University of Michigan Space Physics Research Laboratory		
TIDI **Instrument Language Compiler** **Design & Maintenance Document**	CAGE No. Drawing No. Project Contract No. Page	0TK63 055-3633 TIDI NASW-5-5049 Page 1 of 15

REVISION RECORD

Rev	Description	Date	Approval
	Draft (Internal Release Only)	1 Jun 1998	
	Draft	2 July, 1998	
A	Official Release	4 July, 1998	

APPROVAL RECORD

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

Table of Contents

List of Tables

List of Figures

1. References

1. Musko, S., "TIDI Flight Software Requirements Specification", SPRL File 055-3320, 15 January 1997

2. Gell, D., "TIDI Instrument Command Language Compiler Specification and User's Guide", SPRL File 055-3564, 5 May 1998

2. Introduction

The purpose of this document is to educate the maintenance programmer about the TICL Compiler so that s/he can:

- Modify/extend the compiler in the event of TICL language changes without breaking the underlying design or weakening the underlying structure.
- Quickly identify which files to look in for the answers to technical questions about the compiler.
- Correct any errors that are found during the lifetime of the compiler.
- Use it as a model for writing other, similar tools.

2.1 Intended Audience

This document assumes that the reader is a programmer with a good working knowledge of the "C" programming language, and (to a lesser extent) the UNIX operating system, and the ability to read simple syntax diagrams. No special knowledge of compiler theory is required.

2.2 Vocabulary:

Language elements can be grouped into seven categories:

Compiler Directives: These commands do not produce any executable code. They alter the behavior of the compiler is some way.

Simple Commands: These commands require the generation of a command byte and possibly the generation of encoded parameter bytes. They are "simple" in that they can be handled in a generalized way with a small amount of code and a moderate data table.

Complex Commands: The commands have special needs that disqualify them from the generic handling that is used with simple commands. Examples are commands with string parameters instead of integer parameters, commands that require the opening of files, and commands whose parameters require formatting other than "raw" bytes.

Control Structures: These commands translate into a series of comparisons, labels, and conditional jumps to those labels. They do not affect instrument operations.

Parameters: Since most commands have parameters, and there is a measure of commonality with how those parameters are parsed and compiled, it makes sense to treat parameters as language elements in their own right. In addition, the instrument parameters need to be treated specially because of translation from user-units to counts, read-only versus mutable and other table-driven attributes.

3. Modularity

To manage the complexity of the system, it was broken into functional blocks. The functions in each functional area are collected into a single source file, with those services that need to be used by other functional blocks externalized as prototypes in the appropriate header file. The functional blocks in the system, and their associated source files are listed in Table 1.

Table 1
Functional Block Division by Source File

Functional Block Description	Files containing module
Main program, command-line argument handling	Ticl.*
Signal Handling	TiclSignal.*
File Handling	TiclStreams.*
Source Line scanning/parsing	TiclParse.*
Parsed line dispatching	TiclCommands.*
Placing bytes into the p-Code stream	TiclCompiler.*
Instrument Parameter Compilation	TiclParameters.*
Execute compiler directives	TiclDirectives.*, ticlState.*
Scan table compilation	TiclScanTable.*
Binning table compilation	TiclBinTable.*
Symbol Table Handling	TiclDictionary.*, ticlLabels.*
Control Structure compilation	TiclScope.*
Utility Functions	TiclUtils.*

4. Theory of Operation

The job of the compiler is to translate source tokens into the byte-code commands and parameters that the flight software expects. More complex languages require lexical scanners to identify language tokens, and parsers to assemble these tokens into legal syntactic blocks. Literature like Aho, Sethi, and Ulman (the so-called Dragon book) discuss compilation theory in great detail. Fortunately, this is completely unnecessary in the case of the TICL compiler, since its operation can be summarized in a few lines of pseudo-code as shown in Figure 1.

Figure 1
Pseudo-Code of Compiler Operation

```
open files
set up signal handlers

while there are more lines
  read a line
  parse the line into tokens
  echo it to the listing stream
  compile it, writing error/warning messages as needed
end while

resolve labels

if no errors occurred
  write p-code out to the output file
end if

write compilation statistics summary to listing
close all files
```

This is the essence of the entire program. Do not be daunted by the name "compiler"; this is a straightforward data translation program. Section 4.1, 4.1, gives a more detailed description of each of the lines in Figure 1.

