University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) Department of Computer & Information Science

May 1982

NOPAL Reference Manual: Bottom-Part

Noah S. Prywes
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation

Noah S. Prywes, "NOPAL Reference Manual: Bottom-Part", . May 1982.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-82-145.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/745
For more information, please contact repository@pobox.upenn.edu.

NOPAL Reference Manual: Bottom-Part

Abstract

Nopal is a descriptive non-procedural very high level language for writing specifications of programs for test of analog and digital electronic circuits by Automatic Test Systems (ATS). It can also be used for writing specifications of general purpose computation. Based on a NOPAL specification, the NOPAL processor generates a computer program in the ATLAS test language. A specification in NOPAL may consist of a number of modules,one of which is the main module. The main module consists of a collection of specifications of individual tests given non-procedurally in an arbiitrary order. A test in NOPAL corresponds to the notion of a physical test a unit under test (UUT). A test consists of stimuli (signals) to be applied measurements to be taken and logic for selecting diagnoses depending on the passing or failing of one or more tests. Information about the UUT and automatic test equipment (ATE) may also be included in the main module. The UUT and ATE information is used for conducting various consistency checks and is included in the produced documentation. A module, other than the main module, specify functions which apply stimuli, make measurements or evaluate variables. Interfaces among modules are provided by referencing in one module function specified in another module. The NOPAL processor analyzes the specification of module for consistency, completeness and non-ambiguity and generate error/warning messages and a number of reports which serve as the documentation for the specification. If the specification is error free, the NOPAL processor orders the program events to attain efficiency in computer time and in use of memory. Finally, it generates a program in the EQUATE-ATLAS Test Programing Language. ATLAS programs, generated from modules of a specification, are submitted together for compilation and executed on RCA EQUATE AN-USM-410 computer controlled ATE.

Keywords

automatic testing, atlas, analog testing, digital testing, nonprocedural languages, very high level languages

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-82-145.

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/745

UNIVERSITY OF PENNSYLVANIA
PHILADELPHIA 19104
Automatic Program Generation Project
The Moore School of Electrical Engineering/D2
Department of Computer and Information Science

NOPAL REFERENCE MANUAL

BOTTOM-PART

May 1982

by

Noah S. Prywes

Prepared with Support From:
U.S. Army Communications and Electronics Command
ATE and Computer Software Support Branch
Code DRSEL-ME-SC
Fort Monmouth, New Jersey 07703
Under Contract DAAB07-81-F-1598

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* NOPAL Reference Manual: Bottom-Part		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER Moore School Report
7. AUTHOR(s) Noah S. Prywes		8. CONTRACT OR GRANT NUMBER(s) DAAB07-81-F-1598
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Pennsylvania, Moore School of Electrical Engineering, Department of Computer Science, Philadelphia, PA 19104		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Communications and Electronics Command, ATE & Computer Softward Support Branch, Fort Monmouth, NJ 07703		12. REPORT DATE May 1982
		13. NUMBER OF PAGES 160
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

General Distribution

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

NOPAL; AUTOMATIC TESTING; ATLAS; ANALOG TESTING; DIGITAL TESTING: NONPROCEDURAL LANGUAGES; VERY HIGH LEVEL LANGUAGES

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Nopal is a descriptive non-procedural very high level language for writing specifications of programs for test of analog and digital electronic circuits by Automatic Test Systems (ATS). It can also be used for writing specifications of general purpose computation. Based on a NOPAL specification, the NOPAL processor generates a computer program in the ATLAS test language. A specification in NOPAL may consist of a number of modules,one of which is the main module. The main module consists of a collection of specifications of individual tests given non-procedurally, in an

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

arbitrary order. A test in NOPAL corresponds to the notion of a physical test on a unit under test (UUT). A test consists of stimuli (signals) to be applied, measurements to be taken and logic for selecting diagnoses depending on the passing or failing of one or more tests. Information about the UUT and automatic test equipment (ATE) may also be included in the main module. The UUT and the ATE information is used for conducting various consistency checks and is included in the produced documentation. A module, other than the main module, may specify functions which apply stimuli, make measurements or evaluate variables. Interfaces among modules are provided by referencing in one module a function specified in another module. The NOPAL processor analyzes the specification of a module for consistency, completeness and non-ambiguity and generates error/warning messages and a number of reports which serve as the documentation for the specification. If the specification is error free, the NOPAL processor orders the program events to attain efficienty in computer time and in use of memory. Finally, it generates a program in the EQUATE-ATLAS Test Programming Language. ATLAS programs, generated from modules of a specification, are submitted together for compilation and executed on RCA EQUATE AN-USM-410 computer controlled ATE.

ACKNOWLEDGEMENTS

The writing of this manual was part of a major revison of the bottom-part of the NOPAL system. In particular the revision included the addition of the following capabilities.

1. Data Type definition, propagation and checking.

2. Definition of array type (repeating) tests and their efficient scheduling.

3. Definition of stimuli and measurement functions.

4. Use of Digital stimuli, measurements and digital operations.

The staff engaged in the revisions and additions has contributed to this manual as well. They consisted of:

Maya Gokhale
Chi-Ming Chen
Yuan Shi
Kwang-Shi Shu.

TABLE OF CONTENTS

TABLE OF FIGURES 4

TABLE OF TABLES. 6

CHAPTER 1 OVERVIEW OF THE NOPAL LANGUAGE AND PROCESSOR . 1

1.1 INTRODUCTION 1
1.2 ADVANTAGES OF NOPAL 2
1.3 PROCEDURE FOR USING THE NOPAL SYSTEM 4
1.4 DIFFERENCE IN APPROACH: NOPAL VS. ATLAS 6
1.5 USE OF SYNTACTIC DIAGRAMS 8
1.6 LIMITATIONS IN NOPAL 9
1.7 ORGANIZATION OF THE MANUAL 9

CHAPTER 2 COMMONLY USED NOPAL LANGUAGE ELEMENTS 11

2.1 NOPAL CHARACTER SET 11
2.2 CONSTANTS 12
2.2.1 String Constants .12
2.2.2 Character String Constant 13
2.2.3 Number Constants .13
2.3 VARIABLES,CONNECTIONS,TESTS AND DIAGNOSES 13
2.3.1 FUNCTIONS 16
2.4 KEYWORDS AND LABELS 19
2.5 UNITS 21
2.6 ARITHMETIC EXPRESSIONS 23
2.7 DIGITAL EXPRESSIONS 24
2.8 IF-CLAUSE AND RELATIONAL EXPRESSION 26

CHAPTER 3 MAJOR COMPONENTS OF A SPECIFICATION MODULE . .28

CHAPTER 4 TEST SPECIFICATION 31

4.1 STIMULI, MEASUREMENT OR EVALUATION STATEMENTS 33
4.1.1 Conjunctions 34
4.1.2 Assertions 37
4.1.3 Free Subscripts And Array References 39
4.2 SPECIFICATION OF DIAGNOSES SELECTION LOGIC 44
4.3 SPECIFICATION OF A DIAGNOSIS 48
4.4 SPECIFICATION OF MESSAGES 51

CHAPTER 5 UUT RELATED DECLARATIONS 55

5.1 DIAGNOSABLE COMPONENT FAILURE STATEMENTS 55
5.2 UUT CONNECTING POINT STATEMENTS 57

CHAPTER 6 ATE RELATED DECLARATION STATEMENTS 61

6.1 FUNCTION DECLARATION STATEMENTS 61
6.2 ATE CONNECTION POINT DECLARATION STATEMENT 66

CHAPTER 7 DATA DECLARATION STATEMENTS 69

7.1 ELEMENTARY VARIABLES WITH PRIMITIVE DATA TYPES 69
7.2 DATA STRUCTURES AND HIGH LEVEL DATA TYPES 71

CHAPTER 8 DEFINITION OF FUNCTIONS AND CONNECTIONS FILE .75

8.1 DEFINITION OF FUNCTIONS 75
8.1.1 Definition Of A Function In A Secondary NOPAL Module. 77
8.1.2 Definition Of Functions Directly In ATLAS. 79
8.2 UUT-ATE CONNECTION FILE 79

CHAPTER 9 OPERATING THE NOPAL PROCESSOR 83

APPENDIX I TEMPLATES USED TO COMPOSE REQUIRE CLAUSES TO DEFINE STIMULI AND MEASUREMENT FUNCTIONS . . .84

I.1 ORGANIZATION OF EACH TEMPLATE ENTRY. 84
I.2 NOTATIONS USED IN TEMPLATES. 84
I.3 TEMPLATES ORDERED BY CATEGORIES. 84
I.3.I Category I-Dimension 84
I.3.II Category II-Connection Points Descriptions 85
I.3.III Category III-Digital Test. 85
I.3.IV Category IV-Analog Stimulus. 86
I.3.V Category V-Analog Measurements 87
I.3.VI Category VI-RF Stimulus and Measurements 88
I.3.VII Category VII-Miscellaneous 88

APPENDIX II EXAMPLES 90

II.1 FREQUENCY SPECTRUM ANALYSIS EXAMPLE. 90
II.2 DIGITAL COUNTER EXAMPLE. 96

APPENDIX III ERROR AND WARNING MESSAGES IN THE NOPAL SYSTEM 138

III.1 INDEX TABLE 138
III.2 ERROR AND WARNING MESSAGES. 138

TABLE OF FIGURES

Figure 1.1 The Procedure of Use of NOPAL. 5
Figure 1.2 Syntax of a NOPAL Specification. 9

Figure 2.1 Character Set Used in NOPAL. 12
Figure 2.2 Syntax of a String Constant. 12
Figure 2.3 Syntax of a Character String 13
Figure 2.4 EBNF Specification of Number 13
Figure 2.5 Syntax Diagram of Identifiers. 14
Figure 2.6 Syntax Diagram of Variables, Connecting Points, Tests or Diagnoses 14
Figure 2.7 Syntax Diagram for Functions 16
Figure 2.8 Syntax Diagram for Labels. 21
Figure 2.9 Syntax of Arithmetic Expression. 23
Figure 2.10 Syntax of Digital Expressions. 25
Figure 2.11 EBNF Specification of IF-CLAUSE. 26

Figure 3.1 Syntax of Header Statement for Modules and their Major Subparts: TESTs, MODFUNs, STIMULI and MEASUREMENTS 29
Figure 3.2 Syntax of Statement Headers. 30

Figure 4.1 Syntax of Conjunctions 34
Figure 4.2 Syntax of Assertions 37
Figure 4.3 Syntax of Subscript Definition 40
Figure 4.4 Syntax of LOGIC Statement. 45
Figure 4.5 Syntax of Diagnosis Statements 49
Figure 4.6 Syntax of Message Statement. 52

Figure 5.1 Syntax of UUT Component Failure Statement. . . .56
Figure 5.2 Syntax of UUT Connection Point Declaration Statement. 58

Figure 6.1 Syntax of Function Declaration Statement 62
Figure 6.2 Syntax of ATE Point Declaration. 68

Figure 7.1 Syntax of Data Declaration Statement for Elementary Variables 70
Figure 7.2 Syntax Diagram of Data Structure Declaration Statements .71

Figure 8.1 Syntax Diagram of a Function Header Statement. 76
Figure 8.2 Syntax of ATLAS DEFINE Statement for UUT-ATE Connection 78

Figure II.1 Test Specification SP001 for Example 1 93
Figure II.2 Reformatted Report for Example 1 97
Figure II.3 Cross Reference and Attributes Report for Example 1. 101

