FINAL REPORT

AUTOTHROTTLE TEST PROCEDURE COMPILER

By

J E. Doss
E.S. Acree
C.A. Batterman
S.N. Cole
J.B. Thompson

Software Research Division
Computer Science and Technology Laboratory
Engineering Experiment Station
Georgia Institute of Technology
August 20, 1979

Prepared for

Marconi Avionics, Inc.
4500 N. Shallowford Road
Atlanta, Georgia 30338
Under Contract 40125
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>i</td>
</tr>
<tr>
<td>Summary</td>
<td>ii</td>
</tr>
<tr>
<td>1.0 Introduction</td>
<td></td>
</tr>
<tr>
<td>1.0.1 A Sample Program</td>
<td>2</td>
</tr>
<tr>
<td>1.1 Test Procedures in the ATP Language</td>
<td>4</td>
</tr>
<tr>
<td>2.0 The ATP Language</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Symbols in the ATP Language</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1 Key Words</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1.1 Aliases</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2 Pin Numbers</td>
<td>9</td>
</tr>
<tr>
<td>2.1.3 Values and Tolerances</td>
<td>10</td>
</tr>
<tr>
<td>2.1.4 Character Strings</td>
<td>11</td>
</tr>
<tr>
<td>2.1.5 Comments</td>
<td>12</td>
</tr>
<tr>
<td>2.2 ATP Statements</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1 Preliminary Statements</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2 Adjust Statements</td>
<td>16</td>
</tr>
<tr>
<td>2.2.3 Set Statements</td>
<td>18</td>
</tr>
<tr>
<td>2.2.4 Connect Statements</td>
<td>18</td>
</tr>
<tr>
<td>2.2.5 Disconnect and Remove Statements</td>
<td>20</td>
</tr>
<tr>
<td>2.2.6 Voltage Measure Statements</td>
<td>21</td>
</tr>
<tr>
<td>2.2.7 READ-YES and READ-NO Statements</td>
<td>22</td>
</tr>
<tr>
<td>2.2.8 READ-NUM Statement</td>
<td>23</td>
</tr>
<tr>
<td>2.2.9 Print Statement</td>
<td>23</td>
</tr>
<tr>
<td>2.2.10 Save Statement</td>
<td>24</td>
</tr>
<tr>
<td>2.2.11 Wait Statement</td>
<td>25</td>
</tr>
<tr>
<td>SECTION</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2.2.12 Inline P-Code</td>
<td>25</td>
</tr>
<tr>
<td>2.2.13 Transition Statement</td>
<td>26</td>
</tr>
<tr>
<td>2.2.14 Monitor Statement</td>
<td>27</td>
</tr>
<tr>
<td>3.0 Definition of P-Code</td>
<td>28</td>
</tr>
<tr>
<td>3.1 P-machine Implementation</td>
<td>29</td>
</tr>
<tr>
<td>3.2 Number Representation</td>
<td>32</td>
</tr>
<tr>
<td>3.3 P-Code Formats</td>
<td>32</td>
</tr>
<tr>
<td>3.4 Branch Instructions</td>
<td>33</td>
</tr>
<tr>
<td>3.5 Standard Procedures</td>
<td>34</td>
</tr>
<tr>
<td>3.6 P-Codes</td>
<td>36</td>
</tr>
<tr>
<td>3.6.1 NOP</td>
<td>38</td>
</tr>
<tr>
<td>3.6.2 OUTSTR</td>
<td>38</td>
</tr>
<tr>
<td>3.6.3 GETSTR</td>
<td>39</td>
</tr>
<tr>
<td>3.6.4 CMPSTR</td>
<td>39</td>
</tr>
<tr>
<td>3.6.5 OUTDEC</td>
<td>39</td>
</tr>
<tr>
<td>3.6.6 GETDEC</td>
<td>39</td>
</tr>
<tr>
<td>3.6.7 SETCLK</td>
<td>40</td>
</tr>
<tr>
<td>3.6.8 GETCLK</td>
<td>40</td>
</tr>
<tr>
<td>3.6.9 BRA</td>
<td>40</td>
</tr>
<tr>
<td>3.6.10 BLT</td>
<td>40</td>
</tr>
<tr>
<td>3.6.11 BGT</td>
<td>40</td>
</tr>
<tr>
<td>3.6.12 BLE</td>
<td>41</td>
</tr>
<tr>
<td>3.6.13 BGE</td>
<td>41</td>
</tr>
<tr>
<td>3.6.14 BEQ</td>
<td>41</td>
</tr>
<tr>
<td>SECTION</td>
<td>PAGE</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>3.6.15 BNE</td>
<td>41</td>
</tr>
<tr>
<td>3.6.16 BM</td>
<td>41</td>
</tr>
<tr>
<td>3.6.17 BP</td>
<td>41</td>
</tr>
<tr>
<td>3.6.18 BZ</td>
<td>42</td>
</tr>
<tr>
<td>3.6.19 BNZ</td>
<td>42</td>
</tr>
<tr>
<td>3.6.20 ABS</td>
<td>42</td>
</tr>
<tr>
<td>3.6.21 NEG</td>
<td>42</td>
</tr>
<tr>
<td>3.6.22 CLA</td>
<td>42</td>
</tr>
<tr>
<td>3.6.23 STOP</td>
<td>42</td>
</tr>
<tr>
<td>3.6.24 CMP RX</td>
<td>43</td>
</tr>
<tr>
<td>3.6.25 CMP #</td>
<td>43</td>
</tr>
<tr>
<td>3.6.26 STA Rx</td>
<td>43</td>
</tr>
<tr>
<td>3.6.27 LDA Rx</td>
<td>43</td>
</tr>
<tr>
<td>3.6.28 LDA #</td>
<td>43</td>
</tr>
<tr>
<td>3.6.29 SUB Rx</td>
<td>43</td>
</tr>
<tr>
<td>3.6.30 SUB X</td>
<td>44</td>
</tr>
<tr>
<td>3.6.31 ADD Rx</td>
<td>44</td>
</tr>
<tr>
<td>3.6.32 ADD #</td>
<td>44</td>
</tr>
<tr>
<td>3.6.33 MULT Rx</td>
<td>44</td>
</tr>
<tr>
<td>3.6.34 MULT #</td>
<td>45</td>
</tr>
<tr>
<td>3.6.35 DIV Rx</td>
<td>45</td>
</tr>
<tr>
<td>3.6.36 DIV #</td>
<td>45</td>
</tr>
<tr>
<td>3.6.37 WAIT</td>
<td>45</td>
</tr>
<tr>
<td>3.6.38 OPEN</td>
<td>45</td>
</tr>
<tr>
<td>3.6.39 CLOSE</td>
<td>46</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (CONT'D)

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.2.6 Format of &quot;VOLT&quot; Segment Records</td>
<td>59</td>
</tr>
<tr>
<td>4.1.2.7 Format of &quot;PIN_&quot; Segment Records</td>
<td>59</td>
</tr>
<tr>
<td>4.2 Hardware Definition File - Converted Form</td>
<td>62</td>
</tr>
<tr>
<td>4.2.1 Common Block File (x.D3)</td>
<td>62</td>
</tr>
<tr>
<td>4.2.2 Connection File (x.D1)</td>
<td>62</td>
</tr>
<tr>
<td>4.2.2.1 Format of Connection File</td>
<td>63</td>
</tr>
<tr>
<td>4.2.3 Pin Id File (x.D2)</td>
<td>66</td>
</tr>
<tr>
<td>5.0 Operating Procedures</td>
<td>67</td>
</tr>
<tr>
<td>5.1 Compiler Usage</td>
<td>68</td>
</tr>
<tr>
<td>5.2 Assembler Usage</td>
<td>70</td>
</tr>
<tr>
<td>5.3 Hardware Definition Conversion Process</td>
<td>71</td>
</tr>
<tr>
<td>Appendices</td>
<td></td>
</tr>
<tr>
<td>1 BNF for the ATP Language</td>
<td>73</td>
</tr>
<tr>
<td>2 Generated P-Code</td>
<td>79</td>
</tr>
<tr>
<td>3 Error Messages</td>
<td>85</td>
</tr>
<tr>
<td>4 P-Code Object File Format</td>
<td>93</td>
</tr>
</tbody>
</table>
PREFACE

This report was prepared at the Georgia Tech Engineering Experiment Station under Contract 40125. The work covered by this report was performed in the Computer Science and Technology Laboratory under the supervision of Mrs. Edith W. Martin. The progress reported herein was performed during a 4-month program to develop a compiler for automating the testing of analog flight computers.

This project has been monitored by Brian Wood and Gene Lowe of Marconi Avionics, and their helpful guidance is gratefully acknowledged. Appendix 4 was supplied by W.A. Ritch of Marconi Avionics.
SUMMARY

The objective of this study was to develop a high-level test generation language and compiler for the automated testing of Marconi's Boeing 747 Autothrottle computer. This language allows Marconi's engineers to produce quality control check-out procedures for the Autothrottle computer in an easy to understand format.

The project has been performed in two phases: the analysis phase and the implementation phase. The analysis phase consisted of defining (1) the syntax and semantics of an English-like user-oriented test generation language, (2) a low-level pseudo-code (P-code) to be produced from the test generation language and (3) the architecture of an intermediate virtual computer which would execute the P-code. The virtual computer concept simplifies system implementation, since its definition could be tailored specifically to this type of application (i.e. to automated testing).

The test generation language, developed in the analysis phase, is a regular or type 3 language which is input to the compiler in a user-oriented English style format. The implementation phase consisted of writing the compiler which would convert the test generation language into P-code. An interpreter for the P-code generated by the compiler is being developed concurrently by Marconi Avionics.

The process of automated testing of the Autothrottle computer consists of the following steps. First the electrical checkout procedures are written in the test generation language (about 5000 lines of commands). The procedures are then compiled into P-code which is down loaded to the microcomputer within the test set hardware. The microcomputer interprets its P-code instructions
and applies sequences of analog signals to the autothrottle computer. For each sequence of signals applied, the microcomputer checks for correct responses from the Autothrottle computer's outputs.

The automated check-out of the Autothrottle computers will permit a substantial decrease in the amount of time required to test each unit. Previously each Autothrottle computer had to be tested manually requiring about a man-month of a technician's time.
1.0 INTRODUCTION

The Acceptance Test Procedure (ATP) language is an English-like language developed for writing automatic test procedures. Statements are written in an easy to use, self explanatory structure similar in format to a technicians test manual. Test statements are converted into an automatic test sequence using the ATP compiler. The primary design goal of the ATP compiler was to automate the execution of test specifications which have previously been written for manual testing of Marconi autothrottle computers. The language structure of the original test procedures has generally been retained. However, this structure has been limited and constrained to standardized formats and wording. Standardization helps simplify ATP programming by avoiding an overly complex language.

The ATP compiler converts the ATP language statements into mnemonic pseudo-code (P-code). This P-code is then translated by the ATP assembler into object P-code which is, in turn, converted by an interpreter on a TI 9900 microprocessor into instructions for the Autothrottle Semi-Automatic Test Equipment. Thus, the test specification procedures generated in the ATP language will ultimately control the test performed by the test equipment.
1.0.1 A SAMPLE PROGRAM

BEGIN 1.1

BEGIN 1.1.1

CONNECT J101B-6 TO +28.000V
THE VOLTAGE AT J101-B122 SHALL BE:
MEASURE +22.000V ± 2.000V

END

BEGIN 1.1.2

PRINT "MOVE BIASED SWITCH TO +28.0V";
PRINT "INPUT VOLTAGE AT J101A-60";
READ-NUM 28.000V ± 0.100V

END

BEGIN 1.1.3

MONITOR A TRANSITION AT J101B-122
FROM 9.500V ± 1.000V to 28.000V ± 2.000V
AT TIME 0.600S ± 0.400S

OPEN CIRCUIT J101B-120 (GA INIT)

$ START-P

LDA R4
MULT R5
STA R6

$ STOP-P

PRINT "R4 * R5 =" R6

END

END
Listed above is a sample ATP test program. Notice that the program is broken into blocks which contain various numbers of statements. The outer block of the test sequence is called section 1.1. The inner blocks numbered from 1.1.1 to 1.1.3 are called subsections. Within each subsection are statements which accomplish various objectives. Each statement is written in a specific format or syntax. All of the valid ATP statements are described in Section 2.0 of this Manual.

Note that subsection 1.1.1 of the sample program contains the statement 'CONNECT J101B-6 TO +28.000V'. In order to generate an automated test procedure to complete this statement, a number of conditions must be satisfied. First, the statement must be verified to be certain it is syntactically correct. The ATP compiler verifies the syntax using the syntactic rules described formally in Appendix 1. After verifying that a statement's syntax is correct, the compiler interprets the meaning (semantics) of the statement. The ATP hardware definition tables described in section 4.0 are checked to see if the desired statement can actually be completed. For the example statement to be valid, it must be possible to connect either a 28-volt logic discrete or a DAC set to 28.0 volts to the test pin J101B-6.

After having determined that a statement can actually be completed, the ATP compiler must produce instructions for the actual test computer. In the example the states of switches must be set, and perhaps a DAC value must be adjusted.
Instructions to the test computer are written using a special P-code. The ATP P-code defines a virtual computer tailored for automatic testing. Section 3.0 describes the P-code instruction set. It is possible to use P-code directly within the automatic test sequence when the complete flexibility of the P-machine is required. Section 1.1.3 of the sample test program illustrates the use of inline P-code.

After P-code has been generated by the ATP compiler, it is assembled into a compact binary form by the ATP assembler. Section 5.2 describes the use of the ATP assembler. After assembly, P-code may finally be down-loaded into the test computer for execution.

1.1 TEST PROCEDURES IN THE ATP LANGUAGE

The ATP language is basically a free format language. The free format aspect of the language allows the user to structure the test specification procedures in a natural way, without the restrictions imposed by an inflexible fixed format.

The following guidelines involving statement format must be observed:

1) A high-level statement in the ATP language may be placed on more than one line of the text of a test specification procedure. However, there are some restrictions placed on character strings and comments which should be noted in sections 1.2.3 and 1.2.5.

2) Each line of test specification text may contain up to 132 characters.

3) Tab characters, blanks, and commas are all considered to be delimiters (separators).
4) Four of the basic components of the language (key-words, pin-numbers, numeric values, and character strings) must be separated from each other by one or more delimiters, the only exception is that numeric values and their unit tokens (VOLTS, SECONDS, etc.) do not have to be separated by delimiters.

To specify tests using the ATP language a test procedure must be written following the standard formulation. A test procedure is composed of one or more sections. A section consists of a BEGIN verb, a section number, a section name, zero or more subsections and is concluded with an END statement.

