TIDI Instrument Command Language

Compiler Specification and User's Guide

APPROVAL RECORD			
Function	Name	Signature	Date
Originator	D. Gell		
Flight Software	S. Musko		
Instrument Scientist	W. Skinner		
Program Manager	C. Edmonson		
Systems Engineer			
R&QA	John Eder		

REVISION RECORD

Rev	Description	Date	Approval
	Initial Release	11 Mar 1998	
	· Revised format of the command block file to indicate command boundaries. · Replaced the *.ident* directive with the *program* command.	16 Mar 1998	
	· Added binning table ID to the load_bin_table command. · Deleted command to clear the stored control program holding buffer and append to CPHB from start of stored control program. The package process performs that step.	5 May 1998	
A	· Compiler to delete partial command block files on termination due to signal · Changed compiler speed requirement to a rate · Listing file shows results of symbol substitution and file inclusion. · Diagnostics and compilation statistics are written to stderr, not stdout · No labeled statements · Specify identifier syntax and specify for local variable name and subroutine name. · Replace program compare and jump with control structures · Wait parameter is in centiseconds · Instrument control commands require constants for each parameter · Load_scan_table and Load_Bin_Table require a file parameter, inline compilation of tables performed with .scan_table_start and .bin_table_start directives. · Added control structures	18 June 1998	
A1	· Changed ENDIF to END_IF · Clarified BREAK and CONTINUE · Corrected argument type for telescope and shutter commands, should only be constants. · Added note about conversion from engineering units to scaled units · Revised command block file format	22 June 1998	
B	· Length of a command block line changed to 16 bytes (ex 30) · Delete Appendix E, replaced by document 3634A, renumber appendices · Add error messages to appendix E · Complete Appendices F, G	6 July 1998	

REVISION RECORD

Rev	Description	Date	Approval
C	Reflects TICL compiler as delivered 17 July 1998 · *Load_CPH* replaces *Append_CPH* · Corrected specification of numerical constants · Added engineering unit constants · Changed filter wheel position to encoder position from filter position in *FilterWheel* command. · Added *Boot* and *Allow_WD_Expire* commands · Appendices *Command Implementation, Scan Table Compilation* and *Binning Table Compilation* deleted	16 Jul 1998	
D	Adds memory writing commands required for instrument test · write_byte writes a byte to the specified address · write_word writes a word to the specified address · write_double writes two words to the specified address	25 Jan 1999	
E	Reflects the compiler as released 29 October 1999 (version 1.2) · Adds command to copy control program holding buffer to EEPROM · Adds Appendix E, TICL Design Standard · Style and typo corrections	1 Nov 1999	

Contents

Commands

Tables

0. Introduction

0.1 Purpose

This document defines the specifications for the TIDI Instrument Command Language (TICL) Compiler. The compiler is used to generate instrument commands to be executed by the TIDI Flight Software. These commands can either be organized as a control program and executed from memory within the flight software system, or as immediate commands, which are executed upon receipt by the flight software system.

0.2 Related Documents

1) Musko, S. "TIDI Flight Software Requirements Specification", SPRL File 055-3320, 15 Jan 1997
2) Musko, S. "TIDI Instrument Parameter Definitions", SPRL File 055-3519
3) Gell, D. "TIDI Scan Table File Format", SPRL File 055-3527A, 5 May 1998
4) Gell, D. "Coordinate Frames and Viewing Directions", SPRL File 055-3543, 29 Jan 1998
5) Gell, D. "Binning Table File Format", SPRL File 055-3603, 27 May 1998
6) Rowe, S. " Instrument Language Compiler Design and Maintenance Document", SPRL File 055-3633, 4 July 1998
7) Rowe, S. "Instrument Command Language Compiler, Installation and Usage Guide", SPRL File 055-3634, 4 July 1998

1. Requirements

1.1 Environment

1.1.1 Host Computer

The TIDI Instrument Command Language (TICL) Compiler shall execute on the TIDI data processing system. This system consists of Hewlett-Packard C class workstations and the HP-UX version 10.20 operating system.

The TICL Compiler shall be implemented without using any extensions to ANSI standard C so that it may be ported to other hosts if needed.

The TICL Compiler shall respond to the UNIX SIGTERM signal by performing an orderly program termination. This signal is the standard signal by which the operating system notifies a process that it is to terminate. When the signal is received, the program shall log the termination to the appropriate output file, close all open files deleting invalid command block files, and perform an orderly shutdown.

1.1.2 Target Computer

The TICL Compiler shall generate instrument commands, defined in reference 1, which will be interpreted in the TIDI Flight computer, based on a UT69RH051 microcontroller, an Intel 8051 compatible processor.

The compiler will produce both command language control programs and streams of immediate commands. Command language programs shall not contain memory load, RAM code execution, boot,

autoboot disable, or watchdog timer expiration statements. Immediate command streams shall not contain the program command, subroutine declarations, subroutine calls or control structures.

1.2 Performance

1.2.1 Speed

The TICL compiler shall complete all passes through a TICL program at a rate of at least 100 source code lines per second

1.2.2 Size

There is no requirement upon the size of the Compiler executable image.

1.2.3 Level of Optimization

The compiler shall perform optimizations of the generated code. The optimizations shall remove redundant operations and minimize the size of the generated code.

1.3 Operation

1.3.1 Required Input Files

The TICL compiler shall accept a stream of input statements from a file or from the standard I/O. The statements shall be delimited by a new-line character. Command arguments shall be used to specify the input source.

1.3.2 Output Files Produced

The TICL compiler shall produce a listing file and a command block file. The listing file shall contain the list of source code statements showing the results of symbol substitution and file inclusion, with line numbers, offsets, compilation error messages, and compilation statistics. The command block file contains the sequence of instrument commands, in hex, prefixed by descriptive information required to package the commands into a TIMED command message. The format for the command block file is specified in Appendix A.

1.3.3 File Naming Convention

File names are made of a file description string and a file type string separated by the period "." character:

```
description.type
```

For input files, the description field is specified by the TICL author and should describe the function performed by the TICL program. The default type for source files is "`ticl`".

Output file names are formed by replacing the input file name type with the output file name type. For listing files the type is "`list`" and for command block files type is "`tcmd`".

1.3.4 Run Time Diagnostics and Statistics

The TICL compiler shall produce runtime diagnostics indicating any anomalous conditions encountered during operation.

The TICL compiler will produce a compilation statistics summary indicating the performance of the program. This summary will include the name and location of the input file, the names and locations of the output files, the execution time, the number of compilation errors encountered and any other statistics that will be useful in assessing the performance of the program.

The diagnostics and statistics shall be written to the standard error stream and appended to the listing file.

2. Command Language Syntax

Input the compiler consists of two types of statements: TIDI Instrument Command Language *statements*, and *compiler directives*.

The TIDI Instrument Command Language is used to control the operation of the TIDI instrument. It consists of statements which are grouped into collections called programs. Programs are limited in length such that the resulting pseudo-code will not exceed the length of the control program holding buffer.

A control program consists of a main program, and optional subroutines. All subroutines must be defined prior the main program in the source file. Subroutines may call other subroutines with a maximum nesting depth of 64. Subroutines may be called recursively.

Compiler directives are used to modify the action of the compiler. They do not result in the generation of any pseudo code.

2.1 Statement Formation

Each TICL statement consists of two fields, a command field and a comment field, as shown in the example below:

```
command                ; comment
```

Each field is optional. No statement may exceed 1 line of 80 characters in length.