Figure II.4 Cross References—Tests vs Diagnoses vs Messages vs Test Points vs Functions—for Example 1. 103
Figure II.5 Flowchart Report for NOPAL Specification SP001. 104
Figure II.6 Sequence Report for Example 1. 105
Figure II.7 Error and Warning Report for Example 1 112
Figure II.8 ATLAS Program for Example 1. 113
Figure II.9 Test Specification Digital_Test for Example 2. 120
Figure II.10 Reformatted Report for Example 2 122
Figure II.11 Cross Reference and Attributes Report for Example 2. 125
Figure II.12 Cross References—Tests vs Diagnoses vs Messages vs Test Points vs Functions—For Example 2. 126
Figure II.13 Flowchart Report for NOPAL Specification Digital_Test 127
Figure II.14 Sequence Report for Example 2. 128
Figure II.15 Error and Warning Report for Example 2 132
Figure II.16 ATLAS Program for Example 2. 133

TABLE OF TABLES

Table 2.1 Built-in ATLAS Evaluation Functions. 17
Table 2.2 Built-in NOPAL Functions 18
Table 2.3 Keywords 20
Table 2.4 Units used in NOPAL for EQUATE-ATE 22

Table 9.1 Options in Running the NOPAL Processor 81
Table 9.2 Files Used by the NOPAL Processor. 83

CHAPTER 1

OVERVIEW OF THE NOPAL LANGUAGE AND PROCESSOR

1.1 INTRODUCTION

NOPAL is a non-procedural test programming language. By non-procedural we mean two things. First, the language is descriptive, as opposed to procedural languages which are prescriptive. There are no commands in NOPAL and it consists entirely of declarations and equations or other relations. Second, there is no ordering or timing of events in NOPAL as opposed to procedural languages where the order and timing are determined by the programmer. The objective of the NOPAL system is to reduce the efforts, costs and requisite expertise that are involved in preparation of programs for automatic test equipment, in particular for testing analog systems and components. Prior to focusing on the major topic of the NOPAL system, it is important to place it in the context of the role of Automatic Test Systems.

Automatic Test Systems (ATS) are perceived as an effective tool for reducing the high cost of maintenance. Approximately 70% of maintenance costs can be impacted by the use of ATS as indicated from the following estimates. Labor accounts for 80% of maintenance costs and 87% of the labor time is occupied by diagnosis and fault isolation, with the remainder 13% by the repair itself. ATS are therefore designed to attack the major maintenance cost items of diagnosis and fault isolation. ATS are applicable not only to product manufacturing and repair but also to maintenance of large systems such as in communications, computer networks, medicine, agriculture, transportation and security.

Automatic Test Systems may be viewed as consisting essentially of three main components: 1) the Automatic Test Equipment (ATE), 2) the software and 3) the Units Under Test (UUT). We will focus in this manual on the software for ATS, which consists of test programs for each individual UUT that is to be maintained. A large number of programs is needed for the great diversity of UUT's. The cost of ATS software greatly exceeds the cost of the ATE equipment itself.

The task of a programmer is to analyse the UUT, determine an appropriate set of tests based on which the diagnoses may be selected, and then prepare a program that will be used with the ATE to perform the tests and produce the appropriate diagnoses. This task is complex and laborious. The ATLAS Test Language has been developed as an aid to programming. ATLAS is a fairly low-level language which requires a great deal of user expertise in testing and in the operation of the ATE. The user of ATLAS must also devise algorithms that are efficient in use of computer time and memory.

The NOPAL system further facilitates the development of test programs. The programmer produces a test program specification in NOPAL which consists mainly of description of tests, rather than how the tests are performed by a computer. It is much shorter and simpler than the respective ATLAS program. The user of NOPAL is relieved of considerations of input/output, memory allocation and efficiency considerations. The NOPAL system conducts many checks of the correctness of the specification and informs the programmer of any problems found. This produces a more reliable program. The NOPAL system schedules program events efficiently and outputs a program in the ATLAS language ready to be used with the ATE. It also produces documentation and provides a methodology for updating the test programs throughout the life cycle of the UUT.

1.2 ADVANTAGES OF NOPAL

Ease in specifying a test program: The major part of a NOPAL specification consists of specifications of individual tests. A description of the tests and diagnoses for the UUT must be available prior to the composition of the NOPAL specification. Test descriptions are typically available in technical or maintenance manuals for the UUT or in specifications of acceptance tests. If a good description of these tests exists, the programmer needs only to restate them individually in the NOPAL language. Sometimes the derivation of tests requires deeper analysis of the UUT. Such analysis can be performed manually or by a companion system, called the Top Part of NOPAL. The latter system is based on simulation of faults in electronic circuits and is not described in this manual.

Once a description of the tests has been obtained, the programmer specifies these tests in NOPAL, one at a time, in an arbitrary order. There is no need to consider the order of execution of tests. Each test is relatively independent of the others. Further, a test is divided into relatively independent subparts. Each test contains six. subparts: stimuli to be applied to the UUT, measurements to be taken, relations to be evaluated, logic for selecting the diagnoses, and the respective interaction with the operator through messages and responses. The programmer need not be concerned about the ordering of the tests or the ordering of statements within a test. There is also no need to declare internal variables or provide input/output and iteration statements.

The selection of algorithms that are efficient in use of computer time and memory, normally performed by an ATLAS programmer, is performed automatically by the NOPAL processor.

The user of NOPAL need not be concerned with selection of ATE devices and connection points. Thus, it is not necessary that a user possesses this type of expertise. Instead, the user refers to functions defined elsewhere without requiring knowledge of the stimuli, measurements or computations that are involved. The user of NOPAL must however provide some information on the UUT and ATE which is used for two purposes - for checking consistency and for inclusion in the documentation. The UUT part includes symbolic names for the UUT connection points and a list of the UUT failures which are diagnosed by the program. The ATE part consists of declarations of stimuli, measurement and computational functions used in the specification. These functions are available from a function library or specified in separate modules.

Global checking of the specification: in addition to the analysis of the syntax and semantics of individual statements, the NOPAL processor performs global analysis to detect incompletenesses, ambiguities or inconsistencies in the overall specification. These checks assure that all the variables have been defined, that ambiguous naming of variables has not occurred, and that there is consistency in dimensionality of variables and in the use of subscripts of array variables. The NOPAL system attempts in some cases to correct the specification automatically, while producing warning messages for the user. In other cases, error messages are issued, explaining the problems discovered and soliciting corrections from the user. Since the NOPAL language is non-procedural it is much easier to perform these checks on a NOPAL specification than on a program in a procedural language such as ATLAS. The global checking in NOPAL should lead to a more reliable and complete program.

Debugging and Modification: once a specification has been completed and an ATLAS program generated, the user can run the produced program on the ATE with a UUT to check whether the program performs according to the intentions of the programmer. Initially the user may generate an ATLAS program that includes a trace of all activities. The ATLAS program with a trace includes the output of results of all measurements and tests. Another version, without a trace is generated for productive use (see Section 9). Deviations of output variables from what the user expected may be observed by running the program. Due to the nonprocedural nature of NOPAL, the order of the statements is immaterial, and a variable can have only a single value. There is no need in the debugging to find the appropriate place for the modifications or to trace the different values that are stored in a variable memory location during the execution of a program, as is the case when debugging or modifying an ATLAS program. The user must only examine the statements that define the respective variables. If an error or a change is desired, it is necessary to modify these NOPAL statements and generate a new program in ATLAS.

Documentation: the NOPAL processor produces three types of documentation which are used in debugging and modifying a specification. The first type consists of a listing of the NOPAL specification, which has been reformatted for easy readability. The individual parts of the specification are clearly identified and an indentation scheme is used to indicate the relation between parts and constituent subparts. Next, there are a variety of cross-reference reports. One cross-reference report shows the usage of all identifiers (i.e. variables, connections, stimuli and measurement functions, etc). Other reports cross-reference various components of the specifications, such as the diagnoses selected by respective tests, or the failures of the UUT detected by respective tests. The list of failures allow the user to check that the program indeed tests for all the types of failures which the program is intended to diagnose. The final type of report consists of flowcharts for each of the tests and for the overall program. This type of report is sometimes helpful in debugging, as it also shows the various dependencies between variables and statements.

Maintenance: the documentation generated when a NOPAL specification is compiled serves as a basis for maintenance. The formatted NOPAL specification provides a maintenance programmer with a concise description of the tests. The specification may be modified to meet the changing needs and submitted to the NOPAL processor. The modification may involve modifying, deleting or adding tests. A new ATLAS program and accompanying documentation are then produced. Thus, there is no need to refer to the ATLAS program either in the original development of the program or in the subsequent maintenance. The documentation produced with each updating of a test program is then retained to serve as a basis for future maintenance.

1.3 PROCEDURE FOR USING THE NOPAL SYSTEM

Figure 1.1 illustrates the procedure for using the NOPAL system.

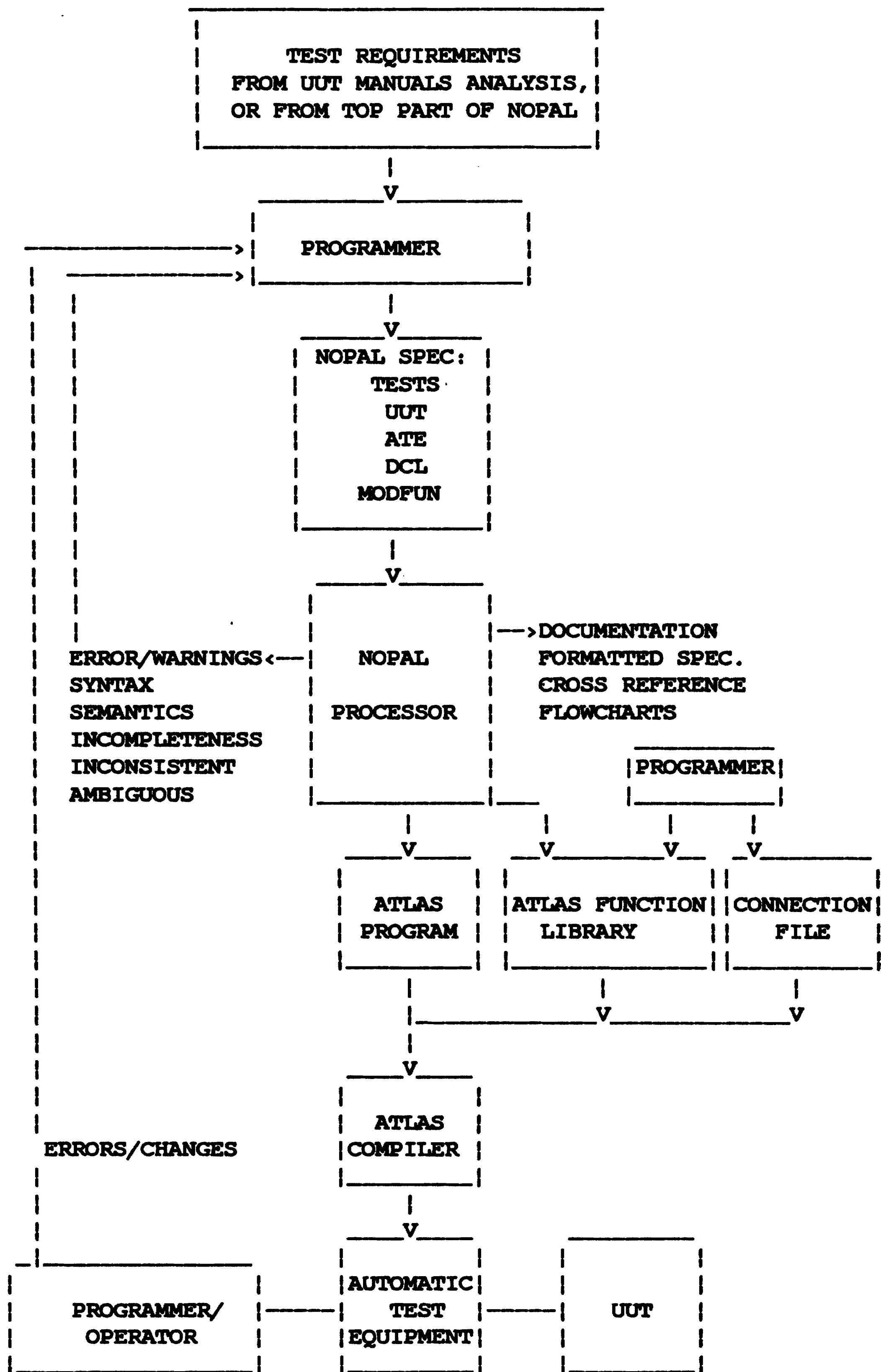

Figure 1.1 The Procedure of Use of NOPAL.