For example:

```
BEGIN 5.1 THIS IS A SECTION NAME

SUBSECTION 0
```

```
.
```

```
.
```

```
.
```

```
SUBSECTION N
```

```
END
```
A subsection consists of a BEGIN verb, a subsection number, a series of zero or more statements and is concluded by an END statement. For example:

BEGIN 5.1.1 STATEMENT 0
.
.
.
.

STATEMENT N

END

The following items should be noted about sections, subsections, BEGIN's, and END's:

1) The section name is limited to 40 characters in length. The character count for the name begins with the first nonblank character to the right of the section number, and the character count ends with either the last nonblank character of the name or the fortieth character of the name, whichever comes first.

2) Subsections do not typically have names attached to them. However, if it is desired to name a subsection then this may be accomplished by putting the name in parenthesis (as a comment).

3) Statements of the language are found in the subsections and not in the sections.

4) The first statement of a subsection may appear either on the same line as the begin statement appears (as in 5.1.1), or it may appear on the following line.

5) Each section and each subsection must be concluded with an END statement.
2.0 THE ATP LANGUAGE

2.1 SYMBOLS OF THE ATP LANGUAGE

The ATP language allows the user to specify such operations as CONNECT, DISCONNECT, ADJUST, and MONITOR in a high-level language. Additional operations which are not available in the high-level language may be specified with in-line P-code statements.

The high-level statements of the ATP language consist of five basic components:

1) Key words - such as ADJUST, BETWEEN, TO, VOLTAGE, and PRINT.

2) Pin-numbers - consisting of mnemonic identifiers of the pin connections on the equipment being tested such as J101A-1, J103-9, etc.

3) Numeric values - consisting of positive and negative integers and real numbers such as +16, +16.0, 0.005, .5 and so on.

4) Character strings - which are found in the print statement and consist of alphanumeric strings of eighty characters or less enclosed with single or double quote marks.

5) Comments - consisting of any descriptive information the user would like to have in the text of the ATP source file. Comments are enclosed in parenthesis. Any information enclosed in parenthesis will not be processed by the compiler and is simply "stripped" from the source file.
2.1.1. **KEY WORDS**

The key words of the ATP language consist of approximately 88 words commonly used in the ATP language high-level statements. These words must be spelled correctly or they will not be identified by the ATP compiler. Some words have aliases which allow variations on the standard form of the word used in the syntax of the ATP language. The following is a complete list of keywords:

1) $START-P 45) PERIOD
2) $STOP-P 46) POINTS
3) ; 47) POS
4) A 48) POSITIVE
5) ADJUST 49) PRINT
6) AND 50) RO
7) AT 51) R1
8) BE 52) R2
9) BE: 53) R3
10) BEGIN 54) R4
11) BETWEEN 55) R5
12) BY 56) R6
13) CHANGE 57) R7
14) CHANGES 58) R8
15) CHECK 59) R9
16) CIRCUIT 60) READ-NO
17) CLOCKWISE 61) READ-NUM
18) CONNECT 62) READ-YES
19) CONNECTION 63) REMOVE
20) COUNTERCLOCKWISE 64) S
21) D 65) S1
22) DEG 66) S2
23) DEGREE 67) S3
24) DEGREES 68) SAVE
25) DIRECTION 69) SEC
26) DISCONNECT 70) SECOND
27) END 71) SECONDS
28) FOLLOWING 72) SET
29) FOR 73) SHALL
30) FROM 74) SIGNAL
31) HZ 75) SYNCHRO
32) IN 76) THE
33) IS: 77) TIME
34) MAX 78) TO
35) MEASURE 80) TRANSITION
37) MIN 81) TRANSMITTER
38) MONITOR 82) UNTIL
39) NEG 83) V
40) NEGATIVE 84) VOLT
41) OF 85) VOLTAGE
42) OF: 86) VOLTS
43) OPEN 87) WAIT
44) PER 88) WITHIN
2.1.1.1 ALIASES

There are some key words that are treated as aliases of each other. For example, D, DEG, DEGREE, and DEGREES are all aliases of each other and any one of them may be used in place of the other. The following is a list of allowable aliases for the ATP language:

1) BE, BE:
2) CHANGE, CHANGES
3) D, DEG, DEGREE, DEGREES
4) IS, IS:
5) NEG, NEGATIVE
6) OF, OF:
7) POS, POSITIVE
8) S, SEC, SECOND, SECONDS
9) TO, TO:
10) V, VOLT, VOLTS

2.1.2 PIN NUMBERS

Pin-numbers consist of mnemonic identifiers of the different pin connections on the autothrottle equipment being tested. The mnemonic identifier consists of from one to ten characters in length. Each pin must be properly defined in the hardware definition file so that all the legal connections for a pin are specified. The following examples illustrate some typical pin numbers:

J101A-57
J103-9
J101B-55
J102-22
J101B-124
2.1.3 VALUES AND TOLERANCES

Two different types of numbers occur in the ATP language: Values and tolerances. Values may be real or integer, signed or unsigned, and may or may not be followed by a unit token such as VOLTS, DEGREES, SECONDS, or HZ. One form of the ADJUST statement allows the user to specify a step size which should not be followed by a unit token.

Tolerances, on the other hand, are immediately preceded by the symbols +- (plus minus) and are always followed by a unit token. Tolerances are found only after a value with a unit token. The unit token of the tolerance must match the unit token of the value that it follows. The unit token of the tolerance is always checked to ensure that it is of the same type as the unit token of the preceding value.

Numeric values and tolerances may range between the values -32767 and +32767. If a number is found that is out of this range an appropriate error message will be printed. Up to three decimal digits to the right of the decimal point will be processed; any additional digits will be ignored and an error message printed. The following values and tolerances are examples of legal value/tolerance usage:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>TOLERANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.000</td>
<td>0.005V</td>
</tr>
<tr>
<td>16.000V</td>
<td>0.050V</td>
</tr>
<tr>
<td>16 SEC</td>
<td>1 SEC</td>
</tr>
<tr>
<td>16 SECONDS</td>
<td>1 SEC</td>
</tr>
<tr>
<td>16 SECONDS</td>
<td></td>
</tr>
</tbody>
</table>
The following values and tolerances are examples of illegal value/tolerance usage:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>TOLERANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+-3.125</td>
<td>(NO VALUE AND NO UNITS)</td>
</tr>
<tr>
<td>16.000 S</td>
<td>+-1.0V</td>
</tr>
<tr>
<td>1.0V</td>
<td>0.005V</td>
</tr>
<tr>
<td>+45000 V</td>
<td>+-100.0 V</td>
</tr>
<tr>
<td>6,000 SEC</td>
<td>+-1,000 SEC</td>
</tr>
</tbody>
</table>

2.1.4 CHARACTER STRINGS

Character strings occur in the PRINT statement. Anything enclosed in single or double quote marks is considered a part of the character string and will be passed without further processing to the output buffer. Character strings are limited to eighty characters in length, including the quote marks.

Because of the maximum length limitations on the character string, character strings should not extend over one line; otherwise, blank characters may be improperly inserted into the character string. The extra blanks will be counted as part of a character string's eighty character limit and therefore may cause an error in processing. The following examples of the PRINT statement illustrate the correct usage of the character string:

```
PRINT "THIS CHARACTER STRING WOULD BE ENTIRELY TOO"
"LONG SO IT HAS BEEN SPLIT INTO TWO STRINGS"
PRINT 'SINGLE OR DOUBLE QUOTE MARKS MAY BE USED'
```
The following examples illustrate some illegal character strings:

PRINT "THIS CHARACTER STRING IS ENTIRELY TOO LONG AND
SHOULD HAVE BEEN SPLIT UP TO AVOID TAKING UP TOO
MUCH SPACE"
PRINT "THIS STRING HAS NO CLOSING QUOTE MARK
PRINT THIS STRING HAS NO OPENING QUOTE MARK"

2.1.5 COMMENTS

A comment convention has been adopted for the ATP language to allow explanatory information to be inserted into the text of a test specification procedure. Comments are literally "stripped" from the test procedure at compile time and they do not cause any code to be generated.

Comments are enclosed in parenthesis which indicate to the ATP compiler that the information enclosed in the parenthesis is not to be processed any further. This implies that parentheses cannot be included within the comment itself. The one exception to the comment convention involves character strings. Any information enclosed inside parenthesis in a character string will be output with the rest of the character string.

A comment may be up to twenty lines in length. If a comment is not concluded within twenty lines an error message will be printed. The twenty line limitation is enforced to avoid having the remainder of a test procedure treated as a comment by the ATP compiler in the event that the closing parenthesis is missing. If it is desired to have longer comments, simply close the first part of the comment when the line limit has been reached and begin on the next line with a "new" comment containing the second portion of the original comment and so on. The following examples illustrate correct
usage of the comment convention:

(THIS IS A SIMPLE ONE LINE COMMENT)

(THIS IS A MULTI-LINE COMMENT WHICH MAY EXTEND OVER AS MANY AS TWENTY LINES OF TEXT, WITH UP TO 132 CHARACTERS PER LINE).

(BLANK LINES MAY ALSO BE INSERTED INTO A COMMENT LIKE THIS ONE BUT, THE BLANK LINE WILL BE COUNTED ALONG WITH THE OTHER LINES IN THE COMMENT TOWARD THE TWENTY LINE MAXIMUM)

The following examples demonstrate some incorrect uses of the comment convention:

(THIS COMMENT HAS NO CLOSING PARENTHESES, WHICH WILL CAUSE THE COMPILER TO SCAN UP TO TWENTY LINES OF TEXT LOOKING FOR THE CLOSING RIGHT PARENTHESES INDICATING THE END OF THE COMMENT.

THIS COMMENT HAS NO OPENING PARENTHESES WHICH WILL CAUSE THE COMPILER TO ATTEMPT TO TREAT THIS COMMENT AS THOUGH IT WAS A HIGH-LEVEL ATP STATEMENT. THE CLOSING RIGHT PARENTHESES WILL BE IGNORED AND MAY CAUSE A SYNTAX ERROR.)
(COMMENT LINE 1: IF A COMMENT EXTENDS OVER MORE THAN
COMMENT LINE 2: TWENTY LINES, THEN THE COMMENT IS TOO
COMMENT LINE 3: LONG AND WILL GENERATE AN ERROR MESSAGE
COMMENT LINE 4: TO THAT EFFECT.

.

.

.

COMMENT LINE 20: THIS COMMENT SHOULD END ON THIS LINE, BUT
COMMENT LINE 21: SINCE IT DOES NOT IT WILL GENERATE AN ERROR)
2.2. ATP STATEMENTS

2.2.1 PRELIMINARY STATEMENTS

Two statements are required as the first and second statements of an ATP source file. The first statement is of the form

\[
\text{TEST SPECIFICATION NUMBER } = \text{string}
\]

where \text{string} is a string of characters with a maximum length of 10. This statement gives the test specification number which is to be copied to the output object file.

The second statement is of the form

\[
\text{HARDWARE DEFINITION FILE } = \text{filename}
\]

where \text{filename} is the name of the hardware definition file to be used by the compiler. The filename should end with a '.DF' extension (e.g. HW747.DF would be a valid name) and have at most 14 characters.
2.2.2 ADJUST STATEMENTS

The adjust statements may be used to step the output of a device (associated with <PIN 1>), starting at its current value, until a measurement (associated with <PIN 2>) reaches a desired value or until the value of the output reaches a limit value. The specified output (from a DC-DAC or D/C) is incremented by the given step size once after the desired step time has elapsed. A measurement is then made from the A/D associated with <PIN 2>. If the voltage is within the desired range ($V_{\text{low}}$ to $V_{\text{high}}$) the statement successfully terminates. The process of stepping the output and measuring the A/D voltage repeats until the desired value is measured or the output reaches a limit value. If the limit value occurs, an error message and appropriate status values are printed on the user's console.
The following two statements are used to adjust DC-DAC values:

ADJUST SIGNAL AT <PIN 1) BY <STEP SIZE>
PER <TIME> SECONDS IN A <SIGN>
DIRECTION UNTIL THE VOLTAGE AT
<PIN 2> CHANGES TO _V +- _V

ADJUST SIGNAL AT <PIN 1> BY
<STEP SIZE> PER <TIME> SECONDS UNTIL
THE VOLTAGE AT <PIN 2> IS _V +- _V

The following type of statement may be used to adjust the synchro transmitter until the voltage at a pin reaches the desired value:

ADJUST SYNCHRO TRANSMITTER BY
<STEP SIZE> PER <TIME> SECONDS
UNTIL THE VOLTAGE AT <PIN #>
IS _V +- _V
2.2.3 SET STATEMENT

The set synchro statement is utilized to set the synchro transmitter to a specific angle. The set statement appears within the ATP test sequence as follows:

SET THE SYNCHRO TRANSMITTER TO ___ DEGREES +/- ___ DEGREES

<DIRECTION>

<DIRECTION> can be either omitted (defaults to 'CLOCKWISE'); or 'CLOCKWISE' or 'COUNTERCLOCKWISE' can be substituted for <DIRECTION>.

2.2.4 CONNECT STATEMENTS

The connect statements are used to connect object devices to automatic test equipment pins. A connection object may be another pin, a D/S source, or a DAC set to a specified voltage.

To connect a test pin to another test pin use the statement:

CONNECT <PIN 1> TO <PIN 2>

To connect a test pin to a DC voltage use one of the following statements. Note that the ATP compiler first checks if a logic discrete can be connected. If not, a DC-DAC is used.

CONNECT <PIN 1> TO ___V
CONNECT <PIN 1> TO A SIGNAL SET TO ___V
To apply an AC voltage to a test pin use the following ATP statement:

\[
\text{CONNECT } _V, _HZ \text{ }<\text{PHASE}> \text{ TO } <\text{PIN 1}>
\]

Either omit <PHASE> (defaults to 0 degrees), or substitute '
, _ DEGREES' in place of <PHASE>.

To connect a digital to synchro source to a test pin use the statement:

\[
\text{CONNECT } <\text{SYNCHRO PIN}> \text{ TO } <\text{PIN 1}>
\]

Substitute either 'S1', 'S2', or 'S3' for <SYNCHRO PIN> in the statement.
2.2.5 **DISCONNECT AND REMOVE STATEMENTS**

The group of disconnect and remove statements may be utilized to remove connections between test objects and pins. All statements in this group function similarly. The first pin number in a statement is used to locate an open circuit entry in the hardware definition table. Any switches associated with the open circuit entry are placed in the specified state required to attain an open circuit status.

