen TextItem RParen ;

tableEntry:
       TextItem      Tab
       (dataEntry | blankEntryOrCont)
       {Tab fieldName (continuation Tab fieldName)*}
       ;

blankEntryOrCont:
       NONE OR Tab NONE OR
       Continuation dataEntry ;

dataEntry:
```

```
       dataType
       Tab argList ;

continuation:
       Continuation {startStruct} dataType
       Tab argList ;

dataType:
       NONE
       | ((   BYTE
              | STRING
              | INT
              | SINGLE
              | FLOAT
              | DOUBLE
       ) {array})
       ;

startStruct:
       TextItem LBracket RBracket
       Colon Continuation ;

array:
       LBracket {TextItem} RBracket
       ;

argList:
       NONE
       | ANY {comment}
       | shiftExpr
       | itemList
       | charList
       ;

shiftExpr:
       TextItem
       ShiftLeft TextItem
       ;

itemList:
       TextItem
       (OR TextItem)*
       {comment} ;

charList:
       Char (OR Char)* ;

comment:
       LParen TextItem {Comma TextItem} RParen ;

fieldName:
       TextItem
       {comment} ;
```

An unexpected design benefit appeared as a result of developing the parser. Although the original intent had been to build a small compiler which would generate packing/unpacking routines (command packets are BigEndian and data types are packed on byte boundaries) for each command type, it proved simpler to encode the data layout for each packet as a character string and to encode/decode based on the character strings. The mini-compiler was used to produce the table of encodings, along with a table of data field names which could be used for constructing the user-level command interface. This approach to encoding data has very little impact on performance, is highly space

efficient, and would not have been practical had the table been generated by hand. For a hand-generated table, there would have been the problem of matching the right character string with the right index (an error prone activity). Maintenance would also have been error-prone: manual translation of structure specification into character string encoding is tedious, and an upgrade to the command language would force a repetition of the table-building process. The sole advantage would be that most of the encodings would not change, so few bugs would be introduced with an upgrade.

Implementation time was about as predicted. This particular grammar and its associated actions were developed and debugged in a few days. One other parser was developed to match user commands with the receiver's responses (the I/O driver required this level of intelligence); the lexer and parts of the grammar could be taken directly from the grammar described above. The total implementation time for generating the core functionality for the driver was under two weeks. A more conventional approach would easily have taken 1-2 months and would have required writing more than twice as much code with less assurance of the robustness and maintainability of the end product--it is unlikely that all of the driver functions would have been adequately tested. It is also likely that less functionality would have been implemented.

One of the nicer features of this implementation approach was that the overall application is implemented as a collection of independently testable units. Additionally, the parts of the application which are most subject to change are encapsulated in the parsed tables, and the tables are both self-documenting and easily maintained. Of equal importance is the maintainability of the PCCTS grammar: if desired, the specification tables could be extended and the grammar updated to match.

Conclusions

The value of domain-specific tools (awk, perl, sed, and the like) is well known. Application-specific tools are also useful, but are subject to implementation tradeoffs: it is an all too familiar occurrence for a software developer to spend more time automating a task than it would have taken to do it manually, even if several repetitions of the manual effort were required. PCCTS makes a considerable difference in the cost of implementing application-specific tools. The parser for the Cassini command language specification would have been difficult to construct in any reasonable time period without PCCTS. Similarly, the table translaters for the TANS Vector command language would not have appeared a reasonable approach were it not for the maintainability and ease of development of the PCCTS grammar.

Literature Cited

Cassini Orbiter Functional Requirements Book: Uplink Formats & Command Tables (CAS-3-291). Jet Propulsion Laboratory, June 30, 1994.

Rich Text Format (RTF) Specification, version 1.3. Microsoft Product Support Services Application Note GC0165, 1/94.

TANS Vector GPS Attitude Determination System Specification and User's Manual. Trimble Navigation, Ltd., 3/95.

Appendix A--a brief extract from CAS-3-291 (preliminary)

6.8 Solid-State Recorder Commands

6.8.1 SSR Command Summary

Cmd Stem	Cmd ID	Descriptive Summary
16CE_BKUP_PWROFF	n/a	Enable Backup Power to SSR-A or SSR-B to be turned off
16CE_PRM_PWR_OFF	n/a	Enable Prime Power to SSR-A or SSR-B to be turned off
16PS_SSR	n/a	Turns ON or OFF Solid State Recorder A/B, or resets the switch from tripped State

6.8.2 SSR Command Listings

Stem: 16CE_BKUP_PWROFF

Name: SSR-A/B Bkup Pwr Off Ena

Purpose: Enable Backup Power to SSR-A or SSR-B to be turned off

Routing: Channel 0; CDS-A and CDS-B

Command Input Fields: 16CE_BKUP_PWROFF, *Off Cmd Status*

Input Parameter Descriptions:

Cmd Field	Data Type	Legal Values	Units	Accuracy
Off Cmd Status	Enumerated	ENABLE, DISABL	n/a	n/a

Translation:

Word #	Bit #	Cmd Field	Data Type	Binary Data (MSB to LSB)	Hex
01	15	Off Cmd Status: ENABLE	Enumerated	1	1
		Off Cmd Status: DISABL		0	0
01	14-13	Unused		00	0
01	12-8	CRC Address	Unsign Integer	0 0110	6
01	7-0	Discrete Bit Select	Unsign Integer	0100 0000	40

Expanded Parameter Description:

Off Cmd Status - When this CRC bit is enabled (set to "1") in either CDS string the PPS will accept an SSPS command to turn off backup power to either SSR A or SSR B..

CRC Address - The CRC address field selects which group of 8 CRC bits or CRC mask bits is to be modified by the command. 16CE_BKUP_PWROFF is in the group of 8 CRC bits with a CRC address field of 06 hexadecimal.

Discrete Bit Select - The Off Cmd Status CRC bit selects the bit from the group 8 CRC bits in which it resides. See also TBD.

Discussion:

TBD

This CRC is also referred to as CRC #54.

Constraints:

This is a non-volatile CRC. It will retain state when power is cycled. After power-on the state of this bit is its previous state.