The procedure starts with obtaining test requirements from UUT manuals, from analysis or from the use of the Top Part of NOPAL. The programmer translates these test requirements into the NOPAL language, one test at a time. The UUT connection points and failure modes and the ATE stimuli and measurement functions must be declared as well. If input or output files are used, it is necessary to provide a declaration of the organization of the data in these files. Additional modules that define functions may be submitted separately. The specification is highly modular and to a large extent the above parts are independent of one another. The programmer can therefore concentrate on composing one part at a time.

Next, the specification module is submitted to the NOPAL processor. It is analyzed and messages are sent to the programmer indicating problems that have been detected. The programmer must make corrections and resubmit the specification to the NOPAL processor until no further errors are detected. Documentation is generated on each submission and may be referenced by the programmer in making the corrections. The documentation contains the formatted specification and cross-reference reports. If no errors are detected in the analysis, the processor produces flow-chart reports. An ATLAS program is finally generated. The specification may include references to functions which are in the ATLAS function library or which are defined in other modules. Simple stimuli and measurement functions can be defined in the NOPAL specification itself. Alternately, the programmer can write the function definitions directly in ATLAS. The programmer must also prepare a connection file in ATLAS that shows the physical UUT-ATE connections that are used.

The ATLAS program, the modules of the respective functions which are used and the connection file are then submitted to the ATLAS compiler, which produces an object language program to be loaded into the ATE. This program must be run in the ATE with the UUT. On the initial testing of the program, the programmer may find that the program performs differently from what was intended. This indicates that the programmer has made errors that must be corrected. The programmer must then modify the NOPAL specification and repeat the entire procedure.

The NOPAL processor itself is written in the PL/I language and runs on Digital Equipment VAX/11 computers. The ATLAS program that is generated must be moved from the VAX/11 computer to the RCA EQUATE AN/USM-410 Electronic Test Equipment for compilation and testing.

1.4 DIFFERENCE IN APPROACH: NOPAL VS. ATLAS

Using ATLAS, a programmer determines the order of program events by the order of respective statements in the program. There is no equivalent facility in NOPAL, as the NOPAL processor makes all the event-scheduling decisions. This is unfamiliar to the experienced ATLAS programmer and the question of how the NOPAL processor determines the

scheduling is frequently raised. A second difference is that in NOPAL a variable must be defined to have only a single value. This is contrasted with procedural languages where a variable may have several values, assigned at different times in the program. These two differences require an approach which is different from that experienced by users of procedural languages.

The NOPAL processor makes scheduling decisions based on recognizing precedence relationships between NOPAL statements. The precedence relationships are recognized through analysis of the respective statements. The simplest and most common type of precedence relationship is between a statement that defines a value of a variable and another statement that uses this variable. Obviously the first statement must precede the latter. There are other precedence relationships derived from the practice of testing. For instance, stimuli are normally applied prior to measurements. Another example concerns preceding one test before another in order not to damage a UUT component. We may want to test first that a certain component is not shorted (or open), before testing for another component failure, which would be damaged if the first component is shorted (or open). Finally, there are precedence relationships that do not concern the correctness of the program, but only improve the efficiency of the program. These precedence relationships are of lower priority. An example is the omission of executing a test which diagnoses a failure which has already been determined not to have failed by a previous test. Another example is grouping tests which use the same stimulus.

While control statements (calls, iterations, etc.) are provided in ATLAS explicitly by the programmer, in NOPAL they are derived automatically from the structure of the data, statements and tests that are used in a specification. NOPAL includes facilities for referring to<u>array</u> variables, statements, diagnoses and tests. These entities may have a number of instances, referred to as <u>elements</u> that are arranged in one or more dimensions. Arrays are useful to allow definition of repetitions. Each instance or element may be referenced through adding subscripts after the names of these entities. The NOPAL processor implements all the implied instances, or elements, through use of iteration loops in the produced program. In scheduling the scope of iteration loops, the NOPAL processor also considers efficient use of memory in the produced program. Namely it attempts to reuse the same memory space for all or some of the elements of an array.

As noted above, the user of NOPAL must be careful to define only one value for each variable. The experienced ATLAS programmer may be concerned that more variables must be used than in writing an ATLAS program and therefore more memory space would be required. This is not the case, as the NOPAL processor uses the same memory space for several elements of a variable. For instance, the use of iteration as in the following ATLAS statements:

```
X = 0
FOR I = 1 THRU N
  X = X + 1
END FOR
```

which are very common in use of a procedural language is not allowed in NOPAL. First there are no iteration facilities in NOPAL. Also it implies N different values for X at different times, and timing need not be expressed in NOPAL. The same notion may be expressed in NOPAL as:

```
IF I = 1 THEN X(I) = 1
         ELSE X(I) = X(I-1) + 1
```

Now X is a vector of N elements. I is a subscript denoting an index of an element in X and I-1 is the index of the next lower element of X. The NOPAL processor generates an ATLAS iteration control statement to evaluate all the elements of an array. It will use memory space for the Ith and I-1th elements of X only. This example illustrates the change of approach required of an experienced ATLAS programmer in initial use of NOPAL.

The rules for deriving the precedence relationships and how they effect the schedule are very complicated. The user need not consider at all the problem of scheduling in composing respective statements. The NOPAL processor will determine the schedule of program events, based on the user's NOPAL specification. If the specification can not be scheduled, then there are some errors where statements may be ambiguous or inconsistent. Error messages explain these problems without referring to scheduling constraints. The NOPAL processor produces reports that show the precedence relationships between the entities in the specification. It also produces a flowchart report. From these reports a user may indeed find all the scheduling considerations. However our objective is that these reports not be used, except in debugging the NOPAL processor itself.

1.5 USE OF SYNTACTIC DIAGRAMS

The syntax of NOPAL is shown in this manual in the form of syntax diagrams. These diagrams use the familiar EBNF (Extended Backus Normal Form) notation. A syntactic diagram defines a syntactic unit in terms of other syntactic units or letters of the alphabet of NOPAL.

The names of syntactic units in the syntactic diagram are enclosed in angle brackets < >.

The symbol ::= means that the syntactic unit on the left is defined as the sequence of units on the right-hand side.

The vertical bar | denotes that the syntactic unit on the right is one of several alternatives.

Syntactic units enclosed in square brackets [] are optional, i.e., they may appear zero or one time. An asterisk following the right square bracket []* means that the units enclosed in the brackets may be repeated zero or more times.

The syntax diagrams are shown indented and assigned level numbers to indicate the depth of definitions of respective units, starting with the highest level unit which is defined in terms of lower level units, and so on. For example the entire NOPAL specification in a collection of statements. This is expressed in Figure 1.2.

```
<NOPAL SPECIFICATION> ::= [<NOPAL STATEMENT>;]*
```

Figure 1.2: Syntax of a NOPAL Specification

The remainder of this report describes the various types of NOPAL statements, their syntax and semantics.

1.6 LIMITATIONS IN NOPAL

EQUATE-ATLAS has some limitations which make it difficult to implement some of the higher level features of NOPAL. It is necessary therefore also to avoid use of these features in a NOPAL specification. These limitations will be removed when an improved ATLAS compiler is used. Following are three limitations that are of particular importance. NOPAL allows using multi-dimensional array variables, while EQUATE-ATLAS allows only single-dimensional arrays. The NOPAL processor in some cases incorporates in the generated program the translation from multiple to single dimensional arrays. In other cases the user is limited to using only single dimension variables also in NOPAL. Next, EQUATE-ATLAS does not allow variables to represent character strings. This limitation must therefore be observed in NOPAL as well. Finally, EQUATE-ATLAS allows use of only global variables in a compilable module. Namely, names of variables must be unique in a compilable program. In NOPAL it is potentially possible to use local variables in individual tests. Namely, the user can use the same names for variables in different tests, provided these variables are referred to only locally in the respective tests. Due to the restrictions in EQUATE-ATLAS it is necessary to use only global variables in the entire NOPAL specification as well. These limitations are explained further at the appropriate places in the manual.

1.7 ORGANIZATION OF THE MANUAL

Section 2 presents the fundamental units of the language (i.e. constants, variables, etc.). These units are also discussed in greater detail in later sections which describe their use. The breakdown of a specification into its major parts and types of statements is described in Section 3. The specification of individual tests is described in Section 4, of the UUT in Section 5, of the ATE in Section 6, and of

input/output files in Section 7. The definition of functions in ATLAS is described in Section 8. Section 9 discusses operating by the NOPAL system. It also describes the reports produced, the control parameters used to inhibit the generation of some reports, and how to generate an ATLAS program which gives a trace of the program activity, which is useful in debugging. Four appendices at the end of the manualconsist of an example, templates for stimuli and measurement functions, and an explanation of error messages.

CHAPTER 2

COMMONLY USED NOPAL LANGUAGE ELEMENTS

This section describes the words and expressions which are the basic elements of NOPAL. The character set of NOPAL is described in Section 2.1. Constants are described in Section 2.2. The use of the character set to give symbolic names to identifiers (for variables, UUT connection points, functions, tests and diagnoses) and to labels (for statements) is described in Section 2.3 The use of functions is described in Section 2.4. Section 2.5 describes the use of keywords to identify individual parts of the NOPAL specification and respective statement types. Reserved words for specifying the units of physical quantities in stimuli and measurements are described in Section 2.6. Arithmetic expressions are described in Section 2.7, digital expressions in Section 2.8, and relational expressions in Section 2.9. These language elements are further discussed in later sections in connection with their use in respective statements.

2.1 NOPAL CHARACTER SET

The alphabet of the NOPAL language is the full ASCII character set. The character set is classified into letters, digits, and special characters. The character set is shown in Figure 2.1.

```
<LETTER> ::=  A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|
              Q|R|S|T|U|V|W|X|Y|Z|@|#|$|_
              a|b|c|d|e|f|g|h|i|j|k|l|m|n|
              o|p|q|r|s|t|u|v|w|x|y|z|
<DIGIT> ::=  0|1|2|3|4|5|6|7|8|9
<SPECIAL_CHAR> ::= .|<|(|+|&|||*|)|;|^|-|/|,|
                   %|_|>|?|:|'|=|"|BLANK
```

FIGURE 2.1: Character Set Used In NOPAL.

Lower case letters in a NOPAL specification not enclosed in quotes are translated by the NOPAL processor into corresponding upper case letters. The processor treats the NOPAL input as a contiguous string. Delimiters must be used between fundamental syntactic units. The special characters are used as delimiters. The fundamental syntactic units, such as constants and variable names, must be completed in a single input line. The user may freely indent or skip lines to improve readability of a specification. For example, the reformatted NOPAL source listing produced by the NOPAL processor, which includes an indentation scheme, may be used as input in subsequent runs.