Within a statement, each field is separated by whitespace, which may consist of one or more spaces, tabs, or combination.

The comment field begins with the semicolon (";") indicator character. The command field consists of a command keyword followed by zero, one, two, or three operands. Operands may be instrument parameter mnemonics, defined in reference 2, numeric constants, identifiers , or string constants.

2.1.1 Comments

The comment field is separated from the remainder of the statement by a semi-colon (";"). Any text following the semicolon in the statement is ignored. The comment field may be the only field within at TICL statement.

Blank lines may be placed anywhere in a TICL source file to improve readability.

2.1.2 Command Keywords

Command keywords specify the action that the instrument is to take. A keyword must be one of those defined in section 3.1, 3.2, 3.3, or 3.5. Abbreviations are not allowed, case is not significant

2.1.3 Identifiers

Identifiers are used as names of instrument parameters, global variables, local variables and subroutines. Identifiers may be up to 32 characters in length and must start with either a letter or an underscore. The remaining characters may be any letter, number or the underscore. Case is not significant. Example valid identifiers are

```
sub1
_trial                          ; leading underscore ok
A_miX123ed_label2               ; a mixed alpha and numeric
_12345                          ; leading underscore
```

Example invalid identifiers are:

```
1sub     ;initial character cannot be a number
label@line                      ; invalid character
_12345678901234567890123456789toolong; too long
```

2.1.4 Operands

TIDI Instrument Command Language operands consist of the following general classes:

Table 1, TICL Operand Classes

class	description
1	Instrument Parameter Mnemonics
2	String Constants
3	numeric constants
4	hexadecimal constants
5	global variables
6	local variables

2.1.4.1 Instrument Parameter Mnemonics

Instrument parameters provide access to the state and configuration of the TIDI instrument. They are identified by a mnemonic, listed in reference 2. Those instrument parameters defined in reference 2 as commandable may be assigned values through the *store*, *inc*, *dec*, *add* and *sub* commands. Example instrument parameter mnemonics are:

```
Tel_4_Housing_Temp
Tel_2_Position
FW_1_Position
```

2.1.4.2 String Constants

String constants consist of alphanumeric characters. The TICL compiler is case insensitive. Examples of string constants are:

```
OFF
On
Open
CLOSed
```

Valid string operands for each command are specified in section 3 (page 29).

2.1.4.3 Numeric Constants

Numeric constants consist of the numerals and optionally one leading plus or minus sign, and one decimal point. Numeric constants are converted to an unsigned integer of appropriate length. Examples of valid numeric constants are:

```
-0.123
12
9812
+1.0
1.0E22
```

An examples of an invalid numeric constant is:

```
1ADE                          ; letters not permitted
```

2.1.4.4 Hexadecimal Constants

Hexadecimal constants consist of the numerals 0 through 9 and the upper or lower case letters A through F, with the letter H appended. Hexadecimal constants may be 1 through 4 digits long (5 digits when used as an address operand). Examples of valid hexadecimal constants are:

```
1234H
FfFfH
500A1H
```

Examples of invalid hexadecimal constants are:

```
1ADE                          ; no letter H suffix
1AZEH                         ; invalid digit "Z"
12ab567H                      ; too many digits
```

2.1.4.5 Engineering unit constants

Engineering unit constants provide the internal representation of instrument parameters. They are formed from the instrument parameter name and the engineering unit value as follows:

```
%InstrumentParameterMnemonic(value)
```

The leading per-cent sign (%) is required. The *InstrumentParameterMnemonic* may be any parameter specified in reference 2. Examples of valid engineering unit parameters are:

```
%CCD_Temp(-87.3)              ; CCD Substrate temperature
                              ; in degrees C
%tel1_position(12.56)         ; position of telescope one
                              ; in degrees
```

2.1.4.6 Global Variables

Global variables are a special class of instrument parameter mnemonics. Global variables may be assigned values through the *store*, *inc*, *dec*, *add* and *sub* commands. Global variables are identified by a simplified version of the mnemonic defined in reference 2:

```
global_nn
```

where "nn" is a number from 01 to 32, inclusive.

2.1.4.7 Local Variables

Local variables may be assigned names through the *local* (§ 3.2.2.7 p. 17) command. Local variable names must be valid identifiers (§2.1.3 page 10). The may have values assigned through the *store*, *inc*, *dec*, *add* and *sub* commands.

2.2 Directive Formation

Each compiler directive consists of an indicator character, a directive keyword, and an optional parameter. Compiler directives are formed as shown in the example below:

```
.keyword parameter
```

The directive keyword follows the period (".") indicator character. The directive keywords are defined in Section 4 (page 28).

3. Commands

Commands are formed from a command keyword and the required parameters. Commands perform one of the following classes of functions: arithmetic, program control, instrument control, or miscellaneous.

In the sections that follow, the command keywords are specified. For each command keyword, the required operands are also specified. The operands may be any of the 6 types defined in Table 1 (page 11). In the syntax tables that follow, the operand requirements are denoted as follows:

(C1,C2…)

Where the class specification "Ci" is a number indicating the class as specified in Table 1 (page 11). For example, the operand for the *jump* command is a statement label, a string constant, is denoted as (2). The first operand of the *sub* command is the destination of the result which can be an instrument mnemonic, a global variable or a local variable, and is denoted as (1,5,6)

3.1 Arithmetic

This section defines the commands that perform arithmetical operations on their parameters. Only integer arithmetic is supported. Constant source operands are converted as needed by the compiler to the internal representation of the destination operand used by the flight code.

3.1.1 Syntax

The following table defines the syntax of each of the arithmetical commands.

Table 2, Arithmetical Commands

keyword	operands	
	destination	source
Store	variable (1,5,6)	variable or constant (1,3,4,5,6)
Add	variable (1,5,6)	variable or constant (1,3,4,5,6)
Sub	variable (1,5,6)	variable or constant (1,3,4,5,6)
Inc	variable (1,5,6)	
Dec	variable (1,5,6)	

3.1.2 Semantics

3.1.2.1 Store

The *Store* command copies the value of the source operand to the destination operand. If the source operand is a numeric constant, it is converted by the compiler to the internal representation of the destination operand used by the flight code.

3.1.2.2 Add

The *Add* command adds the integer values of the source and destination operands and stores the result in the destination operand. If the source operand is a numeric constant, it is converted by the compiler to the internal representation of the destination operand used by the flight code. Assuming that the local variable tmpvar initially has a value of 10, the command

```
ADD tmpvar 2          ; example arithmetic
```

results in tmpvar having the value 12.

3.1.2.3 Sub

The *Sub* command subtracts the integer values of the source operand from the destination operands and stores the result in the destination operand. If the source operand is a numeric constant, it is converted by the compiler to the internal representation of the destination operand used by the flight code. Assuming that the local variable tmpvar initially has a value of 10, the command

```
SUB tmpvar 2          ; example arithmetic
```

results in tmpvar having the value 8.

3.1.2.4 Inc

The *Inc* command increments the integer value of the destination operand leaving the result in the destination operand. Assuming that the local variable loopcnt has a value of 1, the command

```
INC loopcnt
```

increases the value of loopcnt by one to two.

3.1.2.5 Dec

The *Dec* command decrements the integer value of the destination operand leaving the result in the destination operand. Assuming that the local variable loopcnt has a value of 7, the command

```
DEC loopcnt
```

decreases the value of loopcnt by one to six.