Following are the various forms of the disconnect and remove statements:

- `DISCONNECT <PIN 1> FROM __V`
- `DISCONNECT <PIN 1> FROM <PIN 2>`
- `DISCONNECT <PIN 1> FROM SYNCHRO TRANSMITTER`
- `DISCONNECT SIGNAL FROM <PIN 1>`
- `REMOVE SIGNAL BETWEEN <PIN 1> AND <PIN 2>`
- `REMOVE CONNECTION BETWEEN <PIN 1> AND <PIN 2>`
- `REMOVE __V __HZ [__DEGREES] BETWEEN <PIN 1> AND <PIN 2>`
- `REMOVE SIGNAL FROM <PIN 1>`
- `REMOVE __V FROM <PIN 1>`
- `REMOVE <PIN 1> FROM <PIN 1>`
- `REMOVE <PIN 1> FROM SIGNAL`
- `REMOVE <PIN 1> FROM __V`
2.2.6 **VOLTAGE MEASURE STATEMENTS**

Voltage measure statements are used to insure that a voltage measured at a test pin is within tolerance. The following two statements both check that the voltage at \(<\text{PIN 1}>\) is within the range: \(\_V \pm \_V\)

**VOLTAGE AT THE FOLLOWING POINTS SHALL BE:**

\(<\text{PIN 1}>\) MEASURE \(_V \pm \_V\)

**VOLTAGE BETWEEN <PIN 1> AND <PIN 2> SHALL BE:** MEASURE \(_V \pm \_V\)

When a large number of voltages must be checked, the following statements may be used:

**VOLTAGE AT THE FOLLOWING POINTS SHALL BE:**

\(<\text{PIN 1}>\) MEASURE \(_V \pm \_V\)

\(<\text{PIN 2}>\) MEASURE \(_V \pm \_V\)

\(\ldots\)

\(\ldots\)

\(<\text{PIN N}>\) MEASURE \(_V \pm \_V\)

There are two alternative forms of the voltage measure statement. When it is necessary to wait a period of time before a voltage measurement is made, the following statement can be used:

**VOLTAGE AT <PIN 1> SHALL BE:**

MEASURE \(_V \pm \_V\) WITHIN A PERIOD OF: CHECK \(_S \text{ MAX}\)
Sometimes requirements provide that a voltage must remain within tolerance during some time interval. The following, which is equivalent to a monitor statement, provides a means to insure that a measured voltage is within tolerance over a selected time interval.

THE VOLTAGE AT <PIN 1> SHALL BE: MEASURE \_V \_V FOR A PERIOD OF: CHECK _S MIN

2.2.7 READ-YES AND READ-NO STATEMENT

At certain points within the ATP test sequence it may be appropriate to solicit YES or NO responses from the test operator. If the operator answers with an appropriate response the test should continue normally, otherwise an abnormal termination is required. The READ-YES and READ-NO statements both request YES or NO answer from the operator. In the case of the READ-YES construct, the program continues normally if the operator answers with a YES response. For the READ-NO statement program execution continues normally provided a NO response is received.

These operator interaction statements appear within the ATP test sequence as follows:

READ-YES
READ-NO
2.2.8 **READ-NUM STATEMENT**

A few measurements required within automatic test sequences must be performed manually by the test operator. The READ-NUM statement provides a simplified method of retrieving analog data from an operator and then checking the response to insure that it is within a desired value. When the READ-NUM statement is executed, operator response is solicited. The value of the response is checked for the range: \( \_V \pm \_V \). If the value is within range, the test sequence continues. Otherwise, operator response is requested a second time. If the retrieved analog data is again not within range the test sub-section terminates. The READ-NUM statement syntax is:

\[
\text{READ-NUM } \_V \pm \_V
\]

2.2.9 **PRINT STATEMENT**

Print statements allow the ATP programmer to send messages to an operators console during ATP program execution. Either strings or virtual machine registers may be output using the print statement. The syntax of the print statement is:

\[
\text{PRINT } \text{ITEM} 1, \text{ITEM} 2, \ldots, \text{ITEM} \, n
\]

where \( n \) (number of items) is arbitrary, and \( \text{ITEM} \, j \) is either a string of the form

"\text{<1 OR MORE CHARACTERS>}" 

or a register reference of the form RO...R15.

Character strings and register references may be listed in any desired order within the print statement. Semi-colons can be used within strings to effect a carriage-return-line feed operation.
2.2.10 SAVE STATEMENT

Certain test sequences require that an intermediate result be temporarily saved for use latter. For example, the voltage change statement may be used several statements after a reference point for the change value has been established. It is necessary to save the reference value until it is needed by the CHANGE statement. The P-machine registers R0 through R9 may be used to save intermediate data. Resultant values from MEASURE, ADJUST, or MONITOR statements may be saved with the value contained in the register in P-machine registers using the save statement. A voltage change statement may then be used later to check the changed value with the value contained in the P-machine register.

For example:

(CHECK VOLTAGE AT PIN J101A-58)

VOLTAGE AT J101A-58 SHALL BE:

MEASURE 15.0V ± 0.01V

(SAVE RESULT OF MEASUREMENT)

SAVE IN R4

.

.

.

(ANY STATEMENTS WHICH DO NOT REFERENCE R4)

.

.

.

(CHECK FOR 4.0 V CHANGE)

VOLTAGE CHANGE AT J101A-58 FROM

R4 SHALL BE: MEASURE 4.0V ± 0.10V

24
2.2.11 **WAIT STATEMENT**

A wait statement may be used whenever a delay in the automatic test sequence is required. The effect of a wait statement is to delay the test sequence for the specified period of time.

The WAIT statement is used as follows:

```
WAIT ___ SECONDS
```

2.2.12 **INLINE P-CODE**

The output of the ATP compiler is P-code for a virtual machine. The syntax of the P-code is defined in section 3.0 of this document. At times it may be necessary to write P-code for the virtual machine directly within the ATP test sequence. Certain special test cases may be difficult or impossible to implement using the standard ATP syntax. In these cases the user may utilize the full flexibility of the P-code machine by writing instructions inline. P-code instructions must be enclosed between the start statement `$START-P` and the ending statement `$STOP-P`. All intervening instructions are copied directly to the p-code output file for assembly by the P-code assembler. The user must obey the syntax of the P-code assembler when he uses in-line P-code.
For example:

  .
  . (COMPILER STATEMENTS)
  .

$START-P

INPUT % 0A092 G1 (THIS IS A COMMENT)
LDA R0
CONV G1 C3
SUB R2
STA R4
GETDEC
STA R5
OUTSTR "THIS IS THE VALUE" R5
$STOP-P

  .
  . (COMPILER STATEMENTS)
  .

2.2.13 TRANSITION STATEMENT

Monitoring voltage waveform transitions may be accomplished using the transition statement. The statement insures that an initial voltage is maintained within tolerance until the time a transition is to occur (less time tolerance). The measured voltage must then change to the second voltage before the specified time (plus tolerance) has elapsed.
The syntax of the transition statement is:

\[
\text{MONITOR [A]} \ \text{TRANSITION AT} \ <\text{PIN \#}> \ \text{FROM} \ \_\_V \ \leftrightarrow \ \_\_V \\
\text{TO} \ \_\_V \ \leftrightarrow \ \_\_V \ \text{AT} \ \text{TIME} \ \_\_S \ \leftrightarrow \ \_\_S
\]

2.2.14 \ MONITOR STATEMENT

A monitor statement is used whenever a voltage must be guaranteed to remain within a certain tolerance for a minimum period of time. The monitor statement continually reads the A/D associated with a pin and checks that the voltage is within \_\_V \ \leftrightarrow \ \_\_V. When the specified time in seconds has elapsed the statement terminates.

The syntax of the monitor statement is:

\[
\text{MONITOR [THE]} \ \text{VOLTAGE AT} \ <\text{PIN \#}> \\
\text{OF} \ \_\_V \ \leftrightarrow \ \_\_V \ \text{FOR} \ \_\_S
\]
3.0 DEFINITION OF P-CODE

The process of converting English language test statements into an automated test sequence can be greatly simplified through the use of an intermediate P-code machine. Defining a virtual computer tailored specifically to a class of applications can simplify and speed system implementation. Using a P-code interpretive approach for automatic testing of autothrottle computers is likely to result in a flexible, general purpose test system.

The primary goal of the virtual P-machine is to achieve the flexibility to handle the various test sequences required for testing Marconi autothrottle computers. Test sequences may be categorized into instructions which occur frequently and a set of infrequent special test cases. Handling both classes of test instructions efficiently may be accomplished using two levels of P-code.

High Level P-code

High level P-code instructions were designed to handle the most frequent test cases using a minimum of P-code instruction space. Included in this category are the instructions to measure and check voltages, open and close switches, set a voltage, wait a period of time, and check a voltage transition. These categories occur frequently, hence it is desirable to have P-codes available which directly accomplish the required tasks. High level instructions could be coded using the low level P-code instruction set. However, for maximum speed and efficiency the high level instructions should be used where possible. Note that in this documentation many of the high level instruction sequences are actually specified in terms of low level
instructions. This approach is used only to add clarity. High level
instructions are actually hard-coded within the interpreter. The instructions
utilize many of the same utilities used to implement the low level
instructions.

Low Level P-code

In order to achieve the required test system flexibility it is
appropriate to specify a set of low level P-code instructions. The low level
instructions are used only to code special cases which cannot be handled by
the high level instructions.

Low level P-codes were chosen to provide a minimum set of codes which
can be used to implement most automatic test applications. Included are the
P-codes to accomplish simple looping control, comparisons, error processing,
console I/O and the various arithmetic functions. Since most test
applications do not require expression evaluation and subroutine linkages the
P-machine does not include stack oriented instructions. These facilities
could be added later if needed for future applications.

3.1 P-MACHINE IMPLEMENTATION

Marconi's automatic test P-machine will be implemented as a 32-bit
virtual computer. The actual implementation of the P-machine utilizes a
TI-9900 16-bit minicomputer. An interpretive program emulates the 32-bit
pseudo machine.
Data within the pseudo machine are, for the most part, represented as 32-bit quantities. A 32-bit accumulator (referred to as A) is used as the source and destination for the results of most instructions. Sixteen 32-bit general-purpose registers can be used for temporary data storage. A 16-bit program counter is used as a pointer for fetching instructions. The status register records the results of compare and arithmetic instructions. String registers maintain test section data and console input. See Figure 3.1.

The sixteen general registers are categorized into user and system registers. Registers 0 to 9 are reserved for the user as temporary storage registers. Registers 10 through 15 are used by the system to pass data between P-codes and have the following definition:

Register 10 - Available for system use only.
Register 11 - A/D Data converted to the real 32 bit format.
Register 12 - The correct value for the voltage at a pin.
Register 13 - The tolerance associated with the value in Register 12.
Register 14 - Initial value for change P-code.
Register 15 - Temporary for adjust device P-code.
Status Register (S)

EQ  LT  GT  Z  M

Program Counter (PC)

16 bits

Accumulator (A)

32 bits

General Purpose Registers (R0 - R15)

32 bits

String Input Buffer (SI)

80 bytes

Test Section Number (TS)

20 bytes

Figure 3.1 Pseudo Machine Registers
3.2 NUMBER REPRESENTATION

The automatic test P-code interpreter utilizes 32-bit floating point arithmetic for all calculations, thus providing all the accuracy needed by the ATP. The compiler produces 32-bit values, which correspond to the floating point format used by the TI 9900 microprocessor.

3.3 P-CODE FORMATS

Pseudo-instruction opcodes are specified in two categories. The majority are numbered sequentially from 01 hex to 2A hex. These opcodes may be implemented easily with a branch table. All opcodes which reference the P-machine registers have the high order bit of the opcode byte set. Register related instructions all have a decimal value greater than 127. Register reference pseudo instructions consist of two nibbles. The high order nibble specifies the operation to be performed, while the low order nibble contains a register reference. In all cases the accumulator is used for operand 1, and the register reference (0 . . F) hex is used for operand 2.

<table>
<thead>
<tr>
<th>Register reference instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nibble 1</td>
</tr>
<tr>
<td>8→E</td>
</tr>
<tr>
<td>Op code</td>
</tr>
</tbody>
</table>
3.4 BRANCH INSTRUCTIONS

The branch instructions allow the P-machine to transfer control to either a standard procedure or to another section of the P-code. The branch (BRA) instruction causes an unconditional transfer of program control. All other instructions in this category cause a change in the P-code execution sequence only if the appropriate bit in the status register is set (cleared). Each branch instruction includes a single byte opcode and a two byte offset. The two byte offset specifies where program control should transfer if the specified condition is true. If the high order bit of the offset is set, then the low order bits indicate which standard procedure to execute. When the high order bit is cleared then the low order bits contain an offset from the current PC. The offset may be positive or negative, it is represented in 2's complement form (in the low-order 15 bits).

The branch opcodes are represented:

```
8 bit opcode  x | y (15 bits)
```

If $x = 0$ then $y =$ offset in bytes from PC

If $x = 1$ then $y =$ standard procedure reference number
3.5 STANDARD PROCEDURES

SP1 - Record successful completion of test subsection contained in the test section register (TS). Begin execution of next subsection. Note that the entire subsection must be memory resident.

SP2 - Normal test termination. Ready equipment for next test. The BRA SP2 and STOP opcodes have identical effect.

SP3 - Abnormal subsection termination. ADC value out of tolerance. Print results contained in registers R10 - R13 and appropriate ID’s.

SP4 - Abnormal subsection termination. Unable to set DAC to proper value. Print results contained in registers R10 - R13 and appropriate ID’s.

SP5 - Abnormal subsection termination. Monitored voltage out of tolerance. Print elapsed time and results in registers R10 - R13 along with appropriate ID’s.

SP6 - Abnormal subsection termination. Voltage of a transition is out of tolerance. Print elapsed time, registers R10 - R13, and appropriate ID’S.

SP7 - Print normal results of a monitor instruction.

SP8 - Print normal results of an ADC measurement. Results contained in registers R10 - R13.
SP9 - Print normal results of a transition.

SP10 - Restart section.

SP11 - Abnormally terminate testing.

SP12 - Abnormal subsection termination. ADC value changed incorrectly. Print results contained in registers R10 - R14 along with appropriate ID's.

SP13 - Abnormal subsection termination. Unable to set synchro to correct value. Print desired value and tolerance from registers R12 and R13.

SP14 - Abnormal subsection termination. Operator measured value is out of tolerance.