2.2 CONSTANTS

2.2.1 String Constants

String constants must be enclosed within single quotes ('). There are three types of string constants: (1) bit string, (2) octal string, and (3) hexadecimal string. The NOPAL processor incorporates translation of the string into a representation that employs a DIGITAL data type, at most sixty-four bits long. A bit string is a sequence of no more than sixty-four 1's and 0's enclosed by single quotes, and prefixed by the character B. An octal string is a sequence of no more than 21 numerics, 0-7 enclosed by single quotes and prefixed by the letter O. A hexadecimal string is a sequence of no more than 16 numerics, 0-9 or the letters A-F enclosed by single quotes, and prefixed by the letter H. Figure 2.2 shows the syntax diagram for a string constant.

```
1  <STRING CONSTANT> ::= <BIT_STRING>   |   <OCTAL STRING>
                         |  <HEX_STRING>
     2  <BIT_STRING> ::= B'<BIT> [BIT]*'
         3 <BIT> :: = 0 | 1
     2  <OCTAL_STRING> ::= O'<OCTAL DIGIT>[<OCTAL DIGIT>]*'
         3 <OCTAL_DIGIT> ::= 0|1|2|3|4|5|6|7
     2  <HEX_STRING> ::= H'<HEX DIGIT> [<HEX DIGIT>]*'
         3 <HEX_DIGIT> ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F
```

FIGURE 2.2. Syntax Of A String Constant.

2.2.2 Character String Constant

Definition of variables by character strings is not allowed in EQUATE-ATLAS. Therefore, they may be used in NOPAL only to define text of messages and arguments of special built-in functions. The syntax of a character string is shown in Figure 2.3. A character string may contain any character from the NOPAL alphabet, enclosed by a single quote. If a single quote is a part of the string, it is represented by two consecutive quotes.

```
1 <CHAR_STRING> ::= '[<FULL_CHAR>]*'

    2 <FULL_CHAR> ::= <LETTER> | <DIGIT> | <SPECIAL_CHAR>
```

FIGURE 2.3. Syntax of a Character String.

2.2.3 Number Constants

Figure 2.4 shows the syntax diagram for numbers.

```
1 <NUMBER> ::= [<SIGN>] <UNSIGNED_NUMBER>
  2 <SIGN> ::= +|-
  2 <UNSIGNED_NUMBER> :: = <DECIMAL_NUMBER> [<EXPONENT>]
    3  <DECIMAL_NUMBER> ::=  <UNSIGNED_INTEGER>
                              [<DECIMAL_FRACTION>]
                              | <DECIMAL_FRACTION>
      4  <UNSIGNED_INTEGER> ::= <DIGIT> [<DIGIT>]*
      4  <DECIMAL_FRACTION> ::=  .<UNSIGNED_INTEGER>
    3  <EXPONENT> ::= E[<SIGN>] <DIGIT> [<DIGIT>]
```

FIGURE 2.4. EBNF Specification of Number.

A number can be either signed or unsigned integer or decimal floating point. For example, -500, 5 and +1234 are integers. +1.8, -250.01 and 1.23E+08 are floating point numbers. Sixty-four bits are allocated in EQUATE ATLAS for each number and for each decimal or integer variable. A decimal number can be represented with a precision of at most seventeen decimal digits, of which at most ten may be on either side of the decimal point. The exponent must be within the limits of -75 and +75.

2.3 VARIABLES,CONNECTIONS,TESTS AND DIAGNOSES

Identifiers are symbolic names given to variables, UUT-ATE connecting points, tests, diagnoses and functions. An identifier must begin with a letter, which may be followed by letters or digits. Note that @, #, $, and _ are considered letters. The NOPAL identifiers are also used in the generated ATLAS program, except that the $ and _ are translated into . and -, respectively. The NOPAL processor truncates

the identifier beyond the 12th character. The syntax diagram for identifiers is shown in Figure 2.5.

```
<IDENTIFIER> ::= <LETTER>[<LETTER>|<DIGIT>]*
```

FIGURE 2.5. Syntax Diagram of Identifiers.

A variable, a UUT connecting point, a test or a diagnosis may be either a <u>scalar</u>, i.e. have a single element, or a multi-dimensional <u>array</u>, i.e. with multiple elements in each dimension. It is necessary to refer in statements to specific elements of an array. This is done by adding after the name of the array a list of subscript expressions for the dimensions of the array, enclosed in parentheses. The subscript expressions must be in the order of the dimensions of the array and each must denote a value of an index of an element in the corresponding dimension. The syntax diagram is shown in Figure 2.6.

```
1  <VARIABLE CONNECTING_POINT TEST OR DIAGNOSIS> ::=
            <IDENTIFIER> [(<SUBSCRIPT_EXPRESSION>
            [,<SUBSCRIPT EXPRESSION>]*)]
   2  <SUBSCRIPT_EXPRESSION>::= <SUBSCRIPT>
                               | <SUBSCRIPT>-1
                               | <ARITHMETIC_EXPRESSION>
      3  <SUBSCRIPT> ::= <IDENTIFIER>
```

FIGURE 2.6. Syntax Diagram of Variables, Connecting Points, Tests or Diagnoses

Generally, a variable element in NOPAL represents only a single value, i.e. it cannot be assigned a number of values at different times during the execution of the program.

In ATLAS, all the elements of an array can be referenced through a single reference in an iteration loop such as in the following example:

```
FOR I=1 TO N
   V(I) = ---
END FOR
```

It is not possible to provide iteration statements in NOPAL. It is sufficient to use the subscripted form of the variable. Thus it is sufficient to provide the statement

V(I) = ---

As shown in Figure 2.6 we will distinguish between three forms of subscript expressions. The first form consists of only a <u>free subscript variable</u>, which may assume any positive integer value which corresponds to an index of an element of an array. We will refer to the free subscript variable briefly as a <u>subscript</u>. A subscript is an identifier which is separately defined as a subscript in a statement. For example,

V(I) refers to any one of the elements of the one dimensional array (vector) V. If V has 10 elements then I is declared in a statement as a subscript with the bounds 1<=I<=10. Thus I may have all the integer values within these bounds. The second type of subscript expression has the form of subscript minus 1. For example, V(I-1) refers to an element with an index of one less than the value of I. The last form of subscript expression may be an arithmetic expression of any other form.

The following are examples of array identifiers:

V(I,6) refers to the 6th element in the Ith row of V.

CONN(J) refers to the Jth connection point of CONN. Note that it is necessary to use a separately defined connection file to map the vector of the symbolically named connecting point vector CONN into the corresponding ATE connecting points (further discussed in Section 6).

DIAG(K) refers to the Kth instance of the diagnosis DIAG.

Only the first form of a subscript expression, i.e. a free subscript variable, may be used with a connection point or a diagnosis.

The variables in NOPAL have data types that are assigned implicitly or explicitly. There are presently four data types: DEC[IMAL], INT[EGER], DIG[ITAL], and BOOL[EAN]. DECIMAL, DIGITAL and BOOLEAN data types are assigned automatically based on whether a variable is referenced in an arithmetic, digital or boolean expression, respectively. Data types are also propogated from the data type of the values returned by functions and declared in the function declaration statements (discussed in Section 6). DECIMAL is the default data type. A variable that is referenced with subscript expression(s) is implied to be an array. Thus it is not necessary to provide data declaration statements except for a) INTEGER data type, b) to describe tree structured data. Declaration statements are optional; this is described in Section 7.

In the present version of NOPAL the names assigned to variables, connecting points, tests or diagnoses must be unique. We refer to this requirement by saying that these entities must be <u>global</u> to the entire specification. Actually, NOPAL allows the use of the same names to denote different variables in different tests. This is referred to as allowing use of <u>local</u> variables in the scope of respective tests. This capability, however, is not supported by the EQUATE-ATLAS compiler and therefore may not be used in the current version of NOPAL.

2.3.1 FUNCTIONS

A function is a relationship between result arguments and a set of input arguments . The function must also be declared, (discussed in Section 6.1) and separately defined (discussed below and in Section 8). The function accepts input arguments and returns the values of result arguments. The argument list is enclosed in parenthesis and follows the function name. The function definition may involve computation, application of stimuli or taking of measurements at ATE-UUT connection points. An argument of a function may be an arithmetic expression, a digital expression, a condition, a measured value, or a string. An * may also be used to denote a default value of an argument. An argument of stimuli or measurement functions may include the name of the unit of the physical entity that the argument represents. Figure 2.7 gives the syntax diagram for functions.

```
1 <FUNCTION> ::= <IDENTIFIER>[(ARGUMENT>[<UNIT>]
                 [,<ARGUMENT>[<UNIT>]]*)]
  2 <ARGUMENT> ::= <ARITHMETIC EXPRESSION>
                 | <DIGITAL EXPRESSION>
                 | <MEASUREMENT CONDITION>
                 | <STRING CONSTANT>
                 | <CHARACTER STRING>
                 | *
    3 <MEASUREMENT CONDITION>
         ::= <RELATION><ARITHMETIC EXPRESSION>
             [+-<ARITHMETIC EXPRESSION>[%]]
```

FIGURE 2.7. Syntax Diagram For Functions.

Arithmetic and digital expressions are further discussed below in sections 2.7 and 2.8. A measurement condition may be used only in a measurement function, i.e. where measurement devices are employed. The produced program verifies the condition and based on this determines whether the test passes or fails. For example FREQ_METER(<UPPER_LIMIT HZ) measures the frequency and verifies that it is less than the value of the UPPER_LIMIT variable. The measurement condition can also be expressed as deviation from a value by appending the +- operator followed by the amount of deviation, which can be expressed absolutely, or as a percentage using the % symbol.

The definition of a function is provided separately in any one of the following ways.

1) A function may be built-in in the EQUATE-ATLAS compiler or in the NOPAL processor. These are all computational functions. The functions supplied by the EQUATE-ATLAS compiler are shown in Table 2.1, and those supplied by the NOPAL processor are shown in Table 2.2.

FUNCTION NAME	DEFINITION	ARGUMENT RANGE (Approx.)	FUNCTION RANGE (Approx.)
ABS	$\lvert A \rvert$	$-10^{75} <= A <= 10^{75}$	$0 <= F <= 10^{75}$
ALOG	$\text{antilog}_{10}(A)$	$-10^{75} <= A <= 10^{75}$	$0 <= F <= 10^{75}$
ATAN	arctan(A)	$-10^{75} <= A <= 10^{75}$	-PI/2 <= F <= PI/2 radians
COS	cosine(A)	$-10^{75} <= A <= 10^{75}$ radians	-1.0 <= F <= 1.0
DEG	A * 180/PI	$-10^{75} <= A <= 10^{75}$ radians	$-10^{75} <= F <= 10^{75}$ degrees
EXP	e^{A}	$-10^{75} <= A <= 172$	$0 <= F <= 10^{75}$
INT	integer part of A	$-10^{75} <= A <= 10^{75}$	$-10^{75} <= F <= 10^{75}$
LN	$\log_{e} A$	$0 <= A <= 10^{75}$	$-10^{75} <= F <= 172$
LOG	$\log_{10} A$	$0 <= A <= 10^{75}$	$-10^{75} <= F <= 75$
RAD	A * PI/180	$-10^{75} <= A <= 10^{75}$	$-10^{74} <= F <= 10^{74}$
RANDOM	random number	no argument	-1.0 <= F <= 1.0
SIN	sine(A)	$-10^{75} <= A <= 10^{75}$ radians	-1.0 <= F <= 1.0
SQRT	A ** 0.5	$0 <= A <= 10^{75}$	$0 <= F <= 10^{38}$
TAN	tangent(A)	$-10^{75} <= A <= 10^{75}$ radians	$-10^{75} <= F <= 10^{75}$

TABLE 2.1 Built-in ATLAS Evaluation Functions.