3.2 Program Control

This section defines the commands that TICL provides to suspend the execution of the program for a specified time, to define and to invoke subroutines and to declare local variables. The commands provided to control the flow of the program using the IF-ELSE, test at the top loop, and test at the bottom loop are defined in the section3.3 (page 18).

3.2.1 Syntax

The following table defines the syntax of each of the program control statements.

Table 3, Program Control Commands		
keyword	operands	
	destination	source
Wait	Centi-Seconds (3,4,)	
Program	Control Program ID (3,4)	
Call	subroutine name (2)	
Subroutine	subroutine name (2)	
Return		
End		
Local	local variable name (2)	initial value (3,4)

3.2.2 Semantics

3.2.2.1 Wait

The *Wait* command suspends control program execution for the time duration specified in the command operand. Times are specified in centi-seconds, with a resolution of 0.01 seconds. The following example causes a delay of 5.25 seconds

```
wait 525
```

3.2.2.2 Program

The *Program* command is used to set the Active Control Program ID instrument parameter. The parameter `Control_program_id` is the number that will be assigned to the instrument parameter Control_Prgm_Active_ID. It may be a number from 0 through 65535. The compiler will generate an error if this command and the *.immediate* directive are both present.

The command

```
Program 101
```

sets the parameter to 101.

3.2.2.3 Subroutine

The *Subroutine* statement indicates the beginning of a subroutine. The parameter specifies the name of the subroutine and must be a valid identifier (§2.1.3 page 10).

The compiler will generate an error if this command and the *.immediate* directive are both present.

3.2.2.4 Call

The *Call* command transfers control to a named subroutine. No parameters are passed, though global variables may be used to pass values to a subroutine. The compiler will generate an error if this command and the *.immediate* directive are both present.

The following code fragment illustrates the call, subroutine, return, and end statements:

```
.define FOREVER "0 .ne. 0"
.define DAYSIDE 1

subroutine nightmode
         if spacecraft_day_night_stat .eq. DAYSIDE
           return
         end_if

         load_scan_table nightscan.scan
         start_scan
         repeat
           wait 100
         until spacecraft_day_night_stat .eq. DAYSIDE
         return
end

subroutine daymode
         ;stuff
         return
end


program 100
;some initialization here
repeat
         call daymode
         call nightmode
until FOREVER
```

3.2.2.5 Return

The *Return* command transfers control to the statement following the CALL statement that invoked the subroutine containing the RETURN statement. Prior to the transfer of control, the return command also deallocates the local variables allocated within the subroutine. The compiler will generate an error if this command and the *.immediate* directive are both present.

3.2.2.6 End

The *End* statement indicates the end of a subroutine. Local variables declared within a subroutine are visible only between the subroutine and end statements. The compiler will generate an error if this command and the *.immediate* directive are both present.

3.2.2.7 Local

The *Local* statement allocates space for a local variable, initializes the variable to zero and assigns a name to it. Up to 255 local variables may be created within a subroutine. The compiler main-

tains a count of the number of local variables defined, so that they may be freed automatically at the return statement. Local statements must follow the subroutine statement and precede all other statements except comments. The compiler will generate an error if this command and the *.immediate* directive are both present.

3.3 Control Structure Commands

This section defines the commands that TICL provides to control the flow of the program using the IF-ELSE, test at the top loop, and test at the bottom loop. The commands provided to suspend the execution of the program for a specified time, to define and to invoke subroutines and to declare local variables are defined in the section3.2 (page15).

3.3.1 Conditional Expressions

TICL provides binary comparison operators for use in control structure commands. These expressions are of the form

```
<rvalue1> operator <rvalue2>
```

where each rvalue is the identifier of any variable or a numerical constant. The operators permitted are listed in Table 4.

Table 4, Comparison Operators

operator	meaning
.eq.	expression is true if rvalue1 equals rvalue2 expression is false if rvalue1 does not equal rvalue2
.ne.	expression is true if rvalue1 does not equal rvalue2 expression is false if rvalue1 equals rvalue2
.lt.	expression is true if rvalue1 is less than rvalue2 expression is false if rvalue1 is greater than or equal to rvalue2
.gt.	expression is true if rvalue1 is greater than rvalue 2 expression is false if rvalue1 is less than or equal to rvalue2

The expression in the following example is true if the instrument parameter *spacecraft_day_night_stat* is equal to one:

```
spacecraft_day_night_stat .eq. 1
```

The following expression is true if the position of telescope 1 is greater than the value in the local variable *parkpos*

```
tel_1_position .gt. parkpos
```

3.3.2 IF-ELSE-END_IF

The TICL language provide the *If-Else-End_If* conditional control structure statements. The *IF* statement takes 3 parameters which form a conditional expression.

3.3.2.1 Syntax

```
IF <rvalue1> .op. <rvalue2>
        ;statements executed if condition is true

ELSE
        :statements executed if condition is false

END_IF
```

3.3.2.2 Semantics

The commands following the *If* statement are executed if the condition is true. If the condition is false, control is transferred either to the statement following the *Else* command if it is present or the statement following the *End_If* command.

3.3.3 WHILE

The *While* command provides a loop, testing the exit condition prior to each execution of the TICL commands making up the body of the loop. The *While* command takes 3 parameters which form the conditional expression which must be true for the commands in the body of the loop to be executed.

3.3.3.1 Syntax

```
WHILE <rvalue1> .op. <rvalue2>
        ;statements to execute
END_WHILE
```

3.3.3.2 Semantics

The *While* structure functions by first evaluating the conditional expression. If the expression has the value of false, the statement following the *End_While* is executed. Otherwise, the commands between the *While* and *End_While* are executed. When the *End_While* is encountered, the process is repeated.

Within the scope of the *While* command, a *Continue* command causes the continuation condition to be evaluated, if it has the value of true, the command following the *While* is executed otherwise control is transferred to the statement following the *End_While*.

Within the scope of the *While* command, a *Break* command causes control to be transferred unconditionally to the first command following the *End_While* command.

If the conditional is false when the *While* is first encountered, the commands within the scope of the *While* are not executed at all.

3.3.4 REPEAT

The *Repeat* command provides a loop, testing the exit condition after each execution of the TICL commands making up the body of the loop. The *Until* command takes 3 parameters which form the conditional expression which must be true for the commands in the body of the loop to be repeated.

3.3.4.1 Syntax

```
REPEAT
        ;statements to execute
UNTIL <rvalue1> .op. <rvalue2>
```

3.3.4.2 Semantics

The *Repeat* structure functions by executing the commands within its scope once, prior to evaluating the conditional expression. If the expression has the value of false, the statement following the *Until* is executed. Otherwise, the commands between the *Repeat* and *Until* are executed again. When the *Repeat* is encountered, the process is repeated.

Within the scope of the *Repeat* command, a *Continue* command causes the continuation condition to be evaluated, if it has the value of true, the command following the *Repeat* is executed otherwise control is transferred to the statement following the *Until*.

Within the scope of the *Repeat* command, a *Break* command causes control to be transferred unconditionally to the first command following the *Until* command.

3.3.5 BREAK

The *Break* command unconditionally exits an iteration.

3.3.5.1 Syntax

```
REPEAT
        ;statements to execute
        IF <rvalue1> .op. <rvalue2>
          BREAK
        END_IF
        ;more statements to execute
UNTIL <rvalue1> .op. <rvalue2>
        ; statement executed when IF condition is true
```

3.3.5.2 Semantics

A *Break* command within the scope of a *While* or *Repeat* causes control to be transferred outside of the inner most enclosing loop.