SP15 - Operator responded incorrectly
The following P-codes have been implemented:

<table>
<thead>
<tr>
<th>OPCODE (HEXADECIMAL)</th>
<th>MNEMONIC</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>NOP</td>
<td>NO OPERATION</td>
</tr>
<tr>
<td>02</td>
<td>OUTSTR</td>
<td>OUTPUT ASCII STRING</td>
</tr>
<tr>
<td>03</td>
<td>GETSTR</td>
<td>INPUT ASCII STRING</td>
</tr>
<tr>
<td>04</td>
<td>CMPSTR</td>
<td>COMPARE ASCII STRING</td>
</tr>
<tr>
<td>05</td>
<td>OUTDEC</td>
<td>OUTPUT DECIMAL NUMBER</td>
</tr>
<tr>
<td>06</td>
<td>GETDEC</td>
<td>INPUT DECIMAL NUMBER</td>
</tr>
<tr>
<td>07</td>
<td>SETCLK</td>
<td>SET REAL TIME CLOCK</td>
</tr>
<tr>
<td>08</td>
<td>GETCLK</td>
<td>READ REAL TIME CLOCK</td>
</tr>
<tr>
<td>09</td>
<td>BRA</td>
<td>BRANCH UNCONDITIONALLY</td>
</tr>
<tr>
<td>0A</td>
<td>BLT</td>
<td>BRANCH IF LESS THAN</td>
</tr>
<tr>
<td>0B</td>
<td>BGT</td>
<td>BRANCH IF GREATER THAN</td>
</tr>
<tr>
<td>0C</td>
<td>BLE</td>
<td>BRANCH IF LESS THAN OR EQUAL</td>
</tr>
<tr>
<td>0D</td>
<td>BGE</td>
<td>BRANCH IF GREATER THAN OR EQUAL</td>
</tr>
<tr>
<td>0E</td>
<td>BEQ</td>
<td>BRANCH IF EQUAL</td>
</tr>
<tr>
<td>0F</td>
<td>BNE</td>
<td>BRANCH IF NOT EQUAL</td>
</tr>
<tr>
<td>10</td>
<td>BM</td>
<td>BRANCH IF MINUS</td>
</tr>
<tr>
<td>11</td>
<td>BP</td>
<td>BRANCH IF POSITIVE</td>
</tr>
<tr>
<td>12</td>
<td>BZ</td>
<td>BRANCH IF ZERO</td>
</tr>
<tr>
<td>13</td>
<td>BNZ</td>
<td>BRANCH IF NOT ZERO</td>
</tr>
<tr>
<td>14</td>
<td>ABS</td>
<td>TAKE ABSOLUTE VALUE OF ACCUMULATOR</td>
</tr>
<tr>
<td>15</td>
<td>NEG</td>
<td>NEGATE ACCUMULATOR</td>
</tr>
<tr>
<td>16</td>
<td>CLA</td>
<td>CLEAR ACCUMULATOR</td>
</tr>
<tr>
<td>17</td>
<td>STOP</td>
<td>NORMAL TERMINATION</td>
</tr>
<tr>
<td>8X NOTE 1</td>
<td>CMP RX</td>
<td>COMPARE ACCUMULATOR WITH REGISTER</td>
</tr>
<tr>
<td>18 NOTE 2</td>
<td>CMP #</td>
<td>COMPARE ACCUMULATOR WITH IMMEDIATE DATA</td>
</tr>
<tr>
<td>9X NOTE 1</td>
<td>STA RX</td>
<td>STORE ACCUMULATOR IN REGISTER</td>
</tr>
<tr>
<td>19 NOTE 2</td>
<td>LDA RX</td>
<td>LOAD ACCUMULATOR FROM REGISTER</td>
</tr>
<tr>
<td>1A NOTE 2</td>
<td>LDA #</td>
<td>LOAD ACCUMULATOR WITH IMMEDIATE DATA</td>
</tr>
<tr>
<td>BX NOTE 1</td>
<td>SUB RX</td>
<td>SUBTRACT REGISTER FROM ACCUMULATOR</td>
</tr>
<tr>
<td>1B NOTE 2</td>
<td>SUB #</td>
<td>SUBTRACT IMMEDIATE FROM ACCUMULATOR</td>
</tr>
<tr>
<td>CX NOTE 1</td>
<td>ADD RX</td>
<td>ADD REGISTER TO ACCUMULATOR</td>
</tr>
<tr>
<td>1C NOTE 2</td>
<td>ADD #</td>
<td>ADD IMMEDIATE TO ACCUMULATOR</td>
</tr>
<tr>
<td>DX NOTE 1</td>
<td>MULT RX</td>
<td>SIGNED MULTIPLY OF ACCUMULATOR BY REGISTER</td>
</tr>
<tr>
<td>1D NOTE 2</td>
<td>MULT #</td>
<td>SIGNED MULTIPLY OF ACCUMULATOR BY IMMEDIATE</td>
</tr>
<tr>
<td>EX NOTE 1</td>
<td>DIV RX</td>
<td>SIGNED DIVIDE OF ACCUMULATOR BY REGISTER</td>
</tr>
<tr>
<td>1E</td>
<td>WAIT</td>
<td>WAIT THE SPECIFIED TIME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1F</td>
<td>OPEN</td>
<td>OPEN SPECIFIED SWITCH</td>
</tr>
<tr>
<td>20</td>
<td>CLOSE</td>
<td>CLOSE SPECIFIED SWITCH</td>
</tr>
<tr>
<td>21</td>
<td>INPUT</td>
<td>READ THE A/D</td>
</tr>
<tr>
<td>22</td>
<td>CONV</td>
<td>CONVERT THE A/D DATA TO REAL</td>
</tr>
<tr>
<td>23</td>
<td>MEAS</td>
<td>TAKE A MEASUREMENT</td>
</tr>
<tr>
<td>24</td>
<td>SETDEV</td>
<td>SET A DAC OR D/S TO A VALUE</td>
</tr>
<tr>
<td>25</td>
<td>CHECK</td>
<td>CHECK THE OUTPUT OF A DAC</td>
</tr>
<tr>
<td>26</td>
<td>MONV</td>
<td>MONITOR A PIN'S VOLTAGE</td>
</tr>
<tr>
<td>27</td>
<td>TRANS</td>
<td>MONITOR A PIN FOR A TRANSITION</td>
</tr>
<tr>
<td>28</td>
<td>INCD</td>
<td>INCREMENT DAC OR D/S VALUE</td>
</tr>
<tr>
<td>29</td>
<td>DECD</td>
<td>DECREMENT DAC OR D/S VALUE</td>
</tr>
<tr>
<td>2A</td>
<td>CHANGE</td>
<td>CHECK FOR ADC VALUE CHANGE</td>
</tr>
<tr>
<td>2B</td>
<td>ADJDEV</td>
<td>ADJUST THE SPECIFIED DEVICE</td>
</tr>
<tr>
<td>2C</td>
<td>READEV</td>
<td>READ THE DEVICE'S CURRENT VALUE</td>
</tr>
</tbody>
</table>

NOTE: 1) 'X' in the opcode field refers to one of the sixteen general registers (0--F hexadecimal).
2) # refers to immediate data.

The definitions of pseudo instructions are given in sections 3.6.1 through 3.6.51. These definitions employ the following notation:

<table>
<thead>
<tr>
<th>P-CODE PARAMETER</th>
<th>SYMBOLIC FORMAT</th>
<th>OBJECT FORMAT</th>
<th>REMARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) &lt;ADC address&gt;</td>
<td>% &lt;1 to 6 characters&gt;</td>
<td>24 bit word</td>
<td>4 spare bits 4 CRU Index bits 8-bit Multiplex number 8-bit Channel number</td>
</tr>
<tr>
<td>2) &lt;device address&gt;</td>
<td>'DS0'...'DS255'</td>
<td>Bit 15 = 1 Bit 8 = 0 Bits 7-0 = INDEX DIGITAL/SYNCHRO, movement in a clockwise direction.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'DSC0'...'DSC255'</td>
<td>Bit 15 = 1 Bit 8 = 1 Bits 7-0 = INDEX DIGITAL/SYNCHRO, movement in a counter clockwise direction.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'DC0'...'DC255'</td>
<td>Bit 14 = 1 Bits 7-0 = INDEX DC DAC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'AC0'...'AC255'</td>
<td>Bit 13 = 1 Bit 8 = 0 Bits 7-0 = INDEX AC DAC, in phase.</td>
<td></td>
</tr>
</tbody>
</table>
'ACCO'-ACC255' Bit 13 = 1 AC DAC
Bit 8 = 1 180 out of
Bits 7-0 = INDEX phase.

3) <gain spec> 'G0'...'G3' 8 bits A/D gain
4) <conditioning spec> 'C0'...'C16' 8 bits hardware pre-conditioning
5) <switch address> 'S0'...'SFFFF' 16 bits
6) <value> +9999.999 32 bits floating point (the sign is optional.)
7) <tolerance> +-9999.999 32 bits floating point (The '+' is required.)
8) <time> 9999.999 32 bits seconds
9) <step size> +99.999 32 bits step value - volts, deg, etc. (the sign is optional.)
10) <register> R0 ... R15
11) <string> 'up to 80 characters' 8-bit ASCII characters always ended in object format by a null byte.
12) <label reference> 10 character label ID 16-bit offset see section 3.4

The referenced label precedes a P-instruction. It must be terminated by a colon (:). At most 10 labels are permitted within any subsection. Moreover, for each label, there can be at most 10 forward branches.

3.6.1 NOP

Opcode: 06 hex
Results: No operation is performed
Status bits affected: None

3.6.2 OUTSTR <String>

Opcode: 02 hex
Results: Copies the ASCII following the opcode byte to active I/O devices until a null (0) termination character is detected.
3.6.3 GETSTR

Opcode: 03 hex

Results: Retrieves characters from the operator console until a termination character is received. All characters are copied into the string input (SI) buffer. The termination character causes the instruction to terminate with the remainder of the SI buffer blank filled.

Status bits affected: None

3.6.4 CMPSTR <String>

Opcode: 04 hex

Results: The immediate ASCII string is compared byte by byte with the contents of the string buffer. Characters are compared until the termination character is encountered within the immediate string or non-matching characters are detected. Example: A CMPSTR 'Y' instruction would match string buffer data of 'Y', 'YES', 'YEAH', etc. A CMPSTR 'YES' instruction would match any string buffer data containing 'YES' as the first three characters.

Status bits affected: EQ

3.6.5 OUTDEC

Opcode: 05 hex

Results: Converts the contents of the A register to a decimal ASCII string. The string is copied to the active I/O devices.

Status bits affected: None

3.6.6 GETDEC

Opcode: 06 hex

Results: Converted ASCII numeric data is moved to the A register. Retrieves numeric characters from the operator console until a termination char is received. The ASCII data is converted to 32 bit binary and stored in the A register.

Termination chars: Carriage return.
Status bits affected: Z, M

3.6.7 SETCLK
 Opcode: 07 hex
 Results: Set the test system clock with the contents of A.
 Status bits affected: None

3.6.8 GETCLK
 Opcode: 08 hex
 Results: Set the accumulator (A) with the contents of the system clock.
 Status bits affected: Z, M

3.6.9 BRA <label reference>
 Opcode: 09 hex
 Results: PC <- PC + offset
 Status bits affected: None

3.6.10 BLT <label reference>
 Opcode: 0A hex
 Results: If status bit 'LT' is set, then PC <-- PC + offset.
 Status bits affected: None

3.6.11 BGT <label reference>
 Opcode: 0B hex
 Results: If status bit 'GT' is set, then PC <-- PC + offset.
 Status bits affected: None
3.6.12 BLE <label reference>

Opcode: OC hex

Results: If status bit 'EQ' or status bit 'LT' is set, then PC <-- PC + offset.

Status bits affected: None

3.6.13 BGE <label reference>

Opcode: OD hex

Results: If status bit 'GT' or status bit 'EQ' is set, then PC <-- PC + offset.

Status bits affected: None

3.6.14 BEQ <label reference>

Opcode: OE hex

Results: If status bit 'EQ' is set, then PC <-- PC + offset.

Status bits affected: None

3.6.15 BNE <label reference>

Opcode: OF hex

Results: If status bit 'EQ' is not set, then PC <-- PC + offset.

Status bits affected: None

3.6.16 BM <label reference>

Opcode: 10 hex

Results: If status bit 'M' is set, then PC <-- PC + offset.

Status bits affected: None

3.6.17 BP <label reference>

Opcode: 11 hex

Results: If status bit 'M' is not set, then PC <-- PC + offset.

Status bits affected: None
3.6.18 BZ <label reference>
 Opcode: 12 hex
 Results: If status bit 'Z' is set, then PC ← PC + offset.
 Status bits affected: None

3.6.19 BNZ <label reference>
 Opcode: 13 hex
 Results: If status bit 'Z' is not set, then PC ← PC + offset.
 Status bits affected: None

3.6.20 ABS
 Opcode: 14 hex
 Results: Take absolute value of register A. A ← |A|
 Status bits affected: Z, M

3.6.21 NEG
 Opcode: 15 hex
 Results: Negate register A. A ← -A
 Status bits affected: Z, M

3.6.22 CLA
 Opcode: 16 hex
 Results: Clear accumulator (A). A ← 0
 Status bits affected: Z, M

3.6.23 STOP
 Opcode: 17 hex
 Results: Normal termination of a test.
 PC ← Standard Procedure 2.
 Status bits affected: None
3.6.24 CMP RX

Opcode: 8X hex X = 0..F
Results: Compare contents of A with contents of register X and set appropriate status bits.
If A = RX then 'EQ' set.
If A < RX then 'LT' set.
If A > RX then 'GT' set.
Status bits affected: EQ, LT, GT

3.6.25 CMP <value>

Opcode: 18 hex
Results: Compare contents of A with immediate data and set appropriate status bits.
If A = <value> then 'EQ' set.
If A < <value> then 'LT' set.
If A > <value> then 'GT' set.
Status bits affected: EQ, LT, GT

3.6.26 STA RX

Opcode: 9X hex X = 0..F
Results: Store A into register X. RX <-- A
Status bits affected: None

3.6.27 LDA RX

Opcode: AX hex X = 0..F
Results: Load A from register X. A <-- RX
Status bits affected: Z, M

3.6.28 LDA <value>

Opcode: 19 hex
Results: Load A with immediate data. A <-- <value>
Status bits affected: Z, M

3.6.29 SUB RX
Opcode: BX hex X = 0..F
Results: Subtract register X from A. A \(-= A - RX\)
Status bits affected: Z, M

3.6.30 SUB <value>
Opcode: 1A hex
Results: Subtract immediate data from A. A \(-= A - <value>\)
Status bits affected: Z, M

3.6.31 ADD RX
Opcode: CX hex X = 0..F
Results: Add register to A. A \(+= A + RX\)
Status bits affected: Z, M

3.6.32 ADD <value>
Opcode: 1B hex
Results: Add immediate data to A. A \(+= A + <value>\)
Status bits affected: Z, M

3.6.33 MULT RX
Opcode: DX hex X = 0..F
Results: Signed multiply of A by register X. A \(\*< A * RX\)
Status bits affected: Z, M
3.6.34 MULT \(<value>\)

Opcode: \(1C\) hex

Results: Signed multiply of \(A\) by immediate data. \(A \leftarrow A \times \langle value\rangle\)

Status bits affected: \(Z, M\)

3.6.35 DIV RX

Opcode: \(EX\) hex \(X = 0..F\)

Results: Signed divide of \(A\) by register \(X\). \(A \leftarrow A/RX\)

Status bits affected: \(Z, M\)

3.6.36 DIV \(<value>\)

Opcode: \(1D\) hex

Results: Signed divide \(A\) by immediate data. \(A \leftarrow A / \langle value\rangle\)

Status bits affected: \(A, M\)

3.6.37 WAIT \(<time>\)

Opcode: \(1E\) hex

Results: Program execution suspends until the 32 bit \(<time>\) reference has elapsed. The time is given in seconds.

