

# ATLAS COMPILER DEVELOPMENT

**Kenneth C. Chisolm**  
**Software Engineering Division**  
**(TISAD)**  
**Ogden Air Logistics Center**  
**Hill Air Force Base, Utah 84056**  
**801-775-4445**  
**chisolmk@software.hill.af.mil**

**James C. Lisonbee**  
**Software Engineering Division**  
**(TISAD)**  
**Ogden Air Logistics Center**  
**Hill Air Force Base, Utah 84056**  
**801-775-4442**  
**Lisonbej@software.hill.af.mil**

*Abstract - This paper will present methods used in design and development of an ATLAS to C compiler. ATLAS has two major differences from traditional computer languages. ATLAS has an inherent object based organization, many ATLAS language elements are designed around the support of hardware. The other major difference is that programs are frequently written with test station oriented languages embedded within ATLAS code.*

## INTRODUCTION

Many Automatic Test Equipment (ATE) organizations need to replace aging hardware. A complete replacement of hardware frequently involves change in the computer operating system and programming languages. Organizations with a large number of programs under active management will be required to find an automated means of performing program conversion, or to spend a lot of money converting them by hand [9]. The F-16 Analog Test Station Sustainment (FATSS) group developed a compiler to convert legacy ATLAS programs to a form of the C language supported by a replacement test station. The legacy programs also have a Test Station Native Language (TSNL) embedded within them. Most of this paper will discuss how an ATLAS to C language translator was implemented using the traditional compiler tools LEX (LEXical analyzer) and YACC (Yet Another Compiler Compiler). It is assumed that the reader is somewhat familiar with LEX and YACC, at least one version of ATLAS, and has already come to the conclusion they need to construct an ATLAS translator or compiler.

Compiler development can be a useful tool for organizations that are involved in hardware upgrade, and have a large amount of software to support. Organizations that do not own source code for their

compiler may have to write a new one from scratch. Others may be required to modify the output portions of an existing compiler to produce language for use with a new CPU or computer.

## LEXICAL ANALYSIS

The first step in compiler design is to convert source code into a sequence of tokens. Tokens are defined as character strings terminated by separators. Definitions of tokens and separators are subject to change on the fly. The minus sign is a primary example of this sort of ambiguity. In a math statement it always terminates a preceding token, and is a token in its own right. Elsewhere the minus sign may be another character embedded within a token (FALL-TIME). Fortunately we have a tool named LEX that in combination with reference material [1], [5], and [6] provide a great deal of assistance in resolution of these problems.

## COMMON Lexical Design

### *Regular Expressions*

Almost all lexical analyzers will need to recognize some very common concepts like integers, real numbers, white space, and frequently integers expressed in binary, hex, or octal. This is so common that LEX and many text editors provide a concept called a regular expression. Regular expressions allow an integer to be stated as "[0-9]+", where any ASCII character between '0' and '9' will be accepted. The '+' indicates one or more of these characters in a row constitute a single entity, in this case an integer. LEX provides means to specify these common text entities for reuse in construction of more complex tokens. The real number -1.2E-4.6 should provide a hint at how

complex specification of common tokens can be. All the compiler texts referenced in this paper are littered with examples of how to build sets of complex regular expressions.

Regular expression constructs are also useful in definition of ATLAS line numbers, variables, white space, branch definitions, entry point definitions, ATLAS strings, and relay definition formats.

Compiler designers should look for opportunities to build slightly larger concepts into regular expressions. One example is that you might want a new line definition to be a Carriage Return (CR) followed by white space. This will allow treating leading white space as the nuisance it is (recognize it and throw it out). This is an extension of what is done with most white space. The first white space or any separator is important, but remaining white space is just extra useless characters.

It is useful to recognize comments with a regular expression construct. Extra caution may be required with handling of comments. They may run across more than one line, or may be improperly terminated. The handling of these problems will be covered in parsing and context switching sections.