4.1 Program Structure

The following sections provide a more detailed expansion of the pseudo-code given above.

4.1.1 open files

Files in the TICL compiler are augmented stdio FILE structures called IOStream. The IOStream structure stores the FILE* of the file, the fully expanded name of the file, and (if reading) the current line number of the file. This information is used in reporting error message and in writing the listing file. All files are opened right up front, so that if there is a problem, we don't try to go through the compilation process with bad files.

4.1.2 set up signal handlers

Using the POSIX signal handling mechanism, traps are installed to catch those signals that can reasonably be handled by deleting the object and listing files and exiting cleanly. Note that certain signals cannot be trapped, and others should not. See the source file ticlSignals.c for a complete discussion of this.

4.1.3 read a line

The source input files are stored on a stack. This allows .include directives to be handled fairly easily. When end-of-file is reached on the current file, the readLineFromSourceStream function transparently closes the file, pops the stack, and continues reading from the next file. Only when all the files on the stack have been closed does this function return an EOF condition.

4.1.4 parse the line into tokens

The grammar for TICL is so simple that "parse" is probably an overstatement. Since a TICL command/directive is contained entirely on a line, all that is required is to break the line into tokens and store it in a structure called ParsedLine. The line is broken up by strtok using whitespace characters space (ASCII 32), tab (ASCII 9), and newline (ASCII 10). There are two exceptions. First, the semicolon character (;) that short-circuits line processing and causes everything from the semicolon on to be taken as one token and stored in the comment field of the parsed line structure. Second, the double-quote (") character is used to delimit constant strings. Everything between the quote characters is taken as a single token, with the quote characters discarded. Note that this means that commands that expect strings only have to use quotes if the string contains whitespace.

4.1.5 echo it to the listing stream

This serves two purposes. First, it gives the user the context for error messages. Second, it gives an inkling as to what's going on inside the compiler by providing the p-code offset for the beginning of the line and echoes the tokens AS THEY HAVE BEEN SUBSTITUTED by their .defined alter-egos, rather than how they appear in the source file itself. If you are ever stumped as to why a syntax error occurs, look in the listing file and see what the compiler is *really* looking at.

4.1.6 compile it

There is a lot of code hiding behind "compile it." Recall that there are actually 4 languages being compiled. The compiler keeps track of which language is being compiled by using a state machine (see diagram). The STORED_STATE and IMMEDIATE_STATE are similar in that they are compiling languages with 80% overlap. The SCAN_TABLE_STATE and BINNING_TABLE_STATE, as one might expect, shunt the compilation over to the table handling routines rather than the TICL compilation routines. As a simplification, compiler directives (other than .end_bin_table and .end_scan_table) are only executed in NEUTRAL, STORED and IMMEDIATE states.

Having selected the compilation function based on the current Compiler State, one of the several line compilation functions is called. For further details of this process, see the Compilation Details section, below.

4.1.7 writing error/warning messages as needed

Error and warning messages are handled by the functions writeError and writeWarning. The messages are variable argument printf-style messages, and are routed through these routines so that statistics and common formatting can be enforced. The format chosen for messages is such that the emacs editor can be used to parse the error messages and assist in debugging TICL programs.

4.1.8 resolve labels

Early drafts of the command language had labels and JUMP instructions included. This was later revised to high-level control structures IF, WHILE, and REPEAT. However, these control structures still generate labels and JUMPs. Often, the JUMPs are generated before the labels are, so the exact offset to JUMP to is unknown. These unresolved references are stored in a table, and are resolved after the entire program has been compiled (if not before). Functions for handling references are in the ticlLabels module. For details on control structure compilation, see the Compiling Control Structures

Another use for the unresolved label mechanism is to store the number of LOCAL variables that need to be allocated in each subroutine. The number of LOCAL variables is unknown when the SUBROUTINE command is compiled and needs to compile the ALLOCATE command. Likewise for RETURN, where DEALLOCATE commands need to be generated. For these cases, the compiler generates a unique symbol for each subroutine, the value of which is filled in when the END command is compiled.

4.1.9 if no errors occurred

By the time the compiler tries to resolve labels, all syntax errors have been found. If label resolution is successful (which it must be unless the program CALLed a subroutine that doesn't exist or has mismatched control structure blocks), then no further errors can occur, and the p-code is a complete and valid command block.