FUNCTION NAME	DESCRIPTION
FALSE	denotes the value '0'B.
LAST	Returns value of the last element along the dimension given in the second argument. Example B=LAST(A(I),I) defines B as the last element of A(I).
MAX	Selects the maximum element of the first argument (array) along the dimension given in the second argument and returns its value. i.e. HIGHEST(I)=MAX(V(I,J),J) where HIGHEST(I) is the largest element of V(I,J) along the second dimension.
MIN	Same as MAX, but returns minimum value.
SUBS	Defines a free subscript variable. It has two arguments. The first argument is a character string of a list of variable array names and, if more than one dimension, the dimension numbers where the subscript is used. The second argument is the numbers of elements along the respective dimension. * may be used in the second argument to denote that the number of repetitions is defined elsewhere. This is further discussed in section 4.
SUM	Adds all elements of the first argument along the dimension given in the second argument and returns the sum. i.e. SUMQ=SUM(QUAN(I),I) if QUAN= (1,2,3,4) then SUMQ=10

TABLE 2.2. Built-in NOPAL Functions.

2) A function may be defined by the user in the EQUATE-ATLAS language and placed in a user's library file. It must be submitted together with the program produced by the NOPAL processor for the ATLAS compilation.

3) Primitive stimulus functions, which essentially use an ATLAS statement with the "apply" or "do digital test" verbs, and primitive measurement functions which essentially use an ATLAS statement with the "measure" or "monitor" verbs, may be defined within the function declaration in the ATE part of the specification (described in Section 6). The programs for these functions are generated by the NOPAL processor and included in the produced program.

4) Functions may also be defined in the NOPAL language in a separate secondary module. It is called modfun (discussed in Section 8). The NOPAL processor produces an ATLAS program for such a function. The produced function programs must be submitted to the ATLAS compiler together with the program produced for the main module, similar to 2 above.

All the functions referenced in a specification must also be declared in the ATE part of the specification.

2.4 KEYWORDS AND LABELS

A NOPAL specification consists of several independent parts and each part consists of statements. A keyword is used to designate the begining of each part or to identify the type of statement. The keywords are reserved words in NOPAL. A keyword may be followed by a label which is the name of the respective part. The labels are in many cases optional. They are further discussed in the sections which describe the respective parts of the specification. Table 2.3 shows by the indentation the hierarchial breakdown of the specification into parts and subparts and the respective keywords that are used. Generally, the system truncates the keywords beyond the fourth character.

Parts and Subparts	Keywords
Specification	[NOPAL] SPECIFICATION
Test Part	
Test	TEST
Stimuli	STIM[ULI]
Measurement	MEASUREMENT
Statement	CONJ[UNCT]
Variable Declaration	SOURCE,TARGET
Logic	LOGIC
Diagnosis	DIAGNOSIS
Operator Message	OPERATOR MESSAGE
Affected Components	[AFFECTED]COMPONENT
Other Parameters	[OTHER]PARA,PARM
Message Name	TYPE,PRINT
Timing	TIME
Operator Response	RESPONSE
Options	Y,N,PROCEED
Message Definition	MESSAGE
Alias	ALIAS
Text	TEXT
Function	MODFUN
UUT Part	
Components Failed	COMP[FAIL]
Alias	ALIAS
Failure Function	FAIL[FUNCTION]
Parameter	PARA,PARM
Index	INDEX
Protect	PROTECT
Comment	COMMENT
UUT-ATE Connections	UUT_POINT
Alias	ALIAS
Connector Type	CONNECTOR
Limit	LIMIT
ATE Part	
Functions	FUNCTION
Alias	ALIAS
Type	TYPE=S,M,F,E
Number of Pins	PIN
Parameter	PARA,PARM
Value Returned	VALUE
Comment	COMMENT
ATE Connections	ATE_POINT
Alias	ALIAS
UUT Point	UUT_POINT
Comment	COMMENT
Data Declarations Part	
Data Declaration	DCL
Data Type	INTEGER,DECIMAL,DIGITAL, BOOLEAN FILE,GROUP,RECORD,ARRAY
End	END

TABLE 2.3. Keywords.

Figure 2.8 shows the syntax of labels.

<LABEL> ::= <IDENTIFIER> | <UNSIGNED INTEGER>

Figure 2.8. Syntax Diagram For Labels.

A label may be either an identifier or an unsigned integer. Normally in NOPAL a block of statements that constitutes a part follows the header statement which gives the keyword and label of the part. Statements may also be given in an arbitrary order. If they do not follow the header statement then it is necessary to have the part label follow the statement keyword, in parenthesis. For example:

ASRT A(B):<statement>

means that this is an assertion type statement with a label A in a test which has the label B.

This is further explained in Section 3.

2.5 UNITS

An argument of a stimuli or a measurement function may be followed by the name of the physical unit of the respective applied or measured signals. These units are translated by the NOPAL processor into the respective units in the function declaration in the ATE part of the specification. The units used in NOPAL for the EQUATE-ATE are shown in Table 2.4.

BASIC UNIT

Physical Quantity	Basic Unit	Symbol
Current	Ampere	A
Gain (Attenuation)	Decibel	DB
	Percent	PC
		%
Power	Watt	W
	Decibel (above 1MW)	DBM
Angle	Degree	DEG
Frequency	Hertz	HZ
Resistance	Ohm	OHM
Time	Second	SEC
	Minute	MIN
	Hour	HR
Volts	Volt	V
Insertion-Reflection	Coefficient	DB_DEG
Angle + Frequency		DEG_HZ
Impedance + Angle + Frequency		OHM_DEG_HZ

ALLOWED PREFIXES FOR BASIC UNITS

Prefix	Multiplication of Basic Unit
K	10^{3}
MEG	10^{6}
G	10^{9}
C	10^{-2}
M	10^{-3}
U	10^{-6}
N	10^{-9}
P	10^{-12}

TABLE 2.4. Units Used in NOPAL for EQUATE-ATE.

The unit names listed below may be
used in cases where the function has been separately written by the
user directly in ATLAS or specified in a NOPAL module .
In such cases the user must ensure
that the function includes translation of the units
used into appropriate EQUATE-ATLAS acceptable units. The additional
unit
symbols that may be used in these cases are: G, M, CD, CM, FD, FT,
GM, IN, HP, LB, LI, LM, AMP, BAR, ERG, GAL, LUX, PPM, PSI, RAD, RPH,
RPM, RPS, MHD, CU_M, DEGC, DEGF, DYNE, INHG, LINE, MMHG, NEWT, SQ_M,
STER, VOLT, WATT, CU_FT, FT_LB, HENRY, JOULE, IND_IP, POUNDAL,
BRAKE_HP. These units may be also prefixed as shown in Table 2.4

2.6 ARITHMETIC EXPRESSIONS

Figure 2.9 shows the syntax for arithmetic expressions.

```
1 <ARITH_EXPR> ::=  [<SIGN>]<TERM> [<ADD_OP> <TERM>]*
   2 <TERM> ::= <FACTOR> [<MULT_OP> <FACTOR>]*
      3 <FACTOR> ::= <PRIMARY> [** <PRIMARY>]*
         4 <PRIMARY> ::= <UNSIGNED_NUMBER> | <VARIABLE>
                       | <FUNCTION> | (<ARITH_EXPR>)
      3 <MULT_OP> ::= * | /
   2 <ADD_OP>  ::= + | -
```

FIGURE 2.9. Syntax of Arithmetic Expression.

An arithmetic expression is a combination of constants, variables and functions of DECIMAL data types connected by arithmetic operators (+, -, *, /, and **) with subexpressions enclosed in parenthesis. The expression yields a value of a DECIMAL data type when evaluated. The order of precedence in evaluating an arithmetic expression is shown below:

(1) expressions enclosed in parenthesis

(2) functions

(3) array variables

(4) exponentiation (**)

(5) multiplication (*) and division (/)

(6) addition (+) and subtraction (-)

In case of equal precedences the leftmost subexpression is evaluated first, except in exponentiation where the precedence is from right to left.

Examples of arithmetic expressions are:

```
X
(X+Y)*Z
MAX(PQ,Q/P)**SQRT(1/RANGE(X))
```

2.7 DIGITAL EXPRESSIONS

Figure 2.10 shows the syntax for digital expressions.

```
1  <DIGITAL_EXPR> ::= <DIG_TERM> [<DIG_OP> <DIG_TERM>]*

   2    <DIG_TERM> ::= [<NOT>]<DIG_PRIMARY>
                     | <SHIFT_ROTATE><DIG_PRIMARY><ARITH_EXPR>

            3  <DIG_PRIMARY> ::= <DIG_VARIABLE>
                               | <STRING_CONSTANT>
                               | <DIG_FUNCTION>
                               | (<DIGITAL_EXPR>)

   2 <DIG_OP> ::= "& */AND/*
                | "| */OR/*
                | "+ */XOR/*

            3 <NOT> ::= "^*/NOT/*

            3 <SHIFT OR ROTATE> ::= "> */SHIFT/*
                                  | "* */ROTATE/*
```

FIGURE 2.10. Syntax of Digital Expressions.

A digital expression consists of a list of digital variables, constants and functions, of DIGITAL data types and represented as a 64 bit long field, connected by digital operators. The digital operators operate on the respective bit positions. The order of precedence in evaluating a digital expression is shown below:

1) expressions enclosed in parenthesis

2) functions

3) NOT operator

4) SHIFT or ROTATE, to the right if the integer value of the arithmetic expression is positive (to the left if negative).

5) AND operator

In cases of equal precedence, the leftmost subexpression is evaluated first.

2.8 IF-CLAUSE AND RELATIONAL EXPRESSION

Figure 2.11 shows the syntax for an <u>If-Clause</u>.

```
1 <IF_CLAUSE> ::= IF <BOOLEAN_EXPR> THEN
   2 <BOOLEAN_EXPR> ::= <BOOLEAN_FACTOR> [<BOOLEAN OPERATOR>
                         <BOOLEAN_FACTOR>]*
      3 <BOOLEAN OPERATOR> ::=  |  |&
      3 <BOOLEAN_FACTOR> ::= <RELATIONAL_EXPR>
                            | <BOOLEAN VARIABLE>
                            | <BOOLEAN FUNCTION>
                            | (<BOOLEAN_EXPR>)
            4 <RELATIONAL_EXPR> ::= <ARITH_EXPR> <RELATION>
                                      <ARITH_EXPR>
                            | <DIG_EXPR>[^] = <DIG_EXPR>
                5 <RELATION> ::= =|<=|>=|<|>|^=
```

FIGURE 2.11. EBNF Specification of IF-CLAUSE.

The If-Clause is a boolean expression enclosed between the keywords IF and THEN. The If-Clause must be followed by a simple statement, which may be followed optionally by the word ELSE and an additional If-Clause and statement. Statements are described in section 4. A boolean expression may be evaluated to TRUE or FALSE . The boolean expression consists of a list of relational expressions and variables or functions of BOOLEAN data types, optionally preceded by the ^ (NOT) operator and interconnected by & (AND) or | (OR) operators.