In the example above, when the condition in the If statement is true, the program executes commands following the *Until*, otherwise the commands following the *End_If* are executed.

3.3.6 CONTINUE

The *Continue* command unconditionally skips the remainder of a loop.

3.3.6.1 Syntax

```
WHILE <rvalue1> .op. <rvalue2>
        ;statements to execute
        IF <rvalue1> .op. <rvalue2>
          CONTINUE
        END_IF
```

```
            ;other statements to execute
    END_WHILE
```

3.3.6.2 Semantics

A *Continue* statement within the scope of a *While* or *Repeat* causes the remainder of the body of the loop to be skipped, with the program continuing by testing the iteration continuation condition. In a *While* loop, a *Continue* has the effect of transferring control to the top of the iteration and in a *Repeat* loop a *Continue* has the effect of transferring control to the bottom of the loop.

In the example above, when the condition in the *If* statement is true, the statements following the *End_If* are skipped, effectively transferring control to the *End_While* statement, and the continuation condition is tested. If the condition is false, the statements following the *End_If* are executed.

3.4 Instrument Control

The following commands perform instrument control functions. These commands either directly act on a mechanism (*FilterWheel, CalLamp, Telescope*, or *Shutter*) or control the interpretation of a scan table (*Load_Scan_Table, Start_Scan, Stop_Scan, Stop_Scan_Now,* or *Load_Bin_Table*).

3.4.1 Syntax

The following table defines the syntax of each of the instrument control commands.

Table 5, Instrument Control Commands

keyword	operands	
	first	second
FilterWheel	Filter wheel number (3,4)	Filter wheel position (3,4))
CalLamp	WHITE1 WHITE2 NEON HAK OFF (2)	
Telescope	Telescope ID (3,4)	Telescope Elevation Angle (3,4)
Shutter	Telescope ID (3,4)	Open Closed (2)
Load_Scan_Table	file description (2)	
Start_Scan		
Stop_Scan_End		

Table 5, Instrument Control Commands		
keyword	operands	
	first	second
Stop_Scan_Now		
Load_Bin_Table	Bin Table Index (3,4)	file description (2)

3.4.2 Semantics

3.4.2.1 FilterWheel

The *FilterWheel* command changes the filter wheel position. The first parameter is the filter wheel number, either 1 or 2. The second parameter is the encoder position, ranging from 1 through 199.

The following example illustrates the use of this command:

```
FilterWheel 1 52     ; move fw 1 to encoder step 52
FilterWheel 2 18     ; move fw 2 to encoder step 18
```

3.4.2.2 CalLamp

The *CalLamp* command controls the calibration lamps. Only one lamp may be illuminated at a time.

The following example illustrates the use of this command.

```
CalLamp White1               ; illuminate only lamp white1
CalLamp OFF                  ; extinguish all lamps
```

3.4.2.3 Telescope

The *Telescope* command control the elevation angle of each telescope. The first parameter is the telescope ID number (Table 6). The second parameter is the angle in degrees to which the telescope is to be moved. Angles are measured in the telescope frame (reference 4), with zero being horizontal and 90 degrees being towards the nadir.

Table 6, Telescope ID Numbers		
telescope ID	package ID	nominal azimuth
1	A300	45
2	A301	135
3	A302	225
4	A303	315

The following example illustrates the use of this command.

```
Telescope 1 10.25
```

After the execution of this command, telescope 1 will be pointing 10.25 degrees below the spacecraft x-y plane.

3.4.2.4 Shutter

The *Shutter* command controls the position of the safety shutters in each of the telescopes. The first operand is the telescope ID number (Table 6). The second parameter is the position, either "open" or "closed". The following example closes all four telescopes:

```
shutter 1 closed
shutter 2 closed
shutter 3 closed
shutter 4 closed
```

3.4.2.5 Load_Scan_Table

The *Load_Scan_Table* command causes scanning to stop immediately and the current scan table to be replaced by the command table contained in the file (reference 3) specified in the command. The scan table is compiled and the binary version placed in the program. The following example results in scanning being halted and the scan table in the file nightmode.scan being compiled and loaded. The start_scan command causes scanning to be resumed.

```
load_scan_table "/tidi/scan/nightmode.scan"
start_scan
```

3.4.2.6 Start_Scan

The *Start_Scan* command causes the instrument to begin scanning under the control of the previously loaded scan table. Scanning begins at the first scan increment in the table.

3.4.2.7 Stop_Scan_End

The *Stop_Scan_End* command causes the instrument scan to halt after the last exposure of the last scan interval in the current table. All mechanisms are left in the state corresponding to the final exposure of the scan table.

3.4.2.8 Stop_Scan_Now

The *Stop_Scan_Now* command causes the instrument scan to at the end of the current CCD exposure. All mechanisms are left in the state they were in when scanning was stopped.

3.4.2.9 Load_Bin_Table

The *Load_Bin_Table* command causes scanning to stop immediately and the a CCD binning parameter table to be replaced by the table contained in the file specified in the command. The binning table to be replaced is indicated by the index number, which may be any number from 0 through 7, inclusive.. The binning table is compiled and the binary version placed in the program. The following example results in scanning being halted and the binning parameter table in the file nightmode.bin being compiled and loaded as bining table 5. The start_scan command causes scanning to be resumed.

```
load_bin_table 5 "/tidi/scan/nightmode.bin"
start_scan
```

3.4.2.10 Report_Globals

The *Report_Globals* command causes a Control Program Variable Dump TM packet to be dispatched, containing the values of all 32 control program global variables.

3.5 Miscellaneous Commands

The command defined in this section perform various housekeeping or software control functions. In several cases the commands are to be used only in the immediate mode.

3.5.1 Syntax

The following table defines the syntax of the miscellaneous commands.

Table 7, Miscellaneous Commands

keyword	operands		
	first	second	third
NoOp			
Boot			
NoBoot			
Load_Mem	Intel Hex Format file description (2)		
Write_Byte	address (4)	byte value (3,4)	
Write_Word	address (4)	word value (3,4)	
Write_Double	address (4)	word value (3,4)	word value (3,4)

Table 7, Miscellaneous Commands			
keyword	operands		
	first	second	third
Dump	address (4)	length (3,4)	
Calculate_crc	address (4)	length (3,4)	
Start_CP			
Stop_CP			
Run	address (4)		
Clear_CPH			
Load_CPH	file description (2)		
Validate_CPH			
Save CP	PRIMARY SECONDARY (2)		
Allow_WD_Expire			

3.5.2 Semantics

3.5.2.1 NoOp

The *NoOp* command has no effect on instrument operation.

3.5.2.2 Boot

The *Boot* causes the flight software boot code to execute the boot procedure, which initializes the instrument software. The boot code may be entered via the *Allow_WD_Expire* command. It is not intended for use in control programs. The compiler will generate an error message if this command is present without the *immediate* compiler directive.

3.5.2.3 NoBoot

The *NoBoot* command prevents an autoboot from occurring at the end of the Autoboot Time-out period. It is not intended for use in control programs. The compiler will generate an error message if this command is present without the *immediate* compiler directive.

3.5.2.4 Load_Mem

The *Load_Mem* command causes the contents of the Intel Hex format file specified by the command operand to be loaded into the flight computer memory. The compiler will generate an error message if this command is present without the *immediate* compiler directive.