Status bits affected: None

The WAIT instruction could be written in the low level p-codes as follows:

\[
\begin{align*}
CLA \\
SETCLK \\
\text{LOOP:} \\
\quad \text{GETCLK} \\
\quad \text{CMP} \ <time> \\
\quad \text{BLT LOOP}
\end{align*}
\]

3.6.38 OPEN \(<\text{switch address}>\)
Opcode: 1F hex

Results: The switch specified by the two byte <switch address> is opened.

Status bits affected: None

3.6.39 CLOSE <switch address>

Opcode: 20 hex

Results: The switch specified by the two byte <switch address> is closed.

Status bits affected: None

3.6.40 INPUT <ADC address>, <gain spec>

Opcode: 21 hex

Results: Set the ADC to the specified gain and input a word from the ADC address into the A register and R10. R10 <-- A

Status bits affected: Z, M

3.6.41 CONV <gain spec>, <conditioning spec>

Opcode: 22 hex

Results: Convert the contents of A into a 'REAL' value by applying the gain spec and then the conditioning spec. Leave the result in A and R11. R11 <-- A

Status bits affected: Z, M
3.6.42 MEAS <ADC address>, <gain spec>, <conditioning spec>, <desired value>, <tolerance>

Opcode: 23 hex

Results: Set the ADC to the appropriate gain, input the ADC word, convert according to gain and conditioning, and check if within tolerance. If not in tolerance branch to standard error processing 3 with registers set:

\[
\begin{align*}
R10 & \quad \text{(ADC data)} \\
R11 & \quad \text{(converted ADC data)} \\
R12 & \quad \text{(desired value)} \\
R13 & \quad \text{(tolerance)}
\end{align*}
\]

Status bits affected: Undefined

The MEASURE statement is equivalent to the following P-code sequence:

\[
\begin{align*}
&\text{LDA } \text{<desired value>} \\
&\text{STA } R12 \\
&\text{LDA } \text{<tolerance>} \\
&\text{STA } R13 \\
&\text{INPUT } \text{<gain spec>, <ADC address>} \\
&\text{CONV } \text{<gain spec>, <conditioning spec>} \\
&\text{SUB } R12 \\
&\text{ABS} \\
&\text{CMP } R13 \\
&\text{OUTSTR } "\text{MEASURED: "} \\
&\text{LDA } R11 \\
&\text{OUTDEC} \\
&\text{OUTSTR } "\text{WITHIN TOLERANCE"}
\end{align*}
\]

3.6.43 SETDEV <device address>, <value>, <tolerance>

Opcode: 24 hex

Results: set the specified device to the desired value after applying the implicit scale factor. If the D/S is being addressed, bit 8 of the device address specifies the direction of movement; i.e. if set, counterclockwise; if cleared, clockwise.

\[
\begin{align*}
R12 & \quad \text{(value)} \\
R13 & \quad \text{(tolerance)}
\end{align*}
\]

Status bits affected: Undefined
3.6.44 CHECK <ADC address>, <gain spec>, <conditioning spec>

Opcode: 25 hex

Result: checks that the device is set to the value in R12 within tolerance contained in R13. If not in tolerance branch to standard error processing 4 with registers set:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10</td>
<td>(ADC data)</td>
</tr>
<tr>
<td>R11</td>
<td>(converted data)</td>
</tr>
<tr>
<td>R12</td>
<td>(desired value)</td>
</tr>
<tr>
<td>R13</td>
<td>(tolerance)</td>
</tr>
</tbody>
</table>

Check is most commonly used after a 'SETDEV'.

Status bits affected: Undefined

3.6.45 MONV <ADC address>, <gain spec>, <conditioning spec>, <desired value>, <tolerance>, <time>

Opcode: 26 hex

Results: Check that the converted value read from the ADC address remains in tolerance for the time specified. If not, branch to standard error processing 5 with registers set as follows:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10</td>
<td>(ADC data)</td>
</tr>
<tr>
<td>R11</td>
<td>(converted ADC data)</td>
</tr>
<tr>
<td>R12</td>
<td>(desired value)</td>
</tr>
<tr>
<td>R13</td>
<td>(tolerance)</td>
</tr>
</tbody>
</table>

Status bits affected: undefined after execution
The Monitor P-code can be written with the following low level p-code sequence:

```assembly
LDA <desired value> (set up for possible error processing)
STA R12
LDA <tolerance>
STA R13
CLA
SETCLK
LOOP: INPUT <ADC address>, <gain spec>
CONV <gain spec>, <conditioning spec>
SUB R12
ABS
CMP R13
BGT SP5
GETCLK
CMP <time>
BLT LOOP

(OK)
```

3.6.46 TRANS <ADC address>, <gain spec>, <conditioning spec>, <value 1>, <tolerance 1>, <value 2>, <tolerance 2>, <time>, <time tolerance>

**Opcode:** 27 hex

**Results:** Continuously monitor the specified ADC for value 1. The voltage should remain within tolerance 1 until at least time minus time tolerance. The voltage must change to value 2 within tolerance 2 before time plus time tolerance.

**Status bits affected:** Undefined

The Transition P-code can be written as the following sequence:

```assembly
CLA
SETCLK
CMP <time> (t=0 ?)
BEQ REPEAT 1
REPEAT: INPUT <ADC address>, <gain spec>
CONV <gain spec>, <conditioning spec>
SUB <value 1>
ABS
CMP <tolerance 1>
BGT SP6
GETCLK
SUB <time>
CMP <- time tolerance>
BLT REPEAT
```
REPEAT1: GETCLK
SUB <time>
CMP <+time tolerance>
BGT SP6 (the actual P-machine)
   (will expand the time tolerance before
    erroring to SP 6)
INPUT <ADC address>, <gain spec>
CONV <gain spec>, <conditioning spec>
SUB <value 2>
ABS
CMP <tolerance 2>
BGT REPEAT1

(OK)

3.6.47 INCD <device address>, <step size>
Increment the value of the DAC or D/S at device address.
Opcode: 28
Status bits affected: None

3.6.48 DECD <device address>, <step size>
Decrement the value of the DAC or D/S at device address.
Opcode: 29
Status bits affected: None

3.6.49 CHANGE <register>, <ADC address>, <gain spec>, <conditioning spec>,
      <change value>, <tolerance>
Opcode: 2A hex
Results: Set the ADC to the appropriate gain, input the ADC value,
         convert the value using the gain and conditioning specs.
         The converted value shall have changed from the value
         contained in the specified register by 'change value'
         within tolerance. If not in tolerance branch to standard
         error processing 12 with registers set:
         R10 <--- (ADC data)
         R11 <--- (converted ADC data)
         R12 <--- (change value)
         R13 <--- (tolerance)
         R14 <--- (initial value)

50
Status bits affected: Undefined

The CHANGE statement is equivalent to the following P-code sequence:

```
LDA <register>
STA R14
LDA <change value>
STA R12
LDA <tolerance>
STA R13
INPUT <ADC address>, <gain spec>
CONV <gain spec>, <conditioning spec>
SUB R14 (initial value)
SUB R12 (change amount)
ABS
CMP R13 (tolerance)
BGT SP12
```

3.6.50 ADJDEV <device address>, <ADC address>, <gain spec>, <conditioning spec>, <value>, <tolerance>, <step size>, <limit value>, <step time>

 Opcode: 2B hex

Results: Adjust the device starting at its current value until the voltage at the ADC by step site per time address is at 'value' within tolerance. Note that <step size> can be positive or negative. If unable to set the device within tolerance branch to standard error processing with registers set:

```
R12 <--- (desired value)
R13 <--- (tolerance)
```

Status bits affected: Undefined

The adjust device P-code can be written using the following low-level P-codes:

```
READV <device address>
STA R15 (loop counter)
LDA <value>
STA R12
LDA <tolerance>
STA R13
LOOP:
INPUT <ADC address>, <gain spec>
CONV <gain spec>, <conditioning spec>
SUB R12 (desired value)
ABS
CMP R13 (tolerance)
BLE DONE WAIT <step time>
```
INCD <device address>, <step size> (increment/decrement device)
LDA R15
ADD <step size>
STA R15
LDA <step size>
BM NEG (NEGATIVE step size?)
LDA R15
CMP <limit value> (exceeded maximum value?)
BLT LOOP
BRA SP13
NEG:
LDA R15
CMP <limit value> (exceeded minimum value?)
BGE LOOP
BRA SP13

DONE:

3.6.51 READEV <device address>

Opcode: 2C hex

Results: Load the accumulator with the last value written to the specified device.

Status bits affected: Z, M
4.0 HARDWARE DEFINITION FILE/TABLES

The hardware definition file specifies the hardware configuration of the test set to the compiler. The file contains information relating to pins and their possible connections. Also contained in the file are data pertaining to DACS, voltage tolerances, and gains. All information required at compile time which relates to the hardware configuration is specified by this file.

The file should be supplied in the format as described in the following section. The ASCII source file will then be converted into an encoded form compatible with the compiler. For further information on the encoded file, its production, and its use, see Section 5.3 of this manual.
4.1 HARDWARE DEFINITION FILE — SOURCE FORM

4.1.1 OVERALL STRUCTURE

The hardware definition source file will be made up of several segments; each segment will contain a different type of information. For example, one segment will contain information about pins, their A/D addresses, types of connections, etc.; one segment will contain information pertaining to DACS; one segment will have voltage tolerances and ranges; etc.

Each segment will begin with an identifier line which will tell what type of information is in the segment and how many lines of that type of information are following. Thus, the form of the file will be essentially:

```
IDENTIFIER LINE
INFORMATION LINE 1
SEGMENT
  ...
INFORMATION LINE a
```

```
IDENTIFIER LINE
INFORMATION LINE 1
SEGMENT
  ...
INFORMATION LINE b
```

```
IDENTIFIER LINE
INFORMATION LINE 1
SEGMENT
  ...
INFORMATION LINE c
```
The ordering of segments within the file is unimportant with the exception that the segment containing pin information should be the last segment in the file. (Since the pin segment is last, the number of information lines in it may be left blank on its identifier line).

4.1.2 FORMATTING OF LINES

Except for the identifier line, each segment will have a different line format. No line will exceed 80 characters in length. Unless otherwise indicated all information should be left justified within the given column boundaries. Also there should be no embedded blanks in any string; in other words, if a data item is left justified within the given column boundaries and is shorter than the allowable width, only trailing blanks will be considered valid. For example, if the allowable width is 8 characters, the data item is "DS01" and it is to be left justified

\[ D S 0 1 \quad \_ \_ \_ \_ \_ \_ \]

is the only correct entry.

\[ D S 0 1 \quad \_ \_ \_ \_ \_ \_ \]

would be incorrect because of the embedded blank.

\[ \_ D S 0 1 \quad \_ \_ \_ \_ \] 

would be incorrect because the data is not left justified.
4.1.2.1 FORMAT OF IDENTIFIER LINE

The identifier line, being the first line in every segment, will contain a code indicating which segment is upcoming and the number of lines in that segment that follow the identifier line.

FORMAT

2A2 COLUMNS 1-4 should contain the four character type code (left justified)

I3 COLUMNS 9-11 should indicate the number of lines following of the given type (right justified)

Valid type codes (ASCII strings) are:

<table>
<thead>
<tr>
<th>CODE</th>
<th>TYPE OF INFORMATION IN THAT SEGMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAC_</td>
<td>DAC numbers and types, high and low values</td>
</tr>
<tr>
<td>GAIN</td>
<td>A/D gains and corresponding ranges</td>
</tr>
<tr>
<td>COND</td>
<td>conditioning factors</td>
</tr>
<tr>
<td>ANGL</td>
<td>angle tolerance</td>
</tr>
<tr>
<td>VOLT</td>
<td>voltage tolerances and ranges</td>
</tr>
<tr>
<td>PIN_</td>
<td>pin identifiers, their addresses, valid connections,</td>
</tr>
<tr>
<td></td>
<td>switches involved, etc.</td>
</tr>
</tbody>
</table>
4.1.2.2 FORMAT OF "DAC " SEGMENT RECORDS

This segment gives a list of all DACS and their associated ranges. All information lines within the "DAC_" segment should be of the following format.

FORMAT

A2 COLUMNS 1-2 will indicate the type of DAC; "DC" or "AC" are valid entries (ASCII strings)

I3 COLUMNS 3-5 will contain the DAC number; an integer right justified

F10.4 COLUMNS 9-18 will contain a real number (volts) specifying the low value for the DAC

F10.4 COLUMNS 25-34 will contain a real number (volts) specifying the high value for the DAC

F10.4 COLUMNS 41-50 will contain a real number (degrees) specifying the phase of the DAC (NOTE: AC DACS only; this field should be left blank for DC DACS)

Ordering of records within the "DAC_" segment should result in having all DC DAC's in sequential order by number followed by AC DAC's in sequential order by number. DAC numbers should begin at zero and increase by increments of one.
4.1.2.3 FORMAT OF "GAIN" SEGMENT RECORDS

This segment contains gain specs and associated ranges. Gains should begin at zero and increase by increments of 1; their associated ranges should be in descending order.