4.1.10 write p-code out to the output file

The p-code in the array maintained by ticlCompiler.c can now be written to the .tcmd file.

4.1.11 write compilation statistics summary to listing

Per the compiler spec, run-time of the compiler along with source lines compiled and bytes of code produced need to go into the listing.

4.1.12 close all files

As noted earlier, the source files take care of themselves, but now is the time to close the listing and object streams. If errors *did* occur, the object stream file is removed and the listing is left behind for informational purposes.

At this point, the entire compilation process is complete, and an exit code of SUCCESS is returned to the operating system.

4.2 Compilation Details

When the compiler is in the STORED or IMMEDIATE State, commands are compiled using a data-driven approach. Each keyword in the language is stored in a static CommandTable (a keyword/function dictionary), so a quick scan through the table identifies the function that can handle that particular command. Many (about half) of the commands can be handled by the same function. These are referred to as "simple" commands. The rest require individual handling.

Simple commands are stored in a table with the keyword name, flight software command number, and an array of characters that describe each parameter required by the command. These commands are compiled by first emitting the command number to the p-code stream, then building the p-code representation for the parameters.

Compilation of multiple parameters is complicated by the fact that knowledge of the type and size of both parameters needs to be known before the first byte (the parameter descriptor) can be written. This forces us to compile the parameters into a secondary buffer (declared in the compileParameters function), and on completion, copy that buffer out into the actual p-code buffer.

4.3 Compiling Control Structures:

The flight software system supports an assembly language like set of control structures consisting of a COMPARE instruction and a set of conditional JUMP instructions. The TICL compiler supports Pascal-like control structures IF/ELSE/END_IF, WHILE/END_WHILE, and REPEAT/UNTIL. These high-level control structures are converted into the low-level ones supported by the flight software as describe in the following sections.

4.3.1 IF/ELSE/END_IF

This control structure can take two forms; one with an ELSE clause (shown in

Figure 3), and one without (Figure 2).

Figure 2
Conversion of IF/END_IF

Source Language	Compiles into
IF <op1> <comparison-op> <op2> <command-block> **END_IF** <whatever-else>	**COMPARE** <op1> <op2> **JUMP**_<comparison-op> After_if <command-block> After_if: <whatever-else>

Figure 3
Conversion of IF/ELSE/END_IF

Source Language	Compiles into
IF <op1> <comparison-op> <op2> <command-block-1> **ELSE** <command-block-2> **END_IF** <whatever-else>	**COMPARE** <op1> <op2> **JUMP**_<comparison-op> After_if <command-block-1> **JUMP** After_Else After_if: <command-block-2> After_Else: <whatever-else>

Notice that the IF part compiles into exactly the same thing in both cases. The conditional gets turned into its logical opposite (ironically) by leaving the parameters in the same order in the COMPARE command. The jump to After_if is left as an unresolved label until either an ELSE or END_IF is found.

If an ELSE is encountered, then an unconditional JUMP to After_Else is compiled in, and then the After_if label is resolved to be the resulting p-code offset.

The behavior of END_IF depends on whether or not an ELSE was found. If not, then the After_if label is resolved to be the current address. If an ELSE was found, then the After_Else label is resolved instead.

4.3.2 WHILE/END_WHILE

Figure 4 shows how a WHILE/END_WHILE loop is converted into the flight software's control structures.

Figure 4
Conversion of WHILE/END_WHILE

Source Language	Compiles into

WHILE <op1> <comparison-op> <op2>
  <command-block>
END_WHILE
<whatever-else>
	Top_of_While:
	  COMPARE <op1> <op2>
	  JUMP_<comparison-op> After_While
	  <command-block>
	  JUMP Top_of_While
	After_While:
	  <whatever-else>

Within a WHILE loop, the BREAK command compiles into an unconditional jump to After_While, and CONTINUE compiles into an unconditional jump to Top_of_While.

4.3.3 REPEAT/UNTIL

The test-at-the-bottom loop is converted as shown in Figure 5.