### Token Recognition

The goal of a lexical analyzer is to recognize tokens and return them to the parser, or throw them away. Comments by definition produce no output code, and if not for the problems mentioned above could be thrown away. About the only token the F-16 Analog Test Set (FATS) ATLAS translator throws away is the comma separator followed by white space. One crucial feature of LEX is that it will recognize the longest token that fits any character string received. It will receive characters until one fails to fit the set of tokens defined in its source file. This rule is necessary to be able to recognize numbers. If LEX did not look for the longest fit a ten-digit integer would be returned as ten, one digit integers. When complexities like real numbers are considered LEX would quickly be rendered useless without this longest fit recognition rule.

Most of the time the parser worked best when the string encountered by LEX was returned along with its token definition. If a real number rather than an ASCII string representing a real number were returned to the parser, the parser would be required to make a decision on output format independent of the real number input format. This could easily lead to significant round off errors. If a string is passed then the original legacy code programmers decision on

numerical format is maintained. The return of two items is accomplished through the return of a token #defined to represent an integer (named `TOKEN_real`), and filling a global union with the ASCII string representing the token. Any place within the parser that receives a `TOKEN_real` knows the value of the real number can be located as a string in the global data union shared between LEX and YACC code. Other tokens could return integer strings, variable strings, or no additional information, Table 1.

Table 1: ATLAS tokens

Token type	Example
Integer	1234
Real	.12E+3.4
Terminator	\$
Variable	'ABCD'

## ATLAS Lexical Differences

### Reserved words

The primary difference between ATLAS and many other computer languages is that ATLAS appears to have a large number of reserved words. Many languages have a limited number of reserved words. C uses `for`, `else`, `while`, `static`, `float`, etc. as reserved words. The C language probably does not include more than a few dozen reserved words. ATLAS requires that variable strings be surrounded by single quote marks 'VAR'. This rule allows ATLAS terms like `VOLTAGE`, `MEASURE`, `(RES)`, and others to be treated as reserved words. They can be assumed to be distinct concepts inherent to the language itself. When LEX recognizes a reserved word it need only return the definition of the token. The string required to define the token is explicitly defined by the use of the reserved word token.

As a general rule any ATLAS token explicitly specifies the use of an associated hardware concept. For example any time an ATLAS program uses the token `MEASURE` it can be interpreted as a command to perform a measurement. The syntax parser will require `MEASURE` to be followed by a variety of other token types. Each reserved word token will have its own integer defined for passing to the parser. ATLAS reserved words can be grouped into functional categories based on how they are used within the ATLAS language. Careful assignment of reserved word integer values will allow additional useful

information to be derived from the reserved word integer. The FATS hardware drivers are able to re-use the reserved word integers internally, and implement rudimentary type checking based on reserved word category information.

Table 2: ATLAS reserved words

Token Type	Examples
Verb	APPLY, MEASUER, CALCULATE, PRINT
Noun	AC SIGNAL, DC SIGNAL, IMPEDANCE,
Measured Characteristic	(VOLTAGE), (FREQ), (RISE-TIME), (RES)
Functions	SIN, COS, ABS
Pin descriptors	HI, LO
Other Mnemonics	VOLTAGE, FREQ, MAX, UL, LL,
Dimension	V, KV, OHM, A, MA, UA

### Context switching

Token recognition rules can change from time to time. One of the primary places this occurs is following a "PRINT, MESSAGE,". Everything following a "PRINT, MESSAGE," has to be a comma delimited label, or string. Additional analysis shows this rule holds for anything following the term "MESSAGE," LEX provides the means to switch context upon encountering the token "MESSAGE,". After a context switch occurs only terminators and comma delimited labels or strings are recognized as valid tokens.

The FATS translator also switched context upon encountering a CNX token. CNX fields consisted of pin descriptors (HI, LO, A, B, C, etc), integers, ATLAS labels, and everything else was returned as a connect field.

Comments required switching to a context similar to but slightly different from messages. Comments accepted everything up to a CR and returned the string as a comment. CRs were accepted and ignored thus allowing multi-line comments. The legacy ATLAS compiler did not require comments to have terminators. The legacy compiler would produce good output code if an ATLAS new line followed a comment. ATLAS new lines are defined as any of the following: Branch definition, entry point definition, comment definition, or an atlas line number. LEX emulated this behavior, but generated a warning message for the ATLAS programmer.