**FORMAT**

I1 COLUMN 1 Gain spec (0,1,2,3,)
F6.3 COLUMNS 9-14 Half range in volts (symmetrical ranges assumed, so only positive boundaries are entered)

4.1.2.4 FORMAT OF "COND" SEGMENT RECORDS

This segment contains the hardware conditioning factors. Conditioning specs should be in ascending order in increments of one beginning at zero.

**FORMAT**

I2 COLUMNS 1-2 Conditioning Spec (0 to 15 decimal)
E14.7 COLUMNS 9-22 Conditioning factor

4.1.2.5 FORMAT OF "ANGL" RECORD (D/S DEFAULT TOLERANCE)

The ANGL segment contains exactly one information line.

**FORMAT**

F5.3 COLUMNS 1-5 Default angle tolerance
4.1.2.6 FORMAT OF "VOLT" SEGMENT RECORDS

This segment indicates the default voltage tolerances for associated voltage ranges -- either AC or DC. "VOLT" records should be in the following format with DC ranges and tolerances followed by AC ranges and tolerances.

FORMAT

A2  COLUMNS 1-2  type ("DC" or "AC")
F8.3 COLUMNS 9-16  low voltage of range
F8.3 COLUMNS 25-32  high voltage of range
F5.3 COLUMNS 41-45  default voltage tolerance for associated range
(positive value)

4.1.2.7 FORMAT OF "PIN " SEGMENT RECORDS

This segment contains pin identifiers, the A/D addresses (made up of CRU pointer, state, channel) possible connections that can be made to each pin, and A/D scale factors.

FORMAT

5A2  COLUMNS 1-10  Pin identifier. Up to 10 ASCII characters constitute a valid pin identifier. Left justified.
Ex. COLUMN 1 2 3 4 5 6 7 8 9 10
     J 1 0 1 A - 2 _ _
A1  COLUMNS 17  CRU pointer. 1 Hex digit.
A2  COLUMNS 25-26  state 2 Hex digits
A2  COLUMNS 33-34  channel 2 Hex digits
5A2 COLUMNS 41-50 Type of connection. Left Justified with trailing blanks. Valid types are:
1. a 10 character pin identifier
2. DC DACS (ex. "DC0", "DC1", ..., "DC255")
3. AC DACS (ex. "AC0", "AC1", ..., "AC255")
4. D/S PINS (ex. "DS01", "DS02", "DS03")
5. LOGIC DISCRETES:
   "LD+10", LD0, "LD+12", "OPENC"
6. MANUAL (operator controlled pins)
7. MONITOR
8. continuation (designated as "**"). The type of connection is the same as that of the preceding information line. The information in columns 57-59, 65-68, and 73-74 lists an additional switch closure required for connecting the given pin to the specified type.

A1,A2 COLUMNS 57-59 Switch identifier. 3 Hex digits.
2A2 COLUMNS 65-68 Switch state. "OPEN" or "CLOS"
I2 COLUMNS 73-74 A/D conditioning factor. Integer right justified. ("00" to "99" decimal)

Figure 1 illustrates the source form of the "PIN_" segment.
<table>
<thead>
<tr>
<th>Model</th>
<th>Code</th>
<th>Address</th>
<th>Function</th>
<th>Action</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>J101A-109</td>
<td>8</td>
<td>2A</td>
<td>16</td>
<td>MONITOR</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OPENC</td>
<td>950</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LD+28</td>
<td>950</td>
</tr>
<tr>
<td>J101A-110</td>
<td>8</td>
<td>3A</td>
<td>16</td>
<td>MONITOR</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OPENC</td>
<td>A38</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDO</td>
<td>A38</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>A39</td>
</tr>
<tr>
<td>J101A-111</td>
<td>5</td>
<td>93</td>
<td>14</td>
<td>MONITOR</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 1: HARDWARE DEFINITION FILE - SOURCE FORM
4.2 HARDWARE DEFINITION FILE - CONVERTED FORM

The source file after conversion becomes three disk files: x.D1, x.D2, x.D3, where x is a file-name prefix. These files contain connection data, pin identifiers, and information needed in common by other programs, respectively. (See 5.3)

These files may be represented symbolically as interconnected tables of pin identifiers and their valid connections, along with necessary switch data to actually make the connections.

4.2.1 COMMON BLOCK FILE (x.D3)

This sequential file is used to contain information which is to be stored in the common blocks of the ATP compiler. This information includes:

1) A table of sorted pin id's and associated indexes into the random-access connection file (x.D1)
2) DAC specifications
3) A/D gain ranges
4) A table of conditioning factor indexes and associated conditioning factors
5) The default D/S tolerance
6) The voltage tolerances

4.2.2 CONNECTION FILE (x.D1)

This is a random-access file containing the information given in the "PIN_" segment (see 4.1.2.7). Given a specified pin id, the ATP compiler accesses this file by using an index retrieved from its common block table of sorted pin ids and associated indexes. This index points to the set of related information for the given pin.
4.2.2.1 FORMAT OF CONNECTION_FILE (x.D1)

The connection file specifies a pin's A/D address as well as the connections that can be made to that particular pin. The connection file contains two types of records. One record specifies the A/D address and the number of the other type of records immediately following.

The second type of record contains the actual type of connection, switch settings, etc. Thus, there is only one record of the first type for each pin and a variable number of records of the second type for each pin (the number being specified in the first record).

Ex.  

A/D ADDR  3

CONNECTION INFO

CONNECTION INFO

CONNECTION INFO

A/D ADDR  2

CONNECTION INFO

CONNECTION INFO
The form of the first record:

1 16-bit word filled with 1's (to indicate record type)
3 16-bit words containing the ASCII A/D address
1 16-bit word containing the number of "connection-type" records following

The form of the "connection-type" records:

1 16-bit word containing a connection type code
2 16-bit words containing an ASCII switch ID
1 byte containing an index into the conditioning factor table
1 byte containing the switch status (OPEN =0  CLOS =1)
1 16-bit word containing the connection value

The first word of the second type of record, the connection type field, contains a type code which identifies the connection as follows:

=0 Continuation of previous records (used to specify more switch closures)
=1 Connection with another pin (NOTE: In this case, the associated connection value (the last word of this record) is an index into the pin id table where the second pin's identifier is stowed.)

=2 Connection with logic discretes — including open circuits (NOTE: An open circuit is indicated by type =2 with a value of 99 in the associated connection value.)

=3 Connection with a digital/synchro

=4 Connection with an AC DAC

=5 Connection with a DC DAC

=6 Monitor connection

=7 Manual connection
4.2.3 PIN ID FILE (x.D2)

The pin id file is a random access file which contains the identifiers of pins which connect to other pins. Each record in it is of the form:

5 16-bit words containing 10 character ASCII
pin identifier

The pin id file is associated with entries of the connection file with a connection type of 1. The connection values for these entries are indexes into the appropriate record in the pin id file.
5.0 OPERATING PROCEDURES

Before the ATP compiler can be run, several files must be prepared: the source ATP file and the converted hardware definition files. The source ATP file contains a test specification procedure written in the ATP language. The converted hardware definition files contain information on the hardware configuration being used and are produced by a translator from the source hardware definition file. This translator needs to be rerun each time a change is made to the source hardware definition file.

To access the compiler the user must simply type: COMPILE. The files to perform the compilation should be found under the current directory.

The three following sections, 5.1, 5.2, and 5.3 contain more specific information on how to use the compiler. These three sections give additional details on how to properly answer the questions asked by the compiler, the assembler, and the hardware definition translator.
5.1 **COMPILER USAGE**

To invoke the ATP compiler a macro called COMPILE has been created. Once the compiler has been invoked it will then ask the user a series of questions:

= = => INPUT SOURCE FILE NAME:

Here the name of the file containing the ATP test specification procedure should be typed followed by a carriage return. The compiler will then respond

= = => SOURCE FILE IS: Source File Name

**DO YOU WISH TO HAVE A SOURCE LISTING PRODUCED? (Y/N)**

Here a simple Y for YES or N for NO will suffice to answer the question as to whether or not the user desires to have a source listing of the ATP test specification procedure.

= = => TEST SPECIFICATION NUMBER NOT FOUND IN SOURCE

= = => DO YOU WISH TO STOP? (Y/N)
This message will be printed only if a test specification number is not found on the first line of the ATP test specification procedure source file. If the user wishes to process the file, Y (for YES) should be typed followed by a carriage return. If the user wishes to stop at this point N (for NO) may be typed followed by a carriage return.

= = = => TEST SPECIFICATION NUMBER: Test specification number

Here the compiler simply displays the test specification number that was found in the source file. If no number was found then blanks will be printed for the test specification number.

After the compiler has completed processing the input file, it will print a message indicating the number of errors found, if any.
5.2 **ASSEMBLER USAGE**

Once the compiler has completed its processing the assembler is invoked and will ask the following series of questions:

> DO YOU WANT AN OBJECT FILE PRODUCED [Y/N]?

Here the user is asked to respond to the question with Y for YES or N for NO. The object file will contain the assembled P-code. If the user responds with Y then the assembler will request a file name:

> SPECIFY THE OBJECT FILE NAME:

> DO YOU WANT A LISTING OF THE SOURCE P-CODE [Y/N]?

Again a yes or no question is posed to the user as to whether or not a listing of the source P-code is desired.
If the user responds with a Y, then the assembler will ask the following YES or NO question as to whether or not the user wishes to have the object code printed on the listing with the P-code:

\[
= = = = \Rightarrow \text{DO YOU WANT THE OBJECT CODE PRINTED ON THE LISTING [Y/N]?}
\]

After the assembler has completed processing the input file it will print a message indicating the number of errors found by the assembler. At this point the process of compilation and assembly have been completed.

5.3 HARDWARE DEFINITION CONVERSION PROCESS

The hardware definition source file is converted into a form suitable for the ATP compiler through the program "HCONV". "HCONV" is typed at the terminal to initiate the conversion process. The user will then be asked to specify the source file name with a .DF extension, (i.e., "HW747.DF"). The name of this file must be identical to the name cited in the HARDWARE DEFINITION FILE statement.

The conversion process consists of two parts. First all tab characters are removed from the source file; this new source file becomes HW747.DT. This file is then converted into three files used by the ATP compiler: HW747.D1, HW747.D2, HW747.D3. These files are the random access file containing pin connection information, the random access file containing pin ID's which connect to other pins, and the sequential file containing common block information, respectively.

Any changes made to the source form of the file requires "HCONV" to
be run in order to incorporate the changes. However, if no changes are made to the hardware configuration and no errors result when executing "HCONV", then the program needs only to be executed once.
APPENDIX 1 - BNF

The syntax of the language processed by the ATP compiler is given here in Backus Naur Form or BNF. BNF is a commonly used notation to write grammars which specify the syntax of programming languages.

BNF notation is used as a concise and explicit method of defining the language formally. English is not suitable for formal language definition due to its vagueness and because it leads to ambiguous definitions.

The BNF as the defining language is called the syntactic meta-language. The meta-language is a language which is used to talk about another language or the object language. The object language in this case is the language to be processed by the ATP compiler. Symbols of the object language are called terminals. Symbols of the meta-language which denote strings in the object language are called nonterminals.

Nonterminals in BNF are expressed as names written in corner-brackets '<>'. The syntax of the object language is given as a set of "rewriting rules". These rules consist of a nonterminal on the left-hand side of the rewriting rule, the ::= sign indicating that left-hand side of the rule is replaced by the right-hand side, and the sequence of terminals and/or nonterminals which make up the right-hand side of the rewriting rule.

Within the right-hand side of the rule, alternative ways of rewriting a given nonterminal are separated by a vertical bar, | (read "or"). Also on the right side items written in curly brackets, {}, may be repeated zero or more times. Any item which may be considered optional within a rule has been enclosed in square brackets. For example, in the case of the rewriting rule:

\[\text{<SUBSECTION NUMBER> ::= DIGIT STRING \{DIGIT STRING}\}]\]
the nonterminal is SUBSECTION NUMBER (enclosed in corner-brackets) with ::= indicating that the nonterminal SUBSECTION NUMBER is to be replaced by the right-hand side of the rule. The right-hand side of the rule consists of a string of appropriate digits separated by periods. In this case the period followed by a digit string may be repeated zero or more times giving clause numbers of the form 5, 5.5, 5.4.3 etc.

In the case of the STATEMENT rewriting rule (see the BNF) the nonterminal <STATEMENT> may be replaced by any one of the types of statements found on the right-hand side of the rule. Here, <STATEMENT> may be replaced by an <ADJUST STATEMENT> or by an <SET STATEMENT> or by a <CONNECT STATEMENT> and so on.