3.5.2.5 Write_Byte

The *Write_Byte* command replaces the contents of the byte at the address specified by the first argument with the value specified in the second operand. The following example writes the value ABH to the first byte of the program RAM:

```
write_byte 20000H AB ; load value into address 20000H
```

3.5.2.6 Write_Word

The *Write_Word* command replaces the contents of the two bytes beginning at the address specified by the first argument with the value of the second operand.

3.5.2.7 Write_Double

The *Write_Double* command replaces the contents of the four bytes at the address specified by the first argument with the values specified in the remaining two operands. The following example writes zeros to the 4 bytes which start the EEPROM:

```
write_double 30000H 0 0
```

3.5.2.8 Dump

The *Dump* command causes the creation of one or more Memory Dump TM packets, containing the contents of memory beginning at the specified address. The number of bytes to dump is specified by the second operand. The following example causes the first 1000 bytes of program RAM to be dumped:

```
dump 20000H 1000     ; dump from 20000H to 203e8H
```

3.5.2.9 Calculate_Crc

The *Calculate_Crc* command causes the creation of a CRC TM packet containing the CRC calculated for the memory locations beginning with that specified by the first operand and continuing for the number of bytes specified in the second. The following example causes the first 1000 bytes of the program RAM to be checked:

```
calculate_crc 20000H 3E8H
```

3.5.2.10 Start_CP

The *Start_CP* command causes the current control program to be replaced by the pending control program stored in the control program holding buffer. Execution continues with the new control program.

3.5.2.11 Stop_CP

The *Stop_CP* command terminates execution of the current control program. Resumption of control program execution requires the transmission of an immediate command to the instrument via the 1553 bus.

3.5.2.12 Run

The *Run* command transfers control of the flight software to the address specified in the command. This has the effect of terminating the executing control program and exiting the flight code. This is a debugging aid. The compiler will generate an error if this command is present without the *.immediate* directive.

3.5.2.13 Clear_CPH

The *Clear_CPH* command clears the control program holding buffer, making it ready for loading with the *Append_CPH* command.

3.5.2.14 Load_CPH

The *Load_CPH* command replaces the control program holding buffer with the contents of the file. The file contains a string of bytes each represented by two hexadecimal characters separated by whitespace.

3.5.2.15 Validate_CPH

The *Validate_CPH* command stores the CRC calculated over the control program holding buffer in an instrument parameter and compares it to the stored value. If the computed and stored values do not agree an error report TM packet is transmitted.

3.5.2.16 Save_CP

The *Save_CP* command copies the contents of the control program holding buffer to one of two default control program regions in the EEPROM, as specified by the command operand. The control program holding buffer must be validated prior to invoking this command. For example, the following commands validate the holding buffer and saves the contents in the Primary Default Control Program area of EEPROM.

```
Validate_CPH
Save_CP Primary
```

3.5.2.17 Allow_WD_Expire

The *Allow_WD_Expire* command forces the flight computer to reset by allowing the watchdog timer. to expire. The compiler will generate an error if this command is present without the *.immediate* directive.

4. Compiler Directives

Compiler directives effect the operation of the compiler, but do not directly result in code generation. The following compiler directives are defined: *ident, include, define, immediate, purpose, tab_start*, and *tab_end*. Inclusion of compiler directives in a TICL source file is optional.

4.1 Include Directive

The *Include* directive supplies the name of a file whose contents are inserted into the control program, replacing the directive. The directive has the following form:

```
.include file_path
```

The parameter file_path is a UNIX path name. Example valid directives are:

```
.include /tidi/sequences/TICL/baseline.ticl
.include param_defs.hicl
```

4.2 Define Directive

The *Define* directive associates a target token with a replacement. Subsequent occurrences of the target token are replaced with the specified string, matching is done without consideration of case. This occurs before the source code translation begins. The directive has the following form:

```
.define  target replacement
```

Example valid directives are:

```
.define    safpos 15.25
.define    lamp  white1
```

In the first example any subsequent occurrence of the token "safpos" is replaced by the string "15.25". In the second, any occurrence of the string "lamp" following the directive are replaced by the string "white1". Token substitution does not occur within quoted strings.

4.3 Immediate Directive

The *Immediate* directive indicates that the TICL code in the source file is to be compiled and transmitted as immediate 1553 commands, not to be loaded in the control program holding buffer.

If the *Immediate* directive is present as the first non-comment, non-blank line in the source code file, jump, subroutine, call, return and end commands will be ignored by the compiler, with error messages logged. If the *Immediate* directive is not the first non-comment, non-blank line an error will be reported.

4.4 Purpose Directive

The *Purpose* directive supplies descriptive information that the compiler passes along to subsequent uplink processing stages. It is used to supply a text string of up to 132 characters for inclusion in the command message header. The directive has the following form:

```
.purpose   "description of program purpose"
```

The description must be supplied as a quoted string if it includes spaces and can be continued on multiple lines by including additional purpose directives. The parameter supplied on the second and

subsequent purpose directives is appended to the end of the previously supplied string. For example, a short description:

```
.purpose "perform the daily calibration"
```

A long description:

```
.purpose "during the day time blah blah blah"
.purpose " and during the night blah blah blah "
.purpose "and blah blah blah"
```

results in the following purpose string begin passed to latter steps:

```
during the day time blah blah blah and during the
night blah blah blah and blah blah blah
```

Descriptions longer than 132 characters will be truncated by subsequent processing of command messages.

4.5 Scan_Table_Start and Scan_Table_End Directives

The scan table delimiting directives, *Scan_Table_Start* and *Scan_Table_End*, are used to identify the beginning and end of an inline scanning table. The *Scan_Table_Start* directive changes the compiler operating state so that subsequent lines are interpreted as scan table lines according to reference 3. The *Scan_Table_End* directive terminates the scan table compilation and returns the compiler to its original state.

4.6 Bin_Table_Start and Bin_Table_End Directives

The binning table delimiting directives, *Bin_Table_Start* and *Bin_Table_End*, are used to identify the beginning and end of an inline binning table. The *Bin_Table_Start* directive changes the compiler operating state so that subsequent lines are interpreted as scan table lines according to reference 3. The *Bin_Table_End* directive terminates the binning table compilation and returns the compiler to its original state.

A Command Block File Format

A.1 Introduction

The command block file consists of a series of ASCII records, separated by the new-line character. The first several records form the file header. Following the header are packaging directives which specify how the TIDI Commands in the command block are to be packaged into a TIMED command message. The final portion of the file contains the TIDI commands, in hex format.

Header and packaging directives begin with a keyword. A space separates the keyword from its value.

A.2 Header

The records that make up the header are defined in Table 8.

Table 8, Header Contents

re-cord	keyword	acceptable values	description
1	.FFVID	number	File format version ID
2	.TICL	absolute file path	TICL source code file
3	.TCMD	absolute file path	TICL command block file name
4	.CTIME	time string	Creation Time
5	.CNODE	string	Creation Node
6	.CCMD	string	Creation command

The file format version identifier (.FFVID) is intended to match the file with the reading program or library. It is to be changed whenever a new release of the reading software is required. The second header record (.TICL) contains the absolute path name of the TIDI Instrument Command Language source file that was compiled to create the command block file. The next item, .TCMD, contains the absolute path name of the command block file containing the header at the time it was created. The .CTIME record contains the time at which the compiler was run is the TIMED standard ASCII format with fractional seconds omitted. The .CNODE command contains the name of the data processing system node on which the program was run. The final item, .CCMD, records the command used to invoke the compiler.