The master segment, lookup segment, and general information segment all had their own context handling requirements. The FATSS translator threw out all master segment software. This was done because all master segment tasks were embedded within the test station front end.

LEAVE ATLAS statements forced context switching to search for TSNL tokens. RESUME ATLAS statements returned LEX to an ATLAS context mode. The TSNL had its own secondary context switching requirements.

The FATSS group found it convenient to switch to a default context when ATLAS terminators were encountered. This allowed recognition of line numbers, branches, entry points, comments and segment headers in one location. It also provided a means to tell the difference between an ATLAS line number and an integer at the start of a continued ATLAS line.

### PARSING.

Most compiler design reference material, including the sources cited by this article [1] [5] [6], give examples on development of C or Pascal language compilers. The literature indicates parsers must reduce a sequence of tokens to basic concepts that can easily be converted into output statements. Translation from ATLAS to another high level language does not require that parsed concepts be reduced to such a low level. All ATLAS lines of code can be thought of as requiring a simple syntax conversion from ATLAS to C. This is grossly oversimplified, but remains a viable top level viewpoint for an ATLAS translator. As with most simplistic viewpoints this one requires large amounts of refinement to obtain a viable product.

### Hardware interface verbs.

First and foremost we must return to the fact that ATLAS is a computer language designed to facilitate the programming of a sequence of hardware actions. ATLAS reserves several words (verbs) to define the actions a programmer might implement in hardware. Many programmed actions are associated with actual instrumentation. This association occurs through the paired use of ATLAS DEFINE and RENAME statements, or by default. The legacy ATLAS code use of a STOP WHEN verb always results in the programming of Pulse Threshold Detector (PTD) B to detect a programmed point on a signal. In design of a compiler it is helpful to recognize this relationship between verbs and instrumentation hardware.

Each ATLAS statement can be parsed to identify which legacy hardware was used with the command. Once the legacy instrument is identified the compiler developer can produce a line of C code to call the instrument driver (public function) [8] of the replacement instrument.

The design of the compiler will be far easier if the instrument interface requirements have been carefully specified. All instrument interfaces should not only have the same look and feel, but they should be as nearly identical as is practical. The closer they are to identical the more a compiler design team and instrument design personnel can rely on the use of common subroutines to provide instrument interface.

The FATSS project developed a complete set of instrument interface requirements before any code was written [8]. The requirements were developed from a careful review of the capabilities of the legacy system. The end result was a set of instrument interface requirements that allowed all replacement instrumentation to operate properly when translated from legacy language. This at times may mean passing optional parameters to an instrument and designing the instrument to program a default value when a parameter is not present.

One place optional parameters are almost sure to be required is in the use of DC offset voltages. The FATS legacy system had function generators that allowed the use of DC offset voltages and some that did not. All of the replacement hardware could implement DC offset voltages. The easiest solution was to pass a DC offset voltage value or if one is not available pass a null to all function generators. When a public function received a null in place of a DC offset voltage it was required to program a zero volt DC offset. The end result is a set of instrument drivers and interface requirements that are as replaceable as the hardware itself.

Observation of the legacy ATLAS yielded a generic format for instrument usage. The formats consisted of a verb, optional measured characteristics, a noun, followed by zero or more parameters, and finally a connect fragment. The preceding sentence is a YACC parse rule for a line of ATLAS code. The parse rule only needs to be converted to YACC syntax. The ATLAS parameters connect fragments, and other elements all remain to be defined. Any given verb, measured characteristic, and noun combination define instrument usage. The FATS ATLAS compiler required development of separate subroutines to convert parameters to conform to instrument interface requirements.

One other type of statement includes a verb, followed by a defined instrument label with an associated instrument, and additional command components. This constitutes yet another ATLAS line parsing rule. The additional command components must be merged into the ATLAS string associated with the defined instrument label. Most DEFINE statements allow additional command components to consist of anything from as small as a single dimension on up to a large set of parameters. These differences can exist in the additional parameters from one use of a defined label to the next.