The precedence order of evaluation of a boolean expression is as follows:

1) expressions with parenthesis

2) relational expressions

3) ^ operator

4) & operator

In case of equal precedence the expression is evaluated from left to right. If the boolean term evaluates to TRUE, the statement following the THEN is evaluated to obtain a value; otherwise the statement following ELSE is evaluated (if there is an ELSE part). Another If-Clause may follow in the ELSE part. Thus it is possible to nest the If-Clauses in the ELSE part, giving the capability of writing long decision trees to select a statement.

A <u>relational</u> <u>expression</u> consists of two arithmetic expressions connected by one of the relational operators or two digital expressions connected by the = sign. It evaluates to a boolean TRUE or FALSE value. For example, A>B is a boolean expression composed of a relational expression. A>B & C = D is another example of a boolean expression. IF DELAY> 10 THEN is an If-Clause where the boolean expression (in this case a relational expression) evaluates to TRUE only when the value of

DELAY is greater than 10, otherwise it evaluates to FALSE.

CHAPTER 3

MAJOR COMPONENTS OF A SPECIFICATION MODULE

A NOPAL specification is organized hierarchically into relatively independent parts. The objective of this organization is to make it easier for the user to concentrate on the parts, one at a time. A specification consists of a main module and, if necessary, a number of secondary modules. The main module specifies the tests and declares the UUT, ATE and data related entities. A secondary module specifies the modfuns, i.e. the functions. A module must include a header statement identifying the name of the module. Header statements must also be provided for each of the tests or modfuns included in the module. Further, header statements must be provided for the stimuli and measurements which are constituents of respective tests.

The syntax of these header statements is shown in Figure 3.1.

```
<MAIN MODULE HEADER STATEMENT> ::=
                              [NOPAL] SPEC[IFICATION] <LABEL>;
                              | MAIN <LABEL>;
<SECONDARY MODULE HEADER STATEMENT> ::= MODULE <LABEL>;
<TEST HEADER STATEMENT> ::=
                        TEST [<IDENTIFIER>(<SUBSCRIPT><,SUBSCRIPT>*)];
<MODFUN HEADER STATEMENT> ::=
                          MODFUN <LABEL><ARGUMENTS DECLARATION>;
<STIM OR MEAS STATEMENT HEADER> ::=
                           STIM[ULI] [<LABEL>] [(<LABEL>)];
                           | MEAS[UREMENT] [<LABEL>] [(<LABEL>)];
```

Figure 3.1. Syntax of Header Statement for Modules and their Major subparts: TESTs, MODFUNs, STIMULI and MEASUREMENTS.

A header statement consists of a respective keyword (SPEC|MAIN, MODULE, TEST, MODFUN, STIMULI and MEASUREMENT) followed by a label which is used to name the respective part. Note that a test may be either of scalar or array type, i.e. it may be evaluated once or repeated. If of array type, it must be followed by the list of subscripts.

It is necessary to identify the type of the statement in the header of the statement. The syntax for these headers of statements is shown in Figure 3.2.

```
<CONJUNCTION STATEMENT HEADER>::=
                         CONJ[UNCTION]<LABEL>[(<LABEL>)][:]
<ASSERTION STATEMENT HEADER>::=
                {ASSE[RTION]|ASRT}[<LABEL>][(<LABEL>)][:]
<LOGIC STATEMENT HEADER> ::= LOGIC[<LABEL>] [(<LABEL>)][:]
<DIAGNOSIS STATEMENT HEADER> ::= DIAG[NOSIS] [<LABEL>][:]
<MESSAGE STATEMENT HEADER> ::= MESS[AGE][<LABEL>][:]
<UUT COMPONENT FAILURE STATEMENT HEADER>::=
                          COMP[_FL]<LABEL>][:]
<UUT CONNECTION DECLARATION HEADER>::=
                          UUT_[POINT][<LABEL>][:]
<ATE FUNCTION DECLARATION HEADER> ::=
                          FUNC[TION][<LABEL>][:]
<ATE CONNECTION DECLARATION HEADER> ::=
                          ATE_[PNT][<LABEL>][:]
<DATA DECLARATION HEADER> ::= DCL[<LABEL>][:]
<END STATEMENT> ::= END[<LABEL>];
```

Figure 3.2. Syntax of Statement Headers.

Each statement must be prefixed with a keyword indicating the type of the statement. CONJUNCTION, ASSERTION and LOGIC types of statements are used in specification of individual tests. DIAGNOSIS and MESSAGE type statements may be common to a number of tests. UUT, ATE and data declaration statements are also global to the entire module. These headers are further described in Sections 4, 5 and 6 in the discussion of the respective statement types.

A label may be assigned to each statement, purely for use in documentation. Normally the CONJUNCTION, ASSERTION, and LOGIC statements which belong to a test form a block which immediately follows the test header statement. If a statement is placed outside the respective block then it must include in parenthesis the label of the part where it belongs. This makes it possible to insert statements anywhere in the specification.

All the parts in a module must be assembled and sumbitted together to the NOPAL processor. NOPAL modules may be submitted to the NOPAL processor separately. However, the produced ATLAS programs must be compiled jointly. The programmer may concentrate on each module and on each test or function independently.

The succeeding sections describe in greater detail the specification of individual parts and the composition of related statements.

CHAPTER 4

TEST SPECIFICATION

The composition of specifications of tests is the major task in specifying a NOPAL main module. It is also the most demanding of the user of NOPAL. The task consists of composing NOPAL language descriptions of tests, one by one, in any preferred order. The other parts of a main module, the UUT, ATE and DATA declarations, are discussed in Sections 5,6 and 7, respectively. They are used primarily for checking the test specification and are included in the documentation of the test program. Therefore their composition is more mechanical and involves less originality than the composition of tests.

The overall test part consists of individual test specifications. An individual test specification is composed of a number of subparts which are described in respective subsections below. As stated in Section 3, each test is preceded by a header statement consisting of the keyword TEST and an identifier. For example: TEST A;. A test specification normally consists of a block of statements that immediately follow the test header statement. The block of statements is further divided in sub-blocks for STIMULI and MEASUREMENT and a single LOGIC statement. The statements in these sub-blocks describe the stimuli to be applied to the UUT connecting points, the measurements that need to be taken at UUT connecting points and the relations that need to be evaluated. With each test is associated the notion of the test passing or failing, which is based on evaluation of conditions in the respective statements. The LOGIC statement specifies the diagnoses selected based on the test passing or failing.

One objective of NOPAL is to relieve the user from considering the relative timing of the application of the stimuli and measurement devices. The user refers to these devices through function names, and may consider their timing as specified internally to the functions. Thus the timing problem is delegated to the NOPAL process scheduling decisions and/or to the definition of functions (described in Sections 7 and 8).

The order of application of stimuli and measurements is determined by the NOPAL processor. By default, the NOPAL processor schedules the STIMULI block of statements to precede the MEASUREMENT block of statements. However if any of the arguments of a stimulus function depends on the result values of a measurement function, then the order is reversed, and the NOPAL processor schedules the measurement to precede the respective stimulus. In addition the user may specify any additional necessary fixed delays between application of stimuli and measurements as arguments of the respective measurement functions. A sequence of stimuli and measurement applications may also be defined in separate tests and the NOPAL processor determines their schedule.

The above is adequate in cases where the measurement is to be made at some instant, requiring very little time, so that it can be considered as instantaneous. An example is measuring a voltage. The situation is more complicated in cases where a measurement is conducted over a period of time, such as in measuring a time interval or counting a number of events in a time interval. The synchronization of stimuli and measurements must then be specified in the respective function which combines both stimuli and measurements.

Thus the relative timing of stimuli and measurements in a test is either determined by the NOPAL processor or by the function which applies the devices. In composing a specification the user need not be concerned with the timing of application of devices and may view the devices as applied simultaneously and internally synchronized.

A statement that specifies simultaneous application of stimuli or measurements is referred to as a conjunction. If a test requires the application of sets of devices sequentially, one set after the other, then it is necessary to divide the test into corresponding separate sub-tests. A test may contain at most two conjunction statements, one for application of stimuli devices and one for application of measurement devices.

A purely evaluation statement is referred to as an assertion.

STIMULI or MEASUREMENT blocks are optional and may be omitted if not appropriate in a test. If a statement is stated outside the block it must refer in parenthesis to the STIMULI or MEASUREMENT labels, respectively.

Each test may also have a logic statement which specifies the rules for selection of diagnoses based on whether the conditions in the statement are satisfied, i.e. based on the passing or failing of the test. In this way, a test may select diagnoses. Some diagnoses may be selected by the conjunction of several tests.

In composing conjunctions or assertions, the user may refer to scalar or array variables or connection points. A statement where subscripts, array variables or array connecting points are referenced represents multiple instances of the statements, one instance for each

index value of elements in the respective array. We will refer to such statements as array statements. Conjunction and assertion statements are described in Section 4.1.

Tests and diagnoses may also be scalars or arrays. If arrays, they must reference the respective subscripts. A number of rules must be observed in use of array tests. First, the conjunction or assertion statements in an array test that define values of variables must contain a reference to the subscript of the variable. Note that this rule doesn't apply to condition conjunctions where a test of a value of a variable is implied. Next an array test may select only array diagnoses, and a scalar test may select only scalar diagnoses. Array tests and the array diagnoses selected by a test, must be specified with common subscripts, thus relating the Kth element (or instance) of the test to the Kth element (or instance) of a diagnosis. Instances of different array tests may also be related through referring to common subscripts. A kth instance of one test is thus related to the kth instance of another test, if the same subscript variable is used in both tests. This is described further in connection with diagnoses selection in Section 4.2

Diagnoses referenced in logic statements must also be specified in diagnoses statements. The specification includes the referencing of a message, the insertion of constants or variables in the message and operator's response. The specification of diagnoses is described in Section 4.3.

Finally it is necessary to specify the messages referred to by diagnoses in a message statement. Section 4.4 describes composing message statements. Diagnoses and messages may be used in one or several tests. Therefore a diagnosis or a message may be specified anywhere, without reference to a test.

4.1 STIMULI, MEASUREMENT OR EVALUATION STATEMENTS

The objective of these statements is to specify the test's use of stimuli and measurement devices and the conditions for the passing or failing of the test. A statement associated with stimuli or measurement applications is called a conjunction. The stimuli or measurement devices in a conjunction statement are viewed as applied simultaneously or in some internally coordinated synchronization. The statements that are evaluated purely computationally are call assertions. They determine test passing or failure or define values of variables.

4.1.1 Conjunctions

The purpose of conjunctions is to specify use of stimuli devices to apply a signal to the UUT, and metering devices to determine values of measured variables. A measurement conjunction can also verify conditions relating to the test passing or failing. there may be at most one stimuli conjunction and one measurement conjunction per test. The notion of a conjunction corresponds to simultaneous, or internally synchronized, application of devices. If a test requires the application of sets of devices, one set after the other, then it is necessary to divide the test into corresponding separate sub-tests.

A header statement states whether the conjunction is for stimuli or measurement application. If a conjunction statement follows immediately after the respective stimuli or measurement header statement then it needs not include a reference to a label. Otherwise it must refer to the label in the appropriate stimuli or measurement header . The syntax of header statements for tests, stimuli and measurements has been shown above in Figure 3.1.