A.3 Packaging Directives

Packaging directives (Table 9) supply information that will be used by the command message formatting process. These directives either specify processing options to be applied to the message formatting or values to be inserted into the command message header.

Table 9, Packaging Directives

keyword	acceptable values	notes
.PURPOSE	ASCII text string	data specified by the .purpose compiler directive
.TYPE	`immediate` `stored`	indicates whether the command block is to be executed upon receipt or stored in the control program holding buffer

The .PURPOSE packaging directive transmits the descriptive information supplied by the purpose compiler directives to subsequent processing steps. The .TYPE directive is set to the ASCII string "`immediate`" if the .IMMEDIATE compiler directive was present otherwise it is set to the ASCII string "`stored`".

A.4 Command Block

The command block contains the compiled instrument commands. Each byte of command text is encoded as two hexadecimal digits. Bytes are delimited by spaces. Up to 16 bytes may be included on a line. If a command is longer than 16 bytes, the line is terminated with a hyphen and a new line character. If the command is less than 16 bytes in length, it is terminated with only a new line character. No more than one command may be started on a line.

B Error Codes

Error Message Text Explanation and remedial Action
'something' is an illegal real constant 'something' is an illegal hexadecimal constant 'something' is an illegal decimal constant See the compiler spec. for rules for forming numeric constants.
'something' is an unknown science data mode. Valid modes are BINNING and IMAGE. Either of these can be abbreviated down to the first letter, but must match (ignoring case) as many characters as supplied.
'something' is not a comparison operator. Valid comparison operators are .lt., .gt., .eq., and .ne.
'something' is not a legal local variable name. See the Compiler Spec. for legal identifier naming rules.
'something' is not a local variable name. An attempt to use a non-number in a STORE or math operation, when that symbol is neither an instrument parameter nor a declared LOCAL variable.
'something' is not a recognized calibration lamp name. Valid lamp names are HAK, NEON, WHITE1, WHITE2, and OFF.
'something' is not an instrument parameter The compiler encountered an identifier that begins with %, indicating a conversion constant. However, the name following the % is not a recognized instrument parameter. Check spelling.
(internal) Bad size for symbol reference. Symbol references can be either 1 or 2 bytes long. This indicates one that was neither of these. This should never happen.
0 step counts in calculation A scan table specifies an IDR for which the number of steps is 0.
At end-of-program, Missing END_IF, UNTIL, or END_WHILE Some scope was still open at end-of-input. This probably indicates a missing END of some sort.
Attempt to END subroutine in the middle of IF/WHILE/REPEAT END should only come after all other non-subroutine scopes are closed within its subroutine.
Attempt to redefine 'symbol' An attempt was made to define the symbol using a .define, SUBROUTINE, or LOCAL command, when there is already such a symbol defined.
Bin Table gain must be in [1..4] The gain for each bin must be in the given range.

Error Message Text Explanation and remedial Action
Bin table number must be 0 to N inclusive. You used M. The binning table number supplied with clear_bin_table, append_bin_table, or load_bin_table was out of range. The maximum value for this number is defined as MAX_BIN_TABLE_NUMBER at the top of ticlBinTable.c As of this writing, the legal values were 0 to 7.
Bin Table number must be between 1 and N (inclusive) A binning table number supplied with CLEAR_BIN_TABLE, LOAD_BIN_TABLE, or as part of a scan table IDR is out of range.
Can't add 'symbol' to dictionary. Too many symbols. The maximum size for a symbol table is defined by the constant MAX_DICTIONARY_SIZE, defined at the top of ticlDictionary.h
Can't read the parameter file 'filename'. The file given by the INSTRUMENT_PARAMETER_FILE environment variable couldn't be opened for reading.
Checksum error in Intel hex file The last byte of every line must be the 2's compliment value of the sum of all the other data on the line, truncated to 8 bits.
Couldn't open bin table file 'whatever' The LOAD_BIN_TABLE command was given a file that could not be opened for reading.
Couldn't open byte file 'whatever' The LOAD_CPH command was given a file that could not be opened for reading.
Couldn't open include file 'whatever' The file for the given .include directive couldn't be opened for reading.
Couldn't open Intel Hex file 'whatever' The LOAD_MEM command was given a file that could not be opened for reading.
Couldn't open scan table file 'whatever' The LOAD_SCAN_TABLE command was given a file that could not be opened for reading.
ELSE without matching IF ELSE was found outside of an IF scope.
END without SUBROUTINE. END was found without a matching SUBROUTINE.
END_IF without IF. END_IF doesn't match an IF or ELSE. END_IF was found outside of an IF scope.
END_WHILE without WHILE. END_WHILE doesn't match a WHILE. END_WHILE was found outside of a WHILE scope.

Error Message Text Explanation and remedial Action
Error writing P-Code to file. Probably because the disk is full.
Exposure count must be between 1 and N (inclusive) Exposure count is the number of integrations that are performed in one scan step.
Found N lines in the config file instead of M. The TICL configuration file needs to have nothing but numbers in it. Those numbers, and the format they are expected to be in, is documented in the installation guide. This message indicates that there is an error in that file.
Illegal character 'c' in hex record. Intel hex files should only contain hexadecimal numbers, colons, and carriage returns.
Illegal compiler mode change. Cannot go from state N to state M. This usually indicates an extra .immediate or PROGRAM command, but can occur in other rare circumstances. The state numbers given can be decoded by looking at the **`CompilerState`** enumeration in ticlState.h. Legal state transitions are given in ticlState.c
Illegal subroutine name 'something' in CALL. 'something' is not a valid subroutine identifier. See the Compiler Spec. for legal identifier naming rules.
Internal Error - Unknown compiler state N. An attempt was made by the compiler guts to enter an unknown (not merely illegal) state. This should never happen.
Internal error! N passed to compileByteAt. Internal error! N passed to compileWordAt. Something in the compiler's guts called an internal routine and passed an offset beyond the legal range. Like other internal errors, this should never happen.
Internal Error. Unsupported length in numeric constant. The internal routine getNumericValue returned a bad value. This should never happen.
Internal Error: Opening unknown Scope Type Internal Error: Trying to end unknown scope. Internal Error: Closing unknown scope type. The compiler called the open/close NewScope() function with an invalid parameter. This should never happen.
Internal Error: Too many lines of P-Code. The maximum number of source lines allowed in a TICL program is defined by the MAX_SOURCE_LINES symbol at the top of the file ticlCompiler.c. This number includes all the lines from included files.
Internal error: Unknown parameter type. A parameter type other than L, R, A, W, or B was passed to compileParameters. This should never happen.