Not every form of every statement type has been kept in the object language. One of a kind statements can easily be converted to the standard format developed here, and their omission from the language will help to avoid overburdening the language and the compiler which must process it. Variations on the standard statement types have also been limited for the same reason.
BNF FOR THE ATP LANGUAGE

\(<\text{TEST\ PROCEDURE}>\ ::= \ <\text{TEST\ SPECIFICATION\ STATEMENT}>\ <\text{HARDWARE\ DEFINITION\ FILE\ STATEMENT}>\ <\text{SECTION}>\ \{\ <\text{SECTION}>\ \}\n
\(<\text{SECTION}>\ ::= \ \text{BEGIN}\ <\text{SECTION\ NAME}>\ \{\ <\text{SUBSECTION}>\ \}\ \text{END}\n
\(<\text{SECTION\ NAME}>\ ::= \ <\text{SECTION\ NUMBER}>\ <\text{CHARACTER\ STRING}>\n
\(<\text{SECTION\ NUMBER}>\ ::= \ <\text{DIGIT\ STRING}>\ \{\ .\ <\text{DIGIT\ STRING}>\ \}\n
\(<\text{SUBSECTION}>\ ::= \ \text{BEGIN}\ <\text{SUBSECTION\ NUMBER}>\ <\text{STATEMENT}>\ \{\ <\text{STATEMENT}>\ \}\ \text{END}\n
\(<\text{SUBSECTION\ NUMBER}>\ ::= \ <\text{DIGIT\ STRING}>\ \{\ .\ <\text{DIGIT\ STRING}>\ \}\ \{\ <\text{VARIANT}>\ \}\n
\(<\text{VARIANT}>\ ::= \ /\ <\text{DIGIT\ STRING}>\ /\n
\(<\text{STATEMENT}>\ ::= \ <\text{ADJUST\ STATEMENT}>\ |\ <\text{SET\ STATEMENT}>\ |\ <\text{CONNECT\ STATEMENT}>\ |\ <\text{DISCONNECT\ STATEMENT}>\ |\ <\text{VOLTAGE\ STATEMENT}>\ |\ <\text{WAIT\ STATEMENT}>\ |\ <\text{OPEN\ CIRCUIT\ STATEMENT}>\ |\ <\text{OPERATOR\ INTERACTION\ STATEMENT}>\ |\ <\text{SAVE\ STATEMENT}>\ |\ <\text{VOLTAGE\ CHANGE\ STATEMENT}>\ |\ <\text{START\ P-CODE\ STATEMENT}>\ |\ <\text{STOP\ P-CODE\ STATEMENT}>\ |\ <\text{MONITOR\ STATEMENT}>\n
\(<\text{CHARACTER\ STRING}>\ ::= \ <\text{CHARACTER}>\ \{\ <\text{CHARACTER}>\ \}\n
\(<\text{CHARACTER}>\ ::= \ \text{any\ printable\ ASCII\ character}\n
\(<\text{DIGIT\ STRING}>\ ::= \ <\text{DIGIT}>\ \{\ <\text{DIGIT}>\ \}\n
\(<\text{DIGIT}>\ ::= \ 0\ |\ 1\ |\ 2\ |\ 3\ |\ 4\ |\ 5\ |\ 6\ |\ 7\ |\ 8\ |\ 9\n
\(<\text{TEST\ SPECIFICATION\ STATEMENT}>\ ::=\n
\ \text{TEST\ SPECIFICATION\ NUMBER}\ =\ <\text{CHARACTER\ STRING}>\n
\(<\text{HARDWARE\ DEFINITION\ FILE\ STATEMENT}>\ ::=\n
\ \text{HARDWARE\ DEFINITION\ FILE}\ =\ <\text{CHARACTER\ STRING}>\ .\text{DF}\n
\(<\text{ADJUST\ STATEMENT}>\ ::=\n
\ \text{ADJUST\ SIGNAL\ AT}\ <\text{PIN\#}\ \text{BY}\ <\text{STEP\ SIZE}\ \text{PER}\ <\text{TIME}\ \text{IN\ A}\ <\text{SIGN}\ \text{DIRECTION\ UNTIL\ THE\ VOLTAGE\ AT}\ <\text{PIN\#}\ \text{CHANGES\ TO}\ <\text{VOLTS}\ +/-\ <\text{VOLTS}>}\ |

\ \text{ADJUST\ SIGNAL\ AT}\ <\text{PIN\#}\ \text{BY}\ <\text{STEP\ SIZE}\ \text{PER}\ <\text{TIME}\ \text{UNTIL\ THE\ VOLTAGE\ AT}\ <\text{PIN\#}\ \text{IS}\ <\text{VOLTS}\ +/-\ <\text{VOLTS}>}\ |

\ \text{ADJUST\ SYNCHRO\ TRANSMITTER\ BY}\ <\text{STEP\ SIZE}\ \text{PER}\ <\text{TIME}\ \text{SECONDS\ UNTIL\ THE}\n
75
VOLTAGE AT <PIN#> IS <VOLTS> ← <VOLTS>

<SIGN> ::= POSITIVE | NEGATIVE

<SET STATEMENT> ::= 
SET THE SYNCHRO TRANSMITTER TO <ANGLE> ← <ANGLE> [CLOCKWISE | COUNTERCLOCKWISE]

<CONNECT STATEMENT> ::= 
CONNECT <PIN#> TO <CONNECT OBJECT> |
CONNECT <SYNCHRO OBJECT> TO <PIN#> |
CONNECT <VOLTS>, <FREQUENCY> [, <ANGLE>] TO <PIN#>

<CONNECT OBJECT> ::= <PIN#> | <VOLTS> | A SIGNAL SET TO <VOLTS>

<SYNCHRO OBJECT> ::= S1 | S2 | S3

<DISCONNECT STATEMENT> ::= 
DISCONNECT <PIN#> FROM <DISCONNECT OBJECT> |
DISCONNECT SIGNAL FROM <PIN#>

<DISCONNECT OBJECT> ::= <VOLTS> | <PIN#> | SYNCHRO TRANSMITTER

<OPEN CIRCUIT STATEMENT> ::= OPEN CIRCUIT <PIN#>

<REMOVE STATEMENT> ::= 
REMOVE <REMOVE SUBJECT I> BETWEEN <PIN#> AND <PIN#> |
REMOVE <REMOVE SUBJECT II> FROM <PIN#> |
REMOVE <PIN#> FROM <VOLTAGE>

<REMOVE SUBJECT I> ::= SIGNAL | CONNECTION | <VOLTS>, <FREQUENCY> [, <ANGLE>]

<REMOVE SUBJECT II> ::= <VOLTAGE> | <PIN#>

<VOLTAGE> ::= SIGNAL | <VOLTS>
<VOLTAGE STATEMENT> ::= 
  [THE] VOLTAGE AT THE FOLLOWING POINTS SHALL BE:
  {<PIN#>  MEASURE <VOLTS> ← <VOLTS> } 
  [THE] VOLTAGE AT <PIN#> SHALL BE:
  MEASURE <VOLTS> ← <VOLTS>
  [WITHIN A PERIOD OF: CHECK <TIME> MAX ;
  FOR A PERIOD OF: CHECK <TIME> MIN ] !
  [THE] VOLTAGE BETWEEN <PIN#> AND <PIN#> SHALL BE:
  MEASURE <VOLTS> ← <VOLTS>

<OPERATOR INTERACTION STATEMENT> ::= <PRINT STATEMENT> | READ-YES |
  READ-NO | READ-NUM <VOLTS> ← <VOLTS>

<PRINT STATEMENT> ::= PRINT {<REGISTER> | "<CHARACTER STRING>"} [][]

<SAVE STATEMENT> ::= SAVE [IN] <REGISTER>

<VOLTAGE CHANGE STATEMENT> ::= 
  VOLTAGE CHANGE AT <PIN#> FROM <REGISTER> SHALL BE:
  MEASURE <VOLTS> ← <VOLTS>

<WAIT STATEMENT> ::= WAIT <TIME>

<START P-CODE STATEMENT> ::= $START-P

<STOP P-CODE STATEMENT> ::= $STOP-P

<MONITOR STATEMENT> ::= 
  MONITOR [A] TRANSITION AT <PIN#> FROM <VOLTS> ← <VOLTS>
  TO <VOLTS> ← <VOLTS> AT TIME <TIME> ± <TIME> ]
  MONITOR [THE] VOLTAGE AT <PIN#>
  OF <VOLTS> ← <VOLTS> FOR <TIME>

<STEP SIZE> ::= <REAL NUMBER>

<TIME> ::= <REAL NUMBER> <SECONDS>

<VOLTS> ::= <REAL NUMBER> <POTENTIAL>
<ANGLE> ::= <REAL NUMBER> <DEGREES>
<PIN #>  ::= <PIN PREFIX> <CHARACTER STRING>
<REGISTER> ::= R <REGISTER NUMBER>
<REAL NUMBER> ::= fortran-compatible integer or real number
<SECONDS> ::= SECONDS | SECOND | SEC | S
<POTENTIAL> ::= VOLTS | VOLT | V
<DEGREES> ::= DEGREES | DEGREE | DEG | D
<PIN PREFIX> ::= character other than digit, +, or -
<REGISTER NUMBER> ::= 0 | 1 | 2 ... | 15
APPENDIX 2 — GENERATED P-CODE

This section describes the P-code generated from each high-level ATP statement. Note that only a single 'OPEN' or 'CLOSE <switch address>' is used in each example even though multiple switch closures would be generated if defined in the hardware definition file.

1) ADJUST SIGNAL AT <PIN #1> IN A <SIGN> DIRECTION BY <STEP SIZE> PER <TIME> SECONDS UNTIL THE VOLTAGE AT <PIN #2> CHANGES TO __V +— __V

when <SIGN> is POSITIVE

ADJDEV DC?, <PIN #2 A/D ADDR>, <GAIN SPEC> <CONDITIONING SPEC>, <VALUE>, <TOLERANCE>, <STEP SIZE>, <MAX LIMIT VALUE>, <STEP TIME>

when <SIGN> is NEGATIVE

ADJDEV DC?, <PIN #2 A/D ADDR>, <GAIN SPEC>, <COND SPEC>, <VALUE>, <TOLERANCE>, <NEGATIVE STEP SIZE>, <MIN LIMIT VALUE>, <STEP TIME>

2) ADJUST SIGNAL AT <PIN #1> BY <STEP SIZE> PER <TIME> SECONDS UNTIL THE VOLTAGE AT <PIN #2> IS __V +— __V

ADJDEV DC?, <PIN #2 A/D ADDR>, <GAIN SPEC>, <COND SPEC>, <VALUE>, <TOLERANCE>, <STEP SIZE>, <LIMIT VALUE>, <STEP TIME>

3) ADJUST SYNCHRO TRANSMITTER BY <STEP SIZE> PER <TIME> SECONDS UNTIL THE VOLTAGE AT <PIN #1> IS __V +— __V

ADJDEV DS?, <PIN #1 A/D ADDRESS>, <GAIN SPEC>, <COND SPEC>, <VALUE>, <TOLERANCE>, <STEP SIZE>, <LIMIT VALUE>, <STEP TIME>
4) SET THE SYNCHRO TRANSMITTER TO ___ DEGREES ← ___ DEGREES CLOCKWISE

SETDEV DS?, <VALUE>, <TOLERANCE>

SET THE SYNCHRO TRANSMITTER TO ___ DEGREES ← ___ DEGREES COUNTERCLOCKWISE

SET DS?C, <VALUE>, <TOLERANCE>

5) CONNECT <PIN #> TO <PIN #>
CLOSE <SWITCH ADDRESS>

6) CONNECT <PIN #> TO ___V

SETDEV DC, <VALUE>, <DEFAULT TOLERANCE>
CLOSE <SWITCH ADDRESS>
CHECK <PIN # A/D ADDRESS>, <GAIN>, <COND>

7) CONNECT <PIN #> TO A SIGNAL SET TO ___V

SETDEV DC, <VALUE>, <DEFAULT TOLERANCE>
CLOSE <SWITCH ADDRESS>
CHECK <PIN # A/D ADDRESS>, <GAIN>, <COND>

8) CONNECT <S1|S2|S3> TO <PIN #>
CLOSE <SWITCH ADDRESS>

9) CONNECT ___V, ___Hz [, ___DEGREES] TO <PIN #>

SETDEV AC, <VALUE>, <TOLERANCE>
CLOSE <SWITCH ADDRESS>
CHECK <PIN # A/D ADDRESS>, <GAIN>, <COND>
10) DISCONNECT <PIN #> FROM _V
    OPEN <SWITCH ADDRESS>

11) DISCONNECT <PIN #> FROM <PIN #>
    OPEN <SWITCH ADDRESS>

12) DISCONNECT <PIN #> FROM SYNCHRO TRANSMITTER
    OPEN <SWITCH ADDRESS>

13) DISCONNECT SIGNAL FROM <PIN #>
    OPEN <SWITCH ADDRESS>

14) OPEN CIRCUIT <PIN #>
    OPEN <SWITCH ADDRESS>

15) REMOVE SIGNAL BETWEEN <PIN #> AND <PIN #>
    OPEN <SWITCH ADDRESS (USE ONE OF THE PINS)>

16) REMOVE CONNECTION BETWEEN <PIN #> AND <PIN #>
    OPEN <SWITCH ADDRESS (USE ONE OF THE PINS)>

17) REMOVE _V _HZ [__ DEGREES] BETWEEN <PIN #> AND <PIN #>
    OPEN <SWITCH ADDRESS (USE ONE OF THE PINS)>

18) REMOVE VOLTAGE FROM <PIN #>
    OPEN <SWITCH ADDRESS>
19) REMOVE <PIN #> FROM <PIN #>
OPEN <SWITCH ADDRESS>

20) REMOVE <PIN #> FROM SIGNAL
OPEN <SWITCH ADDRESS>

21) REMOVE <PIN #> FROM _V
OPEN <SWITCH ADDRESS>

22) [THE] VOLTAGE AT THE FOLLOWING POINTS SHALL BE:
{
<PIN # i> MEASURE _V +- _V
MEAS <PIN # i A/D ADDRESS>, <GAIN SPEC>, <COND SPEC>, <VOLTAGE>,
<TOLERANCE>

MEAS <PIN # n A/D ADDRESS>, <GAIN SPEC>, <COND SPEC>, <VOLTAGE>,
<TOLERANCE>

23) [THE] VOLTAGE AT <PIN #> SHALL BE: MEASURE _V +- _V
MEAS <PIN # A/D ADDRESS>, <GAIN SPEC>, <COND SPEC>, <VOLTAGE>,
<TOLERANCE>

24) [THE] VOLTAGE AT <PIN #> SHALL BE: MEASURE _V +- _V WITHIN A
PERIOD OF:
CHECK _S MAX
TRANS <PIN # A/D ADDRESS>, <GAIN SPEC>, <COND SPEC>, 0, ← 0,
<VOLTAGE>,
<TOLERANCE>, 0, ← _S
25) [THE] VOLTAGE AT <PIN#> SHALL BE: MEASURE __V +/- __V FOR A PERIOD OF:

CHECK __ S MIN

MONV <PIN # A/D ADDRESS>, <GAIN SPEC>, <COND SPEC>, VOLTAGE>, <TOLERANCE>, __S

26) [THE] VOLTAGE BETWEEN <PIN #1> AND <PIN #2> SHALL BE: MEASURE __V +/- __V

MEAS <PIN #1 A/D ADDRESS>, <GAIN SPEC>, <COND SPEC>, __V, +/- __V

27) PRINT RO "STRING" R1

LDA RO
OUTDEC
OUTSTR "STRING"
LDA R1
OUTDEC

28) READ-YES

GETSTR
CMPSTR "Y"
BNE SP15

29) READ-NO

GETSTR
CMPSTR "N"
BNE SP15

30) READ-NUM __V +/- __V

GETDEC
SUB <VOLTAGE>
ABS
CMP <TOLERANCE>
BGT SP14
31) SAVE [IN] R#  
   LDA R11  
   STA R#  

32) VOLTAGE CHANGE AT <PIN #> FROM R# SHALL BE: MEASURE _V +/- _V  
   CHANGE R#, <PIN # A/D ADDRESS>, <GAIN SPEC>, <COND SPEC>,  
   <VOLTAGE>, <TOLERANCE>  

33) WAIT ___ SECONDS  
   WAIT <TIME>  

34) $START-P  
   (NO GENERATED P-CODE)  

35) $STOP-P  
   (NO GENERATED P-CODE)  

36) MONITOR[A] TRANSITION AT <PIN #> FROM _V1 +/- _V1 TO _V2 +/-  
   _V2 AT  
   TIME _S +/- _S  
   TRANS <PIN # A/D ADDRESS>, <GAIN SPEC>, <COND SPEC>, <V1 VALUE>,  
   <V1 TOLERANCE>, <V2 VALUE>, <V2 TOLERANCE>, <TIME>, <TIME TOLERANCE>  

37) MONITOR THE VOLTAGE AT <PIN #> OF _V +/- _V FOR _S  
   MONV <A/D ADDRESS>, <COND SPEC>, <VOLTAGE>, <TOLERANCE>, <TIME>
APPENDIX 3 - ERROR MESSAGES

Different phases of the ATP compiler and its companion assembler generate error messages indicating that something in the input source file has been encountered which does not conform to a legal language construct (syntax errors) or does not constitute a valid use of the language with the given Hardware Definition Table (semantic errors). This section of documentation gives a complete listing of the error messages which may be encountered while using the ATP compiler and assembler and a brief explanation of what will cause such an error to occur.