The FATSS group took an indirect approach to handling this type of line. We found it difficult to identify how to directly parse a line of code fragmented this badly. We merged the defined command string and the additional command components. Then this new command string was merged with the associated instrument name (for parsing ease and readability) to produce a new line of ATLAS code. The new line of code always started with an instrument name, followed by the tokens required by a simple instrument parse rule. The previous sentence can be construed as yet another parse rule template defined for ATLAS commands.

This is another place where it is critical to use good tools. Performing this sort of in line construction could be very difficult in a build it yourself compiler. LEX provides useful tools to manage this problem. After a line of code has been merged into a new string the new string must be written to a temporary file. Then the compiler developer can use built in LEX commands to switch the source code input stream to accept code from the temporary file. The FATSS group spent a couple of days creating new versions of LEX commands to switch the input stream to the new command string without writing it to a file. This investment in development time eliminated the execution time required to create files, write to them, and finally destroy them. The preceding steps would have been necessary every time a program used a DEFINED ATLAS label (probably half to two thirds of the lines of code in a program do this).

### ***Basic instrument operation***

Parsing connect fragments requires much more explanation. A review of the legacy ATLAS manual indicated the selection of an ATLAS verb has implications on hardware use. There is a distinct hierarchical relationship between ATLAS verbs. The lowest level ATLAS verbs result in a call to exactly one instrument driver. This instrument driver might open or close one or more relays, it could setup activity on a

sensor or source instrument, or it could command a sensor to read a value. These verbs are referred to as simple instrument verbs in Table 3.

The next level of ATLAS verbs, referred to as complex instrument verbs, require the use of more than one simple instrument verb to complete a hardware operation. Extended instrument verbs are complex verbs that require additional processing after instrument access is complete. The final class of verbs are used are called extended verbs, they require multiple verbs in order to complete one hardware action.

Table 3: ATLAS verb organization

Verbs	Handling style
Simple instruments	OPEN, CLOSE, CONNECT, DISCONNECT, SETUP, READ, DELAY
Complex Instruments	APPLY, REMOVE, MEASURE
Extended Instruments (extra processing)	MONITOR, VERIFY, WAIT FOR, DO DIGITAL TEST
Multiple Verb sets	{ADJUST, TO REACH, TO MAXIMIZE, TO MINIMIZE} {START WHEN, STOP WHEN, with MEASURE (TIME)}, SYNC WHEN with
Simple conversion	END, LEAVE, RESUME, PERFORM, SAVE, RECORD, DISPLAY, INDICATE CALCULATE, PRINT, COMPARE, GO TO
Convert and manage variables	DEFINE, DECLARE, FILL, DISCARD, RENAME
Minimal usage	BEGIN, FOR UUT, SPECIFY, DO NETWORK ANALYSIS, DO SPECTRAL ANALYSIS, TERMINATE

### Relay parsing

Connect fragments are defined by an ATLAS CNX token and multiple connect fields. A connect field consists of a pin descriptor (HI, LO, A, B, C, N), followed by one or more pin names. In the LEX portions of this paper we discussed context switching. LEX switched context to define pins as any string that was not a pin descriptor or a terminator (\$).

### Parameter Parsing

Instrument parameter parsing is another concept that requires additional consideration. A parameter is an ATLAS reserved word, usually followed by a number and an optional dimension. As parameters were encountered each of their elements, the parameter token, number string, and a real value for the dimension were all placed in a structure. This allowed each parameter to be stored for future reference by instrument handler subroutines.

Table 4: Example parameter requirements

Token	Additional requires
VOLTAGE, CURRENT, FREQ, PERIOD	Real numbers and optional dimensions
VALUE, BNR, BCD	Integer numbers and optional dimensions
PSLOPE, NSLOPE, INCREASING, DECREASING	None

### Math

The symbol table section of this paper explained the difficulties involved in variable name retrieval. The sequence order of the arithmetic for TSNL is from left to right, instead of from inner to outer brackets as with most of the mathematics. If not for these two problems the FATSS group could have written a stupid math string converter. Instead the FATSS group was required to use traditional recursive descent math parsing.