Error Message Text Explanation and remedial Action
Invalid bin table Interval Definition Record. The wrong number of columns was supplied for this binning table record.
Invalid compiler state N during writeHeader2 Somehow, the compiler was finishing the object header when it should have still been parsing code. This can be caused by compiling an empty file, or by ending input during a scan table or a binning table. In any case, it is bad.
Invalid record type in INTEL hex file The record types supported by the LOAD_MEM command are 00, 01, and 02.
Invalid scan table Interval Definition Record. Partial or invalid IDR found when a complete one was expected. Illegal IDR found when expecting a partial one. Too many or too few values supplied in an IDR line. Also occurs when an unknown directive is encountered in a table.
LOCAL is only valid within a subroutine. Since space for local variables needs to be allocated and deallocated, and since there is no mechanism for deallocating variables at the end of the main part of a program, it is illegal to use the LOCAL command outside of a subroutine.
Malformed bins in Bin Table record. A bin record in the binning table had a start pixel larger than its end pixel, or a start pixel that was not exactly one greater than the end pixel of the previous bin.
Maximum binning table size exceeded. A binning table (when compiled) must not be larger than MAX_BIN_TABLE_SIZE, defined at the top of ticlBinTable.c. As of this writing, that's 2048 bytes.
Maximum of 255 local variables in one subroutine has been exceeded. Congratulations! They said it couldn't be done. Now go and write smaller, simpler subroutines.
Maximum p-Code size of N bytes exceeded. The maximum number of bytes that a compiled TICL program can be is defined by the MAX_PCODE_SIZE symbol at the top of the file ticlCompiler.c.
N is an invalid filter wheel position. Valid positions are 1, 2, 3, 4, 5, 6, 7, and 8. Note that this message is generated for scan tables only.
Nested subroutine. Missing END before this? Found a SUBROUTINE statement within another scope.
No loop structure to BREAK out of. A BREAK was encountered outside the scope of any WHILE or REPEAT loop. This is nonsensical.
No loop structure to CONTINUE in. A CONTIUNUE was encountered outside the scope of any WHILE or REPEAT loop. This is nonsensical.

Error Message Text Explanation and remedial Action
Parameter of binning table .DISPOSE record must be 'read' or 'discard', not 'whatever'. A bad parameter was found in the .dispose record of a binning table.
Path length exceeds N-character maximum. File paths have a maximum length defined by the MAX_PATH_LENGTH symbol at the top of ticlStreams.h
PROGRAM statement must occur at outer-most scope. Are you missing an END? The PROGRAM command was encountered when some scope (e.g. SUBROUTINE, IF/ELSE, WHILE, REPEAT) was still open.
Reference to unresolved label 'symbol' at offset XXXX Each unresolved label has those places where it is referenced written out so that some effort will reveal exactly where the problem is.
Scan table exceeds maximum size. The maximum size (in bytes) of a compiled scan table is defined by the MAX_SCAN_TABLE_SIZE symbol, at the top of ticlScanTable.c
Scan Table is missing some telescope information. Each input record must specify motion for all 4 telescopes. The compiler found a record that didn't. For example a record that specified telescope 'W' followed by another record that specified telescope 'W' would generate this error.
Scope Nesting exceeds N levels. The maximum level of nesting of SUBROUTINE, IF, WHILE, and REPEAT statements is defined by the symbol MAX_SCOPE_NESTING at the top of ticlScope.c.
something is an illegal value for a 24-bit address. something is an illegal value for a 16-bit constant. something is an illegal value for an 8-bit constant. TICL addresses must be in the range 0 to FFFFFFH. 16-bit constants must be in the range 0 to FFFFH, 8-bit constants must be in the range 0 to FFH.
SOME_COMMAND expects N parameters and got M. Self-explanatory. Check the user's guide and correct the program.
Specified step-size of %f does not evenly divide the angle range of XX.XXX. The Remainder is N steps. The step size must evenly divide the angle range.
Step size is too small (or angle range is too big). In a scan table, there are at most 255 steps that can be done in one IDR. If you specify an angle (or altitude) range that cannot be divided by the specified step size in less than 256 steps, the table is invalid. Try using two or more IDRs for the altitude range instead.
Step size of XX.XX degrees is too big to encode in one byte. Use two or more steps. The given step size is more than 127 * 0.005 degrees or less than -128 * 0.005 degrees.

Error Message Text Explanation and remedial Action
Subroutine "name" has already been defined Subroutine names must be unique within a compilation unit.
The angle XX.XX is beyond the legal range for the telescope elevation. As it says. The angle given should be in degrees, remembering that there is a built-in offset defined in the TICL_CONFIG_FILE (default of 20.35 degrees).
The filter wheel must be in [0..199], not N Unlike the scanning table, the FILTERWHEEL command uses encoder counts rather than filter number.
The instrument parameter 'parameter' is not a valid target for assignment A command that requires an L-Value was given an instrument parameter that is marked read-only in the spreadsheet.
The SHUTTER must be "Open" or "Close", but is 'something'. Self-explanatory.
The SW Storage Size supplied for 'parameter' in the spreadsheet is N. It must be one of: 8, 16, or 32. There is an invalid line in the spreadsheet file.
The Telescope code X is illegal. It must be 'A', 'C', 'W', 'F', 'B', '1', '2', '3', or '4'. Check telescope identifier in the scan table's IDR.
The telescope number must be between 1 and 4 (inclusive). You used N. Self-explanatory.
The WHATEVER command is not allowed in a non-IMMEDIATE program. STORED programs may not contain RUN, BOOT, NOBOOT, LOAD_CPH, or LOAD_MEM commands.
The wheel number must be 1 or 2. You used M. Invalid filter wheel number.
Too many tokens on line. Usually indicates forgotten " around a string constant like the purpose string.
Too many unresolved labels. Unresolved symbols are stored in a dictionary, which is bound to a maximum size of MAX_DICTIONARY_SIZE. If you get this error, either simplify the program by getting rid of CALLs, or else increase MAX_DICTIONARY_SIZE.
Unexpected end of scan table. The compiler was parsing an incomplete Interval Definition Record when it found end-of-table.
Unrecognized command: 'something' The token field of a source line was not a recognized TICL command or directive. This is usually the result of a typo, but it can also happen if the compiler is in the wrong mode. For example, a SUBROUTINE command will produce this error if the .IMMEDIATE directive is the first command in the source file.

Error Message Text Explanation and remedial Action
Unresolved label 'symbol' A reference was made to the given symbol, but that symbol was not resolved by the time the end of the program was reached. If the symbol begins with an '@' sign, then this error indicates a problem with scoping within the compiler. Each label means something a little different, as described: @AfterUntil_xxx - Missing UNTIL in a REPEAT loop @NumLocals_xxx - Missing END for a SUBROUTINE @AfterWhile_xxx - Missing END_WHILE for a WHILE loop @UNTIL_xxx - Missing UNTIL in a REPEAT loop that contains a CONTINUE. @AfterIF_xxx - Missing END_IF or ELSE after IF. @AfterELSE_xxx - Missing END_IF after ELSE. These two indicate a more serious internal compiler error: @REPEAT_xxx - Should never happen @TopOfWhile_xxx - Should never happen
Unterminated String Missing " at end of string constant.
UNTIL without REPEAT. UNTIL doesn't match a REPEAT. UNTIL was found without a matching REPEAT.
You need to define the 'INSTRUMENT_PARAMETER_FILE' environment variable. This environment variable is described in the installation guide.