Figure 5
Conversion of REPEAT/UNTIL

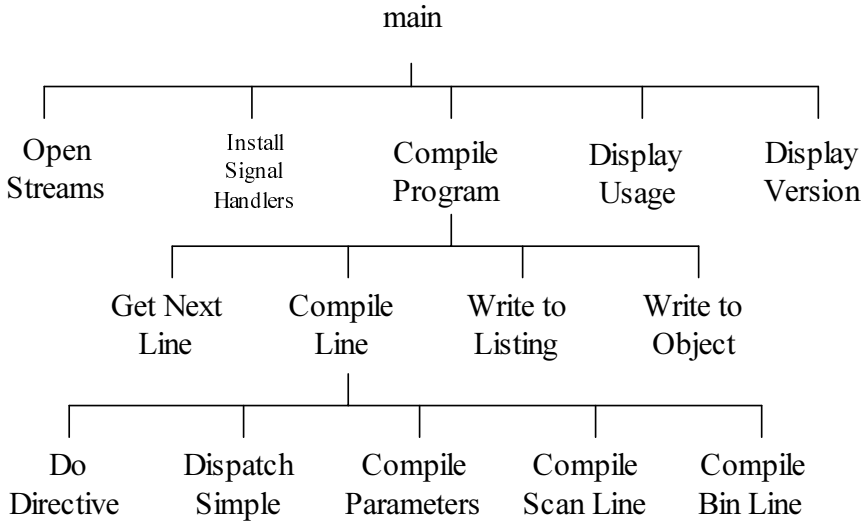
Source Language	Compiles into

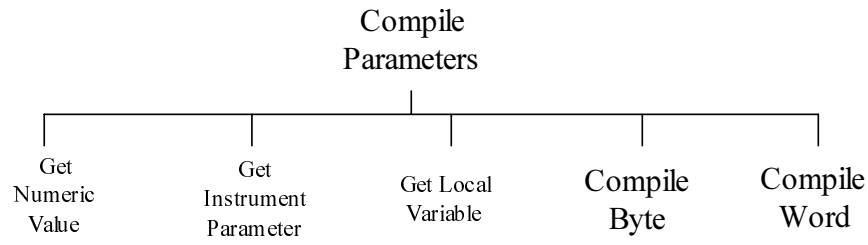
REPEAT
  <command-block>
UNTIL <op1> <comparison-op> <op2>
<whatever-else>
	Repeat:
	  <command-block>
	Until:
	  COMPARE <op1> <op2>
	  JUMP_<comparison-op> Repeat
	After_Until:
	  <whatever-else>

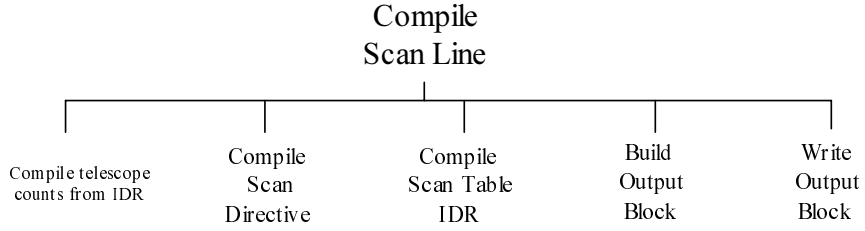
Within the scope of a REPEAT loop, BREAK compiles into an unconditional jump to After_Until and CONTINUE into an unconditional jump to Until.

4.4 Functional Block diagrams

Here are some high-level functional decomposition trees for the compiler, which should provide an idea about which functions call which other functions.







4.5 Scan Table Compilation

Compilation of scan tables is done by changing the state of the compiler. Once the state has been changed, the language accepted by the compiler is radically different. Compilation of scan tables follows the following algorithm:

```
Reset Scan Table Machinery
While not at end of table
    Read a line
    If the line is a directive
    Perform it.
    Else
    Compile as an Interval Definition Record (IDR)
    End If
End While
```

Scan table directives usually just store a value for later use during IDR compilation. The one real exception is the .BIN directive, which is shorthand for loading a whole binning table while in the process of compiling a scan table. Each of the directives is handled by its own function in ticlScanTable.c, and is straightforward.