### Non instrument verbs.

All non-instrument verbs require standard parsing that can be copied from most compiler design reference material.

## ATOM DEVELOPMENT

### Hardware interface verbs

After the parser recognized an instrument it called an instrument handler subroutine. The instrument handler function takes parsed parameters and converts them into a common format at compile time and produces a public function call. The instrument handler subroutine was also required to call pre and post processing subroutines. The pre and post processing subroutines were responsible for making calls to relay driver public

functions. The public function is a runtime executable command that makes a call to actual instrument hardware using a common interface. The FATSS driver paper [8] refers to other functions within the public function, but the whole point to the driver design process is that the public function call is the only aspect of hardware interface compiled C code need know or care about.

### ***Simple instrument operation***

Production of code for simple instrument verbs was a byproduct of calls to instrument handlers. The instrument handler was designed to discover and handle any of the following hardware requirements for all verbs. Simple verbs permitted only one of the following items to be performed.

#### ***Throw relays***

When the FATS compiler encountered a pin in a CNX fragment it called a subroutine that first looked the pin up in the ATLAS relay lookup table. This table was created during the first pass through the compiler. The subroutine then used the relay information to create a relay driver public function call. If the pin was not found an error was produced. The FATS used multiple hardware devices to throw relays. The FATS wanted to throw relays in the same order as the legacy station, but as fast as possible. TPS execution speed was increased by limiting the number of function calls requiring VXI hardware access. As sequences of relays were encountered that all used the same hardware they were buffered to wait for the end of the ATLAS statement or a relay requiring a different instrument was commanded. When one of these conditions was met a relay driver public function call was produced. This method produced lines of C code as ATLAS relays were identified. The C language calls to relay drivers were buffered and output by the instrument handler subroutines.

#### ***Throw relay to instrument***

The legacy ATLAS software was responsible for generating commands to throw relays to connect power from source instruments to the rest of the test station. This was accomplished by passing the instrument name to the instrument's pre and post processing routines. These routines used the instrument name to generate an appropriate public function call.

### ***Command instrument activity***

The only ATLAS command the legacy station made to source instrument drivers was to setup the instrument and produce an output. Some instruments were required to support TSNL commands that utilized multiple commands to setup the instrument and produce an output. The legacy station supported separate commands for sensor instrument setup and reading results. All instrument access commands resulted in the generation of a public function call.

### ***Complex instrument operation***

Complex verbs utilized the same instrument handler function as simple verbs did. The only difference is that they required production code for more than one embedded simple verb.

### ***Extended instrument operation activity***

There are several ATLAS instrument verbs that conform to the instrument definitions above and require processing in addition to public function calls. One of the easiest to recognize is VERIFY. It is as difficult to implement VERIFY as any of the other extended verb.

VERIFY is a command that performs a comparison on results of a measurement made by a public function. This is handled by repetition of a theme. Within each instrument handler we call pre or post processing subroutines that handle production of non-public function calls required for instrument interface. When comparisons are required, the pre processing routines search the parsed parameters, locate comparison adverbs, and then produce runtime code required to set upper and lower limits. The pre and post processing routines are responsible for generating code required by all extended verbs. The subroutines handle production of variable setup, generation of loops, and passing information between instruments.

Numerical comparisons required careful handling in order to conform to legacy station operational requirements. Often the TSNL set up test limits, then at some indeterminate number of commands later compared the limits with a value. The easiest way to get the FATS to operate under these requirements was to copy the legacy system software architecture. Once this was done for low level code it was just as easy, and far safer to reuse TSNL comparison methods for ATLAS code. An attempt to produce a traditional C language post processing comparison statement would require that we guard against the possibility of

mixed language comparisons. This would have doubled (at least) the required lines of C code produced, and increased the probability of producing code with logical errors.