C Compiled Program Example

The source code:

```
.define FOREVER "0 .ne. 0"
.define DAYSIDE 1

subroutine nightmode
   if spacecraft_day_night_stat .eq. DAYSIDE
      return
   end_if

   start_scan
   repeat
      wait 100
   until spacecraft_day_night_stat .eq. DAYSIDE
   return
end

subroutine daymode
;; stuff
   return
end

program 100
;; some initialization here
   repeat
      call daymode
      call nightmode
   until 0 .ne. 0
```

Produces the following .tcmd file:

```
.FFVID 1.0
.TICL /home/rowe_s/contracts/SPRL/TIDI/test/oldStuff/example.ticl
.TCMD /home/rowe_s/contracts/SPRL/TIDI/test/oldStuff/example.tcmd
.CTIME 1998185160824
.CNODE deepthought
.CCMD ticl example
.PURPOSE
.TYPE stored
08 69 00 29 01 -
17 41 e3 01 09 0f 00 -
2a 01 0f -
21 19 04 1a 04 12 34 12 65 01 0a 04 64 01 0a 04 -
64 01 0a 04 6e 02 0a 04 -
21 1d 1e 25 55 76 e1 09 e1 09 e1 09 e1 09 02 01 -
40 00 00 78 e8 03 00 00 00 00 00 00 33 40 00 00 -
78 e8 03 e8 03 d8 d8 d8 d8 -
1f -
0d 64 00 -
17 41 e3 01 09 51 00 -
2a 01 0f -
2a 01 0f -
29 01 -
2a 01 0f -
2a 01 0f -
12 41 b1 64 -
0e 61 00 -
0e 03 00 -
17 44 00 00 0a 6d 00
```

D Immediate Sequence Example

This immediate command sequence:

```
.immediate

stop_cp
clear_cph
load_cph "bytes.dat"
load_mem "intelData.dat"
```

Produces the following .tcmd file:

```
.FFVID 1.0
.TICL /home/rowe_s/contracts/SPRL/TIDI/test/oldStuff/immediate.ticl
.TCMD /home/rowe_s/contracts/SPRL/TIDI/test/oldStuff/immediate.tcmd
.CTIME 1998185160917
.CNODE deepthought
.CCMD ticl immediate
.PURPOSE
.TYPE immediate
11
18
11 18 1b f6 08 a3 00 29 02 12 64 01 09 00 12 64 -
00 0a 00 13 64 00 07 00 14 64 01 03 00 16 04 01 -
15 04 01 22 01 63 0d 0d 02 18 04 cd ab 00 00 01 -
01 00 05 10 1f 2a 02 0f 11 20 21 24 71 0b 1c 17 -
64 00 0a 00 0c 48 00 00 00 08 4a 00 00 00 12 64 -
00 00 00 17 64 00 0a 00 0c 5e 00 00 15 04 00 08 -
4f 00 12 64 00 00 00 00 15 04 00 17 64 00 09 00 -
0b 63 00 00 12 64 00 00 00 17 64 00 0a 00 0c a0 -
00 00 15 04 00 12 64 01 00 00 00 15 04 01 17 64 -
00 08 00 0b 95 00 08 9d 00 17 64 01 09 00 0b 86 -
00 08 75 00 2a 02 0f 12 62 00 b1 05 00 25 00 23 -
01 0e 03 00 23 03 0e 03 00 23 04 0e 03 00 23 00 -
25 04 08 a3 00 29 02 12 64 01 09 00 12 64 00 0a -
00 13 64 00 07 00 14 64 01 03 00 16 04 01 15 04 -
01 22 01 63 0d 0d 02 18 04 cd ab 00 00 01 01 00 -
05 10 1f 2a 02 0f 11 20 21 24
1b 86 71 0b 1c 17 64 00 0a 00 0c 48 00 00 00 08 -
4a 00 00 00 12 64 00 00 00 17 64 00 0a 00 0c 5e -
00 00 15 04 00 08 4f 00 12 64 00 00 00 00 15 04 -
00 17 64 00 09 00 0b 63 00 00 12 64 00 00 00 17 -
64 00 0a 00 0c a0 00 00 15 04 00 12 64 01 00 00 -
00 15 04 01 17 64 00 08 00 0b 95 00 08 9d 00 17 -
64 01 09 00 0b 86 00 08 75 00 2a 02 0f 12 62 00 -
b1 05 00 25 00 23 01 0e 03 00 23 03 0e 03 00 23 -
04 0e 03 00 23 00 25 04
03 00 00 00 10 00 01 02 03 04 05 06 07 08 09 0a -
0b 0c 0d 0e 0f
```

E TICL Design Standard

E.1 Purpose and Scope

Syntactically correct TICL code can be written which may be dangerous to the instrument, may perform unneeded operations, or may violate assumptions used in the analysis of the spectra collected. This design standard is intended to ensure that all TICL code is both syntactically and operationally correct.

This standard defines the permissible ways to implement mechanism motions. It defines the use of TICL global variables. Also specified are TICL commands that are operationally unusual, and should not normally be used.

All TICL code written for operational use must be designed to this standard.

E.2 Mechanism Restrictions

E.2.1 Calibration Lamps

Calibration Lamps should be left illuminated for no longer than necessary.

Within a scan table, adjacent entries my not have different calibration lamps illuminated. There must always be one scan table entry with the calibration lamps off between any two states with different lamps illuminated.

When changing lamp state using the CalLamp command, the lamp must be turned off prior to illuminating another lamp. The following sequence is permissible:

```
CalLamp Neon
CalLamp Off
CalLamp Hak
```

An impermissible sequence , because the lamps are switched directly from one being on to another being on is:

```
CalLamp White1
CalLamp Hak
```

E.3 TICL Global Variable Use

The control program globals `GLOBAL_01, GLOBAL_02 … GLOBAL_32` may be used to send special messages to the ground, to store state information or to act as subroutine arguments. To prevent conflicts the following conventions for control program globals shall be followed.

Control program global `GLOBAL_01` shall be used to specify the process identifier.

Control program globals `GLOBAL_01, GLOBAL_02 … GLOBAL_11` shall be used to contain message information to be transmitted to the ground with the `Report_Globals` command.

Control program globals `GLOBAL_12, GLOBAL_13, … GLOBAL_22` shall be used as state memory. Table 10, Use of Globals for State Memory, contains the assignments. The symbolic names, which should be used to make the TICL more readable, are defined in the include file `dayDrivers.inc`.

Table 10, Use of Globals for State Memory		
global	symbolic name	description
global_12	`TERM_COUNT`	Count of the number of terminator crossings since 00:00 UTC. Shall be maintained by the daily TICL driver.
global_13	`CURRENT_DAY`	Stores the value of `time_utc_day_number` at the when the daily TICL driver starts. Used to determine when the day has changed.
global_14	`DAY_SIDE`	Stores the value of `spacecraft_day_night_stat` when invoking a science table. Used to determine when the terminator has been crossed.
global_15		*unassigned*
global_16		*unassigned*
global_17		*unassigned*
global_18		*unassigned*
global_19		*unassigned*
global_20		*unassigned*
global_21		*unassigned*
global_22		*unassigned*

The remaining control program globals, `GLOBAL_23, GLOBAL_24 … GLOBAL 32` shall be used as subroutine arguments. Subroutines will preserve their arguments in local variables and restore them from local variables prior to returning control to the calling program.

E.4 Operationally Restricted Commands

The following TICL commands are restricted because they can result in a loss of science data:

The *STOP_SCAN_END* command should not be used if the scan table will have more than 5 states before ending. The execution of this command locks out all immediate commands from the 1553 interface until the scan table has been completed. An alternative way to execute a scan table once is as follows:

```
start_scan
while sys_expose_down_count .gt. 5
        wait 250
end_while
```

F TICL Coding Standard

F.1 Purpose and Scope

F.2 Program Header

F.3 Internal Documentation

F.4 Statement Construction

F.5 Logical Structure

F.5.1 General

F.5.2 Modularization

F.5.3 Control Structures

F.6 Conventions