A.3.1 COMPILER ERRORS

1) UNMATCHED QUOTE MARKS BEGINNING ON LINE NUMBER

This error indicates that either the character string on the given line number has not been properly enclosed in quote marks or that the string exceeds the 80 character maximum size limit.

2) ILLEGAL UNITS ON LINE ** TOKEN =

This error message is generated when the unit token following a tolerance is not one of the four legal unit token types (VOLTS, SECONDS, DEGREES, HZ) or a legal unit token alias. This message may also be generated when the unit tokens of a value and its tolerance are not of the same type.

3) ERROR IN NUMBER ON LINE ** TOKEN =

This error indicates that the number token found on the stated line number does not constitute a legal number. This error may also be generated if there is a blank between the sign and the number.

4) ERROR COMMENT TOO LONG OR THE PARENTHESIS ARE NOT MATCHED ***
SEARCH BEGAN ON LINE# AND ENDED ON LINE #

This error indicates that a closing right parenthesis was not found but, was searched for within the range of the given line numbers.
5) ERROR ON LINE # TOO MANY END STATEMENTS FOR THE NUMBER BEGINS

This error message will be printed when more END statements are encountered than BEGIN statements. This error is most likely caused by an extra END statement in the source file.

6) ERROR ON LINE # TOO MANY DIGITS TO THE RIGHT OF THE DECIMAL POINT

When this message is presented it indicates that more than three digits to the right of the decimal point have been encountered. Three decimal digits is the maximum number allowed.

7) ERROR ON LINE # TOO MANY DIGITS TO THE LEFT OF THE DECIMAL POINT

When this error message is printed it indicates that a number has been encountered that is out of the range of -32767 to +32767.

8) SYNTAX ERROR ON LINE NUMBER OCCURRED WHILE PROCESSING TOKEN

A syntax error of this type will occur when the high-level statement being processed does not follow the correct syntax. Check the BNF of the statement to determine what is wrong.

9) INVALID PIN-1 ID REFERENCE ON LINE #

The first specified pin in an ATP statement could not be found in the hardware definition table.

10) INVALID PIN-2 ID REFERENCE ON LINE #

The second specified pin in an ATP statement could not be found in the hardware definition table.

11) UNABLE TO SET DAC TO REQUESTED VALUE ON LINE #

A DAC of the type and value requested is not available for connection to the requested pin.

12) CANNOT MEASURE DESIRED VALUE WITH A/D ON LINE #

The requested measurement is out of range of the A/D after scaling was applied.

13) UNABLE TO SET UP D/S TO DESIRED VALUE ON LINE #

The D/S cannot be set to the requested value.
14) USER MUST SUPPLY TIME TOLERANCE ON LINE #___

A tolerance is not optional for time specifications.

15) INVALID PARSE TABLE ENTRY DETECTED ON LINE #___

The integrity of the parse tables is suspect. This error should never occur.

16) INVALID H/W DEFINITION DATA ON LINE #___

The H/W definition table entry for the requested pin is incomplete or invalid.

17) NO MONITOR SPEC. IN H/W DEFINITION ON LINE #___

It is impossible to complete the requested measurement on the currently active pin. The H/W definition table does not contain a monitor entry.

A.3.2 ASSEMBLER ERROR MESSAGES

1) ****ASSR - INVALID FIRST SYMBOL

This message indicates that a P-code statement began with an invalid or unrecognizable beginning symbol.

2) ****ASSR - INVALID LABEL REFERENCE

This indicates that the object of a branch P-code operator was never declared within the subsection containing the branch operator. Check that the label is actually declared within the same subsection. In the case of a reference to a standard procedure name, check that it is a valid name (SPO TO SP14).

3) ****ASSR - INVALID HEX DIGIT

This indicates that a A/D address or switch identifies contained something other than a valid hex digit ( '0' to '9', 'A' through 'F').

4) ****ASSR - TOO MANY LABEL REFERENCES

More than 10 forward references are made to the same label within the current subsection.

5) ****ASSR - MORE THAN 10 LABELS

More than 10 forward different label declarations were specified in the current subsection.
6) ****ASSR - DUPLICATE LABELS

Two or more declarations of the same label were specified in the current subsection.

7) ****ASSR - NOT A LITERAL STRING

The P-code parameter specified was not a literal string. A literal string should be a string of alphanumeric characters enclosed within single or double quotes. (e.g., "XYZ123 456")

8) ****ASSR - NOT A VALUE PARAMETER

The P-code parameter specified was not a value parameter. A value parameter should be a single real number (e.g., -14.253)

9) ****ASSR - NOT A TOLERANCE PARAMETER

The P-code parameter specified was not a tolerance parameter. A tolerance parameter should be a single real number with a '+' or '-' prefix with no embedded blanks (e.g., +-1.234).

10) ****ASSR - NOT A GAIN PARAMETER

The P-code parameter specified was not a gain parameter. The gain parameters are G0 through G3.

11) ****ASSR - NOT A CONDITIONING PARAMETER

The P-code parameter specified was not a conditioning factor index. The conditioning parameters are CO through C16 and should have been declared in the hardware definition file.

12) ****ASSR - NOT A REGISTER PARAMETER

The P-code parameter specified was not a register reference. The valid registers are R0 through R15.

13) ****ASSR - NOT AN A/D REFERENCE

The P-code parameter specified was not an A/D reference. An A/D reference should be a 6 digit hex number preceded by an '%' (e.g., %0123AB)

14) ****ASSR - NOT A DEVICE REFERENCE

The P-code parameter specified was not a device reference. A device reference should be a 'DS', 'DC', or 'AC' followed optionally by a 'C' followed by the device number (e.g. DSO, AC1, ACC1, DC255)
15) ****ASSR - NOT A SWITCH REFERENCE

The P-code parameter specified was not a switch reference. A switch reference should be a 'S' followed with up to four hex digits (e.g., SABC, S123, SOF1)

16) ****ASSR - INVALID REGISTER REFERENCE

The P-code parameter specified was an invalid register reference. The register references are R0 through R15.

17) ****ASSR - INVALID VALUE OR TOLERANCE

An invalid real number was specified. Note that embedded blanks are not allowed within a real number and its sign.

18) ****ASSR - INVALID SWITCH NUMBER

More than four hex digits were specified for a switch reference.

19) ****ASSR - INVALID GAIN NUMBER

A gain was specified outside of the range G0 to G3.

20) ****ASSR - INVALID CONDITIONING NUMBER

A conditioning factor index was specified outside of the range from C0 to C16.

21) ****ASSR - INVALID ADC NUMBER

More than six hex digits were specified for an A/D reference.

22) ****ASSR - LITERAL OVER 255 CHARS

More than 255 characters were specified within a literal string.

23) ****ASSR - SYMBOL EXCEEDS 10 CHARACTERS

A symbol (other than a literal string) was detected which exceeded 10 characters.

24) ****ASSR - COMMENT EXCEEDS 500 CHARS

A comment was detected that exceeded the 500 character maximum.

25) ****ASSR - COMMENT NOT TERMINATED

An end of file or I/O error occurred while processing a comment.
26) **ASSR - ERROR READING P-CODE FILE

An error occurred from a FORTRAN read statement while accessing the source P-code file.

27) **ASSR - SUBSECTION TOO LARGE

The size of the object code for the current subsection exceeded 600 bytes. Split the subsection into two subsections.

28) **ASSR - INVALID P-CODE PARAMETER

The specified P-code parameter could not be identified as any known symbol.

A.3.3 ERROR MESSAGES FROM HARDWARE DEFINITION SOURCE FILE CONVERSION

The following error messages can occur during the conversion of the hardware definition file. Processing of the file will continue unless otherwise indicated; however, any error message will necessitate the process of conversion until no errors are generated.

1) **ERROR - INVALID RECORD TYPE
   - HEADER RECORD SKIPPED

This error indicates that one of the identifier lines in the file had an invalid type. The valid types are DAC, GAIN, COND, ANGL, VOLT, PIN, or a blank line (allowed for readability). Any other abbreviations or the correct types not occurring in columns 1-4 will cause this error.

2) **ERROR - NO PIN SECTION FOUND

This error indicates that no pin information was found in the hardware definition source file.

3) ***DC DAC ENTRIES OUT OF ORDER

This error occurs if the DC DACS are not numbered sequentially in increments of the beginning at zero.

4) ***AC DAC ENTRIES OUT OF ORDER

This error occurs if the AC DACs are not numbered sequentially in increments of one beginning at zero.
5) ****INVALID AC PHASE

This error occurs when an AC phase is specified as something other than 0 or 9.5. The phase will be set to 0 should this occur.

6) ****INVALID ENTRY IN DAC TABLES

This error occurs if something other than "DC" or "AC" is in columns 1-2. (Note: Blank lines are allowed for readability so they will not cause this error.)

7) ****ERROR - GAIN ENTRIES OUT OF ORDER

This error occurs if the gains are not numbered sequentially from zero in increments of one.

8) ****ERROR - GAIN RANGES NOT IN DESCENDING ORDER

This error occurs if the gain ranges are not in descending order; i.e., the largest range value must correspond to the smallest gain spec.

9) ****ERROR - CONDITIONING ENTRIES OUT OF ORDER

This error occurs if the conditioning specs are not numbered sequentially from zero in increments of one.

10) ****INVALID VOLTAGE TOLERANCE

This error occurs if the low voltage value is greater than or equal to the high voltage value specified in a line of the voltage tolerance segment.

11) ****INVALID VOLTAGE TOLERANCE RECORD

This error occurs if "DC" or "AC" is not in columns 1-2 in the voltage tolerance segment lines. (Note: Blank lines are allowed.)

12) ****OVER 50 CONNECTION

Indicates that there were more than fifty connections for the specified pin.

13) ****ERROR IN READING INPUT FILE

A FORTRAN error condition occurred while reading the specified pin.
14) ****TOO MANY PINS IN H/W DEFINITION

More than 500 pins were specified. Processing stopped.

15) ****ERROR IN WRITING TO CONNECTION FILE

A FORTRAN error occurred on the write to the random access connection file for the specified pin.

16) ****NOT A VALID HEX DIGIT

This error indicates something other than a valid hex digit was indicated for the CRUPTR, STATE, or CHANNEL for the specified pin.

17) ****ERROR IN WRITING TO PIN ID FILE

A FORTRAN error occurred while writing to the random access pin ID file.

18) ****WRITE ERROR IN WRITING OUT FILE - .D3

A FORTRAN error occurred while trying to write to HW747.D3 (the file which contains the pins, pointers, and DAC information for the common blocks).
APPENDIX 4 - P-CODE OBJECT FILE FORMAT

A P-code Object File (hereinafter PCOF) is a series of ASCII characters, in coded lines that is to (eventually) reside on a TI digital cassette. A PCOF has a logical and a physical structure.

A4.1 PHYSICAL STRUCTURE

A PCOF is made up of a series of lines. The beginning of the file is the beginning of medium, and is known as the beginning of information (BOI). The end of file (EOF) is an EOF line. Within the file, a PCOF is also divided into groups. A group is terminated by an end-of-group (EOG) line.

A line is just a series of ASCII characters terminated by a carriage return character (CR). Because of TI limitations, a line may be at most 86 characters long, including the CR. There are two basic types of lines, the coded line and the separator line.

The separator line is just a separator, followed by a CR. For example, the end-of-group (EOG) line is the group separator (ASCII character GS), followed by the CR. The end-of-file (EOF) line is the file separator (ASCII character FS), followed by the CR.

A coded line is just 1 to 85 printable ASCII characters (codes 20H to 7EH) followed by a CR.

A4.2 LOGICAL STRUCTURE

A PCOF is logically divided into sections, with each section divided into subsections. It is headed by an identification group.
A4.2.1 FIRST GROUP

The first group on a PCOF is the PCOF identification group. This group is a series of lines, as described below.

<table>
<thead>
<tr>
<th>Line nr.</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>date &amp; time of compilation</td>
<td>YY/MM/DD.HH.MM.SS</td>
</tr>
<tr>
<td>2</td>
<td>date &amp; time of assembly</td>
<td>YY/MM/DD.HH.MM.SS</td>
</tr>
<tr>
<td>3</td>
<td>test specification number</td>
<td>AAAAAAAA</td>
</tr>
<tr>
<td>4</td>
<td>variants, separated by commas,</td>
<td>NN,NN,...,NN.</td>
</tr>
<tr>
<td></td>
<td>ended with a period</td>
<td></td>
</tr>
</tbody>
</table>

A4.2.2 SECTIONS

A section begins with a beginning of section line (BOSL) and is the first line of a group. It ends with the EOG line which is before the next section, or it ends with the EOF.

FORMAT for the BOSL:

.SSSSSSSSSSSSSSSSSSSSSSS

A4.2.3 SUBSECTIONS

A subsection usually begins with the first line of a group, and always terminates with an EOG line. The only time a subsection identification line (BOSSL) is not the first line in a group is when the subsection is the first subsection in a section, in which case the BOSL preceeds the BOSSL.

The format for BOSSL is:
identifies this line as a BOSSL

VV -variant identifier. A blank means that this subsection applies to all variants.

SSS...S -subsection number

After the BOSSL, the object code for this subsection follows, organized into Object Code Lines. (OCL)

The format for an OCL is:

:NNAAAARRBBBBBB...BBBBCC

: -identifies this as an OCL
NN -number of data bytes on this line
AAAA -address of first data byte
RR -record type
  00 - absolute address record
BB...BB -each two characters is a data byte
CC -checksum

The 8 bit truncated sum of all ASCII-encoded-bytes on this line should be zero

Note that each "number" takes two characters and is an ASCII-encoded-hex representation of a byte. The address AAAA is just two bytes.