### **Multiple ATLAS verbs**

The final instrumentation design problem was production of C language code that coordinates operation of multiple instruments. Legacy ATLAS software presented two impediments to the normal method of outputting C code when each instruments was encountered. Some ATLAS commands employed non-continuous multi verb sets. SYNC WHEN statements are not always adjacent to commands that use it.

The other problem is that ATLAS often expects multiple verbs to perform all instrument setup functions, close all relays, and then start instrument operation. This requires inter-weaving of executable code among several ATLAS commands.

Code production for both of these problems were solved through buffering C language public function calls. For example when a START WHEN command arrived, the public function call to setup the instrument went to the C output file. The public function call to throw relays could not occur until after the STOP WHEN command was setup. This meant the START WHEN relay driver calls had to be buffered and output later. When a STOP WHEN function arrived it was required to output its own instrument setup command, then output the buffered START WHEN relay calls, all prior to producing its own relay driver calls. All verbs used in multiple verb commands had distinct requirements for buffering and retrieving public function calls. The START and STOP WHEN example always works the way described above.

Using YACC to DIRECTLY recognize missing statements would have been very difficult. A simple yet effective approach was to test command buffers before use. If a buffer was unexpectedly empty an appropriate error message was produced. This would produce an error when a START WHEN command was missing. These storage buffers were also tested prior to filling and at the bottom of each segment to assure all buffered commands were properly consumed. This was designed to produce errors when part of a multi-verb command was programmed. As an example if a STOP WHEN was missing from source code the next START WHEN or the bottom of the segment would find information from the previous START WHEN command abandoned in the buffer and produce an error.

The ATLAS ADJUST and TO REACH are a combination of multiple and extended verbs provide compiler developers with a very delightful coding exercise. While more caution is required here no additional design concepts are needed.

### **Symbol Tables**

There are many complexities embedded in the naming convention of ATLAS and TSNL variables. The ATLAS and TSNL code use different variable names to represent the same address space. Legacy ATLAS code contains several additional complexities associated with variable names that required careful handling. These problems combine to make it difficult to employ a straight text based name conversion for variables. The FATS compiler was required to link ATLAS and TSNL names together in order to produce viable C language code. A symbol table was developed to provide an efficient means of variable management. We were able to retrieve the same translated variable name using the TSNL or ATLAS strings as the input token.

The symbol table can be viewed as a set of compiler subroutines that allow the storage of all legacy source code variables. The symbol table also provides a means to retrieve the C language string for any given legacy language variable name.

The symbol table also provides fields to associate defined data types with symbol usage.

Most legacy test language statements use a variable in one way or another. Symbol table routines must be used whether the variable is being stored or retrieved by the legacy code. Several non-traditional variables are also passed into symbol tables.

The legacy languages include a relay lookup table feature that enables mnemonics to be associated with relay pin definitions and digital test unit pin numbers.

Symbol table access is also useful for storage and retrieval of line numbers that are associated with ATLAS branch and go to statements. Symbol tables aid in assuring that all jump commands have a defined address to jump to.

Variables are often defined in a file-input stream after they are first used. This is quite common in the use of jump addressing. The FATSS group used a multiple pass compilation method to assure all translated variable names were available when needed.

## **First pass**

The first pass of the translation identifies variable declarations and stores them in symbol tables for future use.

Variable mnemonics are placed in a symbol table on the first pass. Variable mnemonics are retrieved from the symbol table as needed. Several other items are stored in the symbol table along with the variable mnemonics. The legacy ATLAS name, TSNL variable name, data type, array size, and the translated C language name must all be stored in the lookup table. ATLAS procedure and instrument definition statements must store additional information in the symbol table.

ATLAS RENAME statements will append instrument names to instrument definitions stored in the symbol table.

The legacy ATLAS variable declaration statements create global variables that are placed in a global symbol table. Any variable that appears to the left of a math statement equal sign and isn't found in the global symbol table can be placed in a local symbol table. Each segment of legacy source code has its own unique local symbol table. This is the way local variables were recognized and stored for later use.

## **Second pass**

The second pass of the compiler extracts all needed information from the symbol tables.