The syntax of a conjunction is shown in Figure 4.1.

```
1  <CONJUNCTION STATEMENT>::=<CONJUNCTION STATEMENT HEADER>
                       [<IF CLAUSE>]<TRIPLET>[&<TRIPLET>]*
                 [ELSE[<IF CLAUSE>]<TRIPLES>[&<TRIPLET>]*]*
                   [<SOURCE-TARGET DECLARATION>];
               | <CONJUNCTION STATEMENT HEADER>SAME AS<LABEL>
                      [EXCEPT<TRIPLET>[&<TRIPLET>]*];

   2  <CONJUNCTION STATEMENT HEADER>::=
                               CONJ[<LABEL>][(<LABEL>)]][:]

   2  <TRIPLET>::=<CONNECTION LIST>=<FUNCTION>

        3<CONNECTION LIST>::=<<CONNECTING POINT>[,<CONNECTING
           POINT>]*>>

   2<SOURCE TARGET DECLARATION>::=
                    [SOUR[CE]:<VARIABLE>[,<VARIABLE>]*]
                    (TARG[ET]:<VARIABLE>[,<VARIABLE>]*]
```

FIGURE 4.1. Syntax of Conjunctions.

A conjunction statement consists of three parts. The first part of a conjunction statement is the conjunction statement header. It consists of the keyword CONJ, followed optionally by a label and if necessary also by the label of the appropriate stimuli or measurement, in parentheses.

The second part, which is the heart of the conjunction statement is an optional If-Clause followed by a list of triplets. Each triplet describes application of some test devices. A triplet is so called because it has three parts, as follows: 1) a list of UUT connection

points where devices are to be connected, 2) the = sign and 3) a function reference. The list of connecting points must be enclosed in angle brackets. Only a function that applies stimuli or measurements may be referenced in a conjunction statement. The referenced function includes internally the use of stimuli and measurement devices. It may also include evaluation of other arguments. The selection of ATE devices requires in some cases detailed knowledge of the ATE. However, the user of NOPAL refers to device applications through referencing a function, without having to know the details of how the function employs devices. The definition of the function is done separately, and placed in a library for general use, or possibly defined after the entire specification has been completed. The user need not be concerned with the internal workings of the function, only with the input and result arguments. Function references have been described in Section 2.3.2. Function definition is described in Sections 7 and 8. The arguments of the function may be control parameters or expressions that must be evaluated or verified. Triplets are connected by the & symbol denoting their parallel application. A set of triplets may be selected based on whether the preceding boolean condition in the If-Clause evaluates to TRUE.

The third part of a conjunction statement is the SOURCE-TARGET declaration. It consists of two lists of variables. First is the list of independent variables, i.e. those variables evaluated elsewhere , or input, and only referenced in the conjunction statement. These variables are called the SOURCE variables. The second list is of the dependent variables, i.e., those evaluated based on the conjunction statement. These are called the TARGET variables. Identifying the SOURCE variables is optional. However the TARGET variables must be identified.

Following is an example of the use of conjunctions. Assume that a test calls for application of V1 DC Volts between UUT test points T1 and GROUND, and measuring the voltage at an array of five tests points T2(I) and GROUND. I is a subscript which may have any integer value between 1 and 5. We will refer to the values measured at test point T2(I) as V2(I). We will use a function named DC_APP for applying a DC voltage, and a function DC_MEAS for measuring DC voltage. The above may be specified in the four statements as follows.

```
STIM S1;

CONJ:(<T1,GROUND>=DC_APP(V1 V));

MEAS M1;

CONJ:(<T2(I),GROUND>=DC_MEAS(V2(I) V) TARGET:V2(I));
```

The first statement is a stimuli header statement with the label S1. It is followed by a conjunction (with label ommitted) which states that the V1 volts are applied by the function DC_APP at the UUT

connecting point T1. Next, the measurement header with label M1 is followed by a conjunction that describes DC volts measurements, by the function DC_MEAS, at each of the five array connecting points T2(I). These measurements determine the values of the respective five elements of the array variable V2. The stimuli conjunction is a scalar statement, while the measurement conjunction is an array statement namely it has five instances for every value of I from 1 to 5. The subscript I must be separately defined by a SUBS function as follows.

```
I=SUBS('V2',5);
```

meaning that I corresponds to the first (and only) dimension of V2 which has 5 elements. Subscript definition is further discussed in Section 4.1.3.

Sometimes several tests use the same or similar stimuli or measurements. Also the conjunctions may be long and complex. To avoid rewriting the same conjunction in several tests it is possible to refer in one test to a conjunction statement in another test. This is shown by the alternative conjunction statement syntax shown in Figure 4.1. The reference to another conjunction consists of using the keywords SAME AS followed by the label of the stimuli or measurement header statement where the conjunction is fully specified. It is also possible to qualify this reference by adding the keyword EXCEPT followed by modified or added triplets. The NOPAL processor then copies the full conjunction and modifies the triplets which have the same connecting points as the triplets that follow EXCEPT keyword. For example, assume that we want to make in another test the same measurements as in the previous example, only we want to add application of DC voltage V3 as points T3 to GROUND. This can be written as follows.

```
STIM;
CONJ:SAME AS S1 EXCEPT <T3,GROUND>=DC_APP(V3 V);
MEAS;
CONJ: SAME AS M1;
```

Use of the If-Clause together with some of subscripts can also shorten the statement of a test. For instance the above two examples may be considered as two instances (or 2 rows of five elements) of the same test. Let the subscript J refer to these two instances. Then we can state the above examples as

```
STIM;
CONJ: IF J=1 THEN <T1, GROUND>=DC_APP(V1 V)
              ELSE(<T1,GROUND>=DC_APP(V1 V))&(<T3,GROUND>=
               DC_APP(V3 V));
MEAS;
CONJ:(<T2(I),GROUND>=DC_MEAS(V2(J,I)  V)) TARGET: V2(J,I);
```

The subscript definitions are

```
J=SUBS('V2:1',2);
I=SUBS('V2:2',5);
```

V2 must now have two-dimensions. There are two rows, each with five elements. The two rows correspond to different stimuli. The IF-Clause specifies that for J>1 it is also necessary to apply V3 voltage to test point V3.

As described in Section 2.3.2, a condition argument may be used in a measurement function. The condition is applied to the measured value. The measurement function can then be viewed as performing the checking of the condition. The test is considered to fail if the condition is not satisfied. For example, the measurement conjunction in the last example may be written as

```
MEAS;
CONJ:<T2(I),GROUND> = DC_MEAS(=1.0+-.1 V);
```

It means that the voltage measured between point T2(I) and ground is compared with the limits of .9V and 1.1V. If it is, the test passes, otherwise, the test fails.

Note that this conjunction does not have a TARGET declaration as no variable is defined by it. We will refer to a measurement conjunction that does not define a variable, i.e. without a TARGET variable, on a condition conjunction.

4.1.2 Assertions

There are two types of assertions. The first type is a specification of a condition for a test passing or failing. The second type is a definition of a value of a variable. Thus, an assertion must be composed for each test passing condition (not specified in a conjunction condition argument) and for each variable that is defined. The assertions are purely computational and may not reference stimuli or measurement functions. Figure 4.2 describes the syntax of assertions.

```
1  <ASSERTION STATEMENT> ::=<ASSERTION STATEMENT HEADER>
                   [<IF CLAUSE>]<SIMPLE ASSERTION>
                   [ELSE <ASSERTION>]
                   [<SOURCE TARGET DECLARATION>];
      2 <ASSERTION STATEMENT HEADER>:=<ASSERTION KEYWORD>
                                   [<LABEL>[(<LABEL>)]]:
          3 <ASSERTION KEYWORD>::= ASSE[RTION] | ASRT
      2 <SIMPLE ASSERTION>::=<CONDITION> | <DEFINITION>
          3 <CONDITION>::=<ARITHMETIC EXPRESSION><RELATION>
                       <ARITHMETIC EXPRESSION>
                       [+-<ARITHMETIC EXPRESSION>[%]]
                       |<DIGITAL EXPRESSION>[^]=<DIGITAL EXPRESSION>
          3 <DEFINITION>::=<VARIABLE>=<ARITHMETIC EXPRESSION>
                          |<VARIABLE>=<DIGITAL EXPRESSION>
```

Figure 4.2. Syntax of Assertions.

As shown, an assertion consists of three parts. The first part is a statement header using the keyword ASRT or ASSE[RTION], and optionally a label of the assertion. If an assertion is not in the block of

statements that follows the respective STIMULI or MEASUREMENT header statement, then it is necessary to provide, also in parenthesis, the label of the header statement.

The second part of the assertion statement consists of an optional If-Clause followed by a simple assertion. A simple assertion may be either a test passing condition or a definition of a value of a variable. A definition is an equation with a variable on the left-hand side, an equality sign (=) and an arithmetic or digital expression on the right hand side. A condition consists of an expression on the left hand side, a relation and an expression on the right hand side. The two expressions must be either both arithmetic or digital. The right hand side arithmetic expression may be followed by a specification of deviation limits consisting of a +- operator followed by an expression of the amount of deviation. The deviation may be absolute or expressed in percent using the % character. The relation used in an arithmetic condition may be =, ^=, =>, =<, >, <. Only = and ^ =relations can be used in a digital conditon. If an If-Clause is used, then the simple assertion may be followed by the ELSE keyword followed by another assertion.

If the assertion is a definition of a value of a variable then if an ELSE part is used it must refer after the ELSE to the same left-hand side variables as in the simple assertion following the THEN. Namely an assertion can define only one variable, which must be declared as the TARGET variable. An assertion that has a target variable does not effect the outcome of the test.

A condition assertion may have only SOURCE variables, while a definition assertion must also have a single TARGET variable. The definition of SOURCE variables is optional. If there are only SOURCE variables then the simple assertion defines a test passing condition. A condition evaluates to TRUE or FALSE. If it evaluates to FALSE then the entire test fails. If none of the condition assertions or the condition conjunction in a test evaluate to FALSE, (i.e. they will either evaluate to TRUE or define variables or apply stimuli) then the test passes. The diagnoses are selected based on the passing or failing of tests.

Following is an example of a condition defining assertion:
ASRT: RES>100;

This is a simple assertion which specifies the condition that the value of a SOURCE variable RES should be greater than 100. Note that it does not have a target variable.

A condition may be conveniently stated by specifying allowed deviations. This is illustrated in the three sets of assertions below.

```
(1) ASRT: RES = 100 +- 5%;
(2) ASERT: RES = 110 +- 5.5;
(3) ASRT: RES < 115.5;
    ASRT: RES > 104.5;
```

These are thee ways to specify the condition that RES be between 115.5 and 104.5. The first and second forms are more convenient for specifying desired ranges or allowed deviations on results of testing.

Assertions can use an IF-Clause, which may also be nested in the ELSE parts. The evaluation of the assertion is determined by selection of either the THEN or the ELSE parts. The use of IF-Clause is illustrated in the example below.

```
    ASRT; IF V<60 THEN F = 5E6 +- 60
                  ELSE F = 5E6 +- 2.5;
```

There are here two source variables V and F (their declaration is optional). If V is less than 60, then F should lie between 4,999,940 and 5,000,060. Otherwise it should be between 4,999,997.5 and 5,000,002.5.

4.1.3 Free Subscripts And Array References

The array or repetition specification facility of NOPAL saves labor and reduces the chances of errors in a user's specification. NOPAL facilitates specifying repetition of evaluation of statements and tests and selection of diagnoses (and implied iterations) through use of a combination of the following three methods:

1. by references to repetitive array elements of tests, variables, connections or diagnoses.

2. by explicit declaration of subscripts, using an assertion form with the SUBS function

3. by defining a variable number of elements of an array dimension through defining (in another assertion) a special variable named END_<SUBSCRIPT>.