Compiling a scan table IDR is more complicated than it should be, because the flight software wants to minimize the number of bytes that are uploaded to perform a particular scan. That means that a lot of extra complication is built into the compiler. Here is how compiling an IDR works:

```
Read an IDR
If the IDR is complete
    Build the Output Record
    Write the Output Record
Else
    If this partial IDR completes the set
    Build the Output Record
    Write the Output Record
    Else
    Keep the information provided
    End If
End If
```

Because there is not a one-to-one mapping between input records and output records for scan tables, we have to use some fairly complicated logic to build up the output record with between one and four input records. Each input record describes the motion for one, two, or four telescopes. We need the motion for all four telescopes before we can build the output record. The function compileTelescopeCountsFromIDR() does this.

Building the output record consists of saving off the state of the instrument and breaking the telescope motion up into up to 5 pieces. The first (optional) output record is a positioning record with a large erase time and an exposure time of zero. This record is generated automatically by the compiler to allow the instrument mechanisms a chance to get into position for the next scan. If the mechanisms are already in position, then this record is not generated. This logic is contained in the function buildOutputRecord().

The (up to four) records following that are the scan lines that move the telescopes along their course. The first line moves the telescope with the shortest (in terms of number of steps) scan through its entire scan, while moving the other telescopes partially through their motion. Each line after that moves the remaining telescopes, with each subsequent line moving one or more of the telescopes to its final position. This logic is also contained in the function buildOutputRecord().

Finally, when an output record has been built, it can be compiled into the staging buffer. The writeOutputRecord() function does this.

4.6 Binning Table Compilation

Compiling a binning table is very similar to compiling a scan table. The pseudo code is identical, except that binning tables have no machinery to reset.

```
While not at end of table
    Read a line
    If the line is a directive
    Perform it.
    Else
    Compile as an Interval Definition Record (IDR)
    End If
End While
```

Because there is a one-to-one mapping between binning table input records and output records, no convoluted logic is required to compile them. Still, binning tables are compiled into a holding buffer and then spooled out to the "main" p-code stream when end-of-table is reached. The logic for doing this spooling is identical to that for scan tables.

5. Maintenance Activities

This section describes how to perform common maintenance activities on the compiler. For in-depth troubleshooting, see the Troubleshooting Guide.

5.1 Adding new values to the configuration file

Add the appropriate static variable declaration to ticlConfig.c. Declare the corresponding accessor function in ticlConfig.h, and define it in ticlConfig.c. Add the fscanf statement to read it in readConfigurationFile in ticlConfig.c.

5.2 Adding a new simple command to the language

Add the new command's code to the list in ticlCodes.h. Add the new command, along with the encoding of its parameters and its code to the table of simple commands in compileCommand() in ticlCommands.c

5.3 Adding a new complex command to the language

Add the new commands code to the list in ticlCodes.h. Write a handler for the new command in ticlCommands.c Declare that function to be static at the top of ticlCommands.c. Add the command and the name of the handler to the list of complex command handlers in compileCommand() in ticlCommands.c.

5.4 Adding a new instrument parameter

Add the new parameter to the spreadsheet. Export the spreadsheet as tab-delimited text and install it as directed in the installation document (usually placed in /usr/local/data). No code changes are required.

5.5 Adding new columns to the instrument parameter spreadsheet

Add the new columns to the spreadsheet. Export the spreadsheet as tab-delimited text and install it as directed in the installation document (usually placed in /usr/local/data). Add the new symbols to the COLs enumeration at the top of ticlParameters.c. If the compiler needs to use these new columns, then add new fields to the ParameterInfo structure in ticlParameters.h, and add code to read them in the readLineOfParameterData function in ticlParameters.c.

Appendix A, Design Notes

Q: Why not use a ready-made scanner/parser?

A: TICL is a very simple, line-oriented language. As such, I chose not to use a third-party lexical scanner (like lex) or parser (like yacc). In fact, the bulk of the scanning and parsing of the TICL programs is contained in a couple of functions in ticlParse.c, where strtok is used to break the lines into their constituent tokens.

Another factor that decided me against using a traditional compiler compiler is that TICL is mode based, i.e. depending on the keywords that have been compiled, the language accepted by the parser changes. TICL is actually four languages: The immediate-mode TICL, the stored-program TICL, the Scan Table language, and the Binning Table language. The TICL compiler needs to be able to parse and translate all four of these languages. It is very hard to put several scanners and parsers into the same executable using flex/bison and impossible using lex/yacc. Simpler (in this case) just to do it by hand.