Variables retrieved from the symbol table return the translated name of the variable. Often this name requires additional modification. It may need type casting, digital masking or addition of array subscripts. There were several additional subroutines written to modify variable names as needed.

Most ATLAS instrument commands require the use of a defined instrument template. Lines of ATLAS code that use the template may contain additional instrument command elements that must be merged into defined locations within the instrument template. Symbol table software included subroutines to perform this task.

Retrieval of procedure calls also performs tasks to type cast passed parameters. It is important with this version of legacy ATLAS to carefully manage the format variables are passed in. Some variables are passed by value and others are passed by reference.

The data storage format indicators are radically different between ATLAS and C.

All relay field mnemonics must use their own symbol table access routines to reconstruct lookup table mnemonics and retrieve all associated pin definitions. Then these pin definitions can be passed to subroutines used to generate relay public function calls.

## **Math**

Traditional compilers output parsed math commands through a sequence of calls to math atoms. The atoms contain math operands and variables required to produce low level assembly code. Assembly code may require additional pop and push stack operations to properly execute math commands. The concepts required to convert ATLAS math code to C language are somewhat different from standard compiler design. Standard compilers are able to produce simple assembly language calls for each math fragment that is encountered by a recursive descent parser. The nature of this parser design guarantees that proper execution order is maintained. High-level language translators must convert math atoms to strings, and pass those strings back up to the next higher level of the parser. This means where a traditional parser can dump a math fragment when it is found, an ATLAS translator must go through a lot of careful work to return math strings back out of a heavily recursive call stack.

The compiler replaced all ATLAS math function calls with standard ANSI function calls prototyped in "math.h" header file.

Some care was required in handling of digital versus analog math. The following examples demonstrate results of ATLAS to C conversion of digital math.

ATLAS statement:

```
CALCULATE, 'FI WORD' = 'DIA MAP'(22) OR  
O'100000' $
```

C statement:

```
global_FIWORD = global_DIAMAP[22] | 0x8000;
```

ATLAS statement:

```
CALCULATE, 'DI' = SHIFT 'DI' LEFT 3 BITS $
```

C statement:

```
global_DI = global_DI << 3;
```



## System level verbs.

The design team looked for LEAVE and RESUME ATLAS statements and performed context switching when they were encountered.

## Standard software translation

Most of the remaining source code commands required simple syntax conversion. PRINT, INDICATE, and DISPLAY are the closest thing the remaining ATLAS verbs presented to a difficult translation problem. The verbs themselves imply the output file descriptor (PRINT - line printer, and CRT / stdout) required by C code. ATLAS and TSNL source code have far simpler print usage than C. The compiler design team was required to concatenate print strings, replace any variables with %f, or %d, and place the translated variable string at the end of the C language print statement.

## REFERENCES

- [1] J.R. Levine, T. Mason, D. Brown. *lex & yacc*. Sacramento: O'Reilly & Associates, Inc., 1991
- [2] W. Street, Lex/YACC (actually FLEX and BISON), [http://www.monmouth.com/user\\_pages/wstreett/lex-yacc/lex-yacc.html](http://www.monmouth.com/user_pages/wstreett/lex-yacc/lex-yacc.html)
- [3] B.W. Kernighan, D.M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall Inc., 1978
- [4] H. Schildt, *C The complete Reference*. Berkeley: Osborne McGraw-Hill, 1987
- [5] S. Bergman, *Compiler Design Theory, Tools, and Examples*. Dubuque, Iowa: Wm. C. Brown Publishers, 1994
- [6] J. Holms, *Building Your Own Compiler With C++*. Englewood Cliffs, NJ: Prentice-Hall Inc., 1995
- [7] J. Lisonbee, K. Chisolm and M. Sithivong, *Concurrent Engineering for Automatic Test Station Development*, 1999 IEEE.
- [8] L. Vu and A. Khan, *Instrument Driver Design*, 1999 IEEE
- [9] K. Chisolm and J. Lisonbee, *ATLAS Compiler Development*, 1999 IEEE