A subscript variable must be declared in a separate assertion using the SUBS function <u>for each test where it applies</u>. Subscript names are global in a specification module. Thus the same subscript name may be used in more than one test only if they are of the same range, i.e. have the same upper bound. The syntax of subscript definition is shown in Figure 4.3.

```
1 <SUBSCRIPT DEFINING ASSERTION>::=
        <SUBSCRIPT> = SUBS('<PARENT_LIST>',<UPPER BOUND>)
                      TARGET <SUBSCRIPT>;
     2 <SUBSCRIPT>::= <IDENTIFIER>
     2 <PARENT LIST>::= <VARIABLE>[:<DIMENSION NUMBER>]
                        [,<VARIABLE>,[:<DIMENSION NUMBER>]]*
     2 <UPPER BOUND>:= <UNSIGNED INTEGER> | *
```

Figure 4.3. Syntax of Subscript Definition.

A free subscript variable must be a scalar (i.e. unsubscripted) variable. SUBS is the subscript defining function. The first argument of SUBS is a list of variable names which use the free subscript, enclosed in single quotes. If there is more than one dimension then the dimension number that indicates the subscript's position must be added after each variable. thus, V:3 means that the subscript is referenced only in the third dimension of variables V. If the dimension number is not specified then the first dimension is assumed.

The lower bound of a subscript is 1. The upper bound may be given as the second argument of the SUBS function. An upper bound may also be given as '*', meaning that its value may vary. In such a case the upper bound of the subscript must be defined in a separate assertion, as will be discussed below. Each subscript must be defined separately through use of the SUBS function. For example, a subscript I may be defined as a subscript of a variable A with 4 elements by the following assertion:

```
I = SUBS('A',4) TARGET: I;
```

Then an assertion:

```
A(I) = I+5 TARGET: A(I);
```

implies that A(1)=6, A(2)=7, A(3)=8, and A(4)=9.

Consider another example.

```
ASRT:  I = SUBS ('A',10)          TARGET:  I;            (1)

ASRT:  J = SUBS ('A:2, B:1',5)    TARGET:  J;            (2)

ASRT:  IF I<5 THEN A(I,J) = J     ELSE A(J,I) = I
            TARGET:  A(J,I);                             (3)

ASRT:  A(J,I)<=B(J);                                     (4)
```

The first statement defines 'I' as the first subscript of variable A with 10 elements. The second statement defines J as the second subscript of A and also the first subscript of variable B, with 5 elements in the respective dimensions. If I and J are the only declared subscripts in the test module, then we may conclude that A is a two dimensional array with the size declared as 10 by 5, and B is a vector with 5 elements. The third statement defines the values of all the elements of the array variable A(I,J). The fourth statement, gives a condition on the upper limit of A, namely that A is less or equal to

B(J). The passing or failing of the test is based on the evaluation of the condition in the fourth statements.

The upper bound of a subscript may be a variable that depends on other variables. In this case the upper bound is denoted in the subscript defining assertion as *. in this case, it is necessary to define the upper bound by an assertion. This is done as follows. A boolean variable named END_<SUBSCRIPT> is used to denote the condition of the last element along the array dimensions in the subscript definition statement. An assertion is used to define the condition that the variable END_<SUBSCRIPT> has the value FALSE on the last element of the array dimension. This is illustrated by the following example. Assume that we want to perform a measurement repeatedly, each time increasing the applied voltage at T2 until the first occurrence of a measurement (V(I)>100) at T1 . We do not know how many times the measurement is to be repeated but we know how to recognize the last repetition. This may be stated as follows.

```
STIM;
CONJ:   IF I=1 THEN <T2,GND>=DC_APP(.1 V)
                ELSE <T2,GND>=DC_APP(I*.1 V);
MEAS;
CONJ:   <T1,GND>=DC_MEAS(V(I) V) TARGET: V(I);
ASRT:   IF V(I) > 100 THEN END_I = FALSE TARGET:  END_I;
ASRT:   I = SUBS ('V',*) TARGET: I;
```

The first statement is a stimuli header statement. The second statement is an array statement of repeatedly applying I*.1 volts to a test point T2. Next is a measurement header statement. It is followed by an array statement where elements of V(I) are evaluated based on voltage measurememts. The next to the last statement defines END_I=FALSE on the first occurence of V(I)>100. The last statement defines the subscript I.

The use of subscripted variables and UUT points implies definition of arrays. It is necessary to take care that each element is defined unambiguously. Consider the following example:

```
ASRT: I = SUBSCRIPT ('A', 3)  TARGET:  I;
ASRT: A(I) = I+5; TARGET:  A(I);
ASRT: B = A(I); TARGET:  B;
```

The first two statements suggest that A(1) = 6, A(2) = 7, and A(3) = 8. But the third statement is <u>ambiguous</u>. B is multiply related to the three values of A. The third statement must be modified to be unambiguous. For instance by adding an IF-Clause;

```
    ASRT:  IF I = 3 THEN B = A(I);
or
    ASSRT:B = LAST(A(I),I);
```

A subscript expression of a variable must be of one of the following forms:

1. A subscript term ----- e.g. I in A(I) where I is a free subscript,

2. An expression of the form (I-1), -- e.g. A(I-1).

3. Any other form of an expression. This includes a simple variable ---- e.g. X in B(X,I) where X is not declared as free subscript, or another subscripted variable ---- e.g. B(I) in A(B(I)).

Only form (1) is permitted with a target subscripted variable in a conjunction or in an assertion. Also only form (1) is permitted in array connecting points or diagnoses. This is discussed further in subsequent sections.

A test is considered to be an array test with the subscript I only if all the definition statements (not necessarily condition statements) in the test use subscript expressions of the forms 1 or 2 above. (This

condition is imposed to assure efficiency in the produced program.)

The six assertions shown below are further examples. Suppose I and J are free subscripts.

```
X(I) = XO TARGET: X(I); valid                        (1)
A(I) = B(I) +5 TARGET: A(I); valid                   (2)
B(I) = C(I,J) +X(I) TARGET: B(I); invalid            (3)
B(I) = C(I,4)+X(I) TARGET: B(I); valid               (4)
VAL = C(1,5) TARGET: VAL; valid                      (5)
D(I) =D(I-1) + 1 TARGET: D(I); valid                 (6)
```

The first statement is used to define the whole array X with the scalar value XO. The second statement is used to compute target variable A(I) from source variable B(I). These two statements are syntactically and semantically correct. However, the third statement has a two dimensional source variable C which has 2 subscripts while the target variable B is a one dimensional vector This violates the rule that each element of the target variable should be defined unambiguously. If J had been replaced by a constant or a scalar or if there was an IF-Clause with a condition on J, then assertion 3 would have been legal as well. Such situations are shown in the assertions (4) and (5). Assertion (6) shows a relationship between the Ith and I-1th elements of array D. It is being used to define integers from 1 to the upper bound of I.

Consider another example where the range of a subscript is variable, as follows. It is desired to apply a DC voltage to T1 and ground so that the current supplied is 1 + -.1 Amperes. We can use an extrapolation to repeatedly estimate the voltage until the measured current is within the desired bounds. This can be stated as follows.

```
STIM;
CONJ: <T1,GND>=DC_APP(V(J) V);
MEAS:
CONJ: <T1>=A_MEAS(I(J) A) TARGET:I(J);
ASRT:  IF J=1 THEN V(J)=.01
              ELSE V(J)=V(J-1)/I(J-1) TARGET:V(J);
ASRT: J = SUBS('V,I',*) TARGET J;
ASRT: IF(I(J)>.9)&(I(J)<1.1)
              THEN END_J=FALSE TARGET:END_J;
```

V(J) is the voltage applied to T1, in volts, and I(J) is the current measured into T1, in Amperes. V(J-1)/I(J-1) is an extrapolation of the voltage that would correspond to I of 1 Ampere. DC_APP is a function that applies a DC voltage and A_MEAS is a function that measures a current. Starting with a voltage of .01 volts, extrapolated voltages will be repeatedly applied until the current I is within the bounds 1+-.1 Amperes.

4.2 SPECIFICATION OF DIAGNOSES SELECTION LOGIC

The purpose of the LOGIC statement is to specify the diagnoses that are selected based on conditions in the measurement conjunction and/or in assertions in a test. The syntax of the LOGIC statement is shown in Figure 4.4.

```
1 <LOGIC STATEMENT> ::= LOGIC[<LABEL>[(<TEST IDENTIFIER>)]]:

                              <DIAGNOSES LIST>;

     2 <DIAGNOSES LIST>::= <LOGIC OPERATOR><DIAGNOSIS>
                           [,<LOGIC OPERATOR><DIAGNOSIS>]*
          3 <LOGIC OPERATOR::= |
                              ||^
                              |&
                              |&^
                              |*
                              |?
```

FIGURE 4.4. Syntax of LOGIC Statement.

The statement consists of a header followed by list of diagnoses, each preceded by a <u>logic</u> <u>operator</u>.

The header consists of a keyword LOGIC, followed optionally by a label and, in parentheses, by the identifier of the test where the statement belongs, finally followed by an optional colon(:). The test identifier needs to be used only if the logic statement is not within the block of test statements that follows the respective test header statement.

An element of a diagnoses list consists of a logic operator followed by a diagnosis. As explained in Section 2.3.1, the test and diagnoses may be subscripted to denote an array of diagnoses. The meaning is that an element of an array test may select corresponding elements of array diagnoses based on evaluation of corresponding elements of array statements and other statements in the test. A diagnoses list may consist either of only scalar diagnosis, or of only array diagnosis with the same subscripts. The logic operator specifies the dependence of selection of the diagnosis on the outcome of the evaluation of test statements. Note that a test is considered to have failed if <u>any</u> of the conditions in the statements is evaluated to FALSE.

There are six logic operators with the following meanings.

Disjunction: The symbol | preceding a diagnosis means that the diagnosis is selected only on test passing, i.e. if all the conditions in statements in a test evaluate to TRUE. The symbol |^ is used similarly to denote selection of the diagnosis only if test fails, i.e. if <u>any</u> of the conditions in the statement in a test evaluate to FALSE. The disjunction logical operator denotes that selection depends on the test and is independent of the outcome of other tests. The implication is that the diagnosis is issued immediately at the end of the test, since it depends on the test outcome. <u>A diagnosis with a disjunction operator is unique to one test</u>, namely it can be referenced in the LOGIC statement of only one test.

Conjunction: The symbol & preceding a diagnosis means that the selection of the diagnosis depends on the passing of this test and also on some other test(s) where the same diagnosis is referenced in the LOGIC statement with a conjunction operator. The symbol &^ similarly denotes that the diagnosis is selected if this test fails and also depends on the outcome of other tests. The same diagnosis must then be also referenced in a logic statement of one or more other tests, preceeded by the & or &^ logical operator. The diagnosis is selected only if all these tests where the & operator is used pass and all the tests where the &^ operator is used fail, as respectively specified. The implication is that the diagnosis is issued immediately after the last test on the outcome of which it depends.

Independent: The symbol * preceeding a diagnosis means that the diagnosis is selected independently of the outcome of the test. This type of diagnosis selection is typically useful for selecting diagnosis messages which contain instructions for operator actions that may be preparatory for the test. A diagnosis with the * operator may still report values of variables evaluated in the test. In this case it is implied that it is issued only after the evaluation of the test. Otherwise it is issued before the test. A diagnosis with the * operator must also be associated with a unique test.

After: The symbol ? denotes selection of the test as soon as possible after the specified diagnosis is considered for selection or nonselection by another test. This is the opposite to the selection of a diagnosis by a test, which is denoted by the other logic operators. Namely a diagnosis, of another unspecified test, selects the present test to follow as soon as possible. Normally, the test is scheduled immediately after the selection or nonselection of the diagnosis with the ? operator. However, sometimes there are conflicting precedence relations that indicate that one or more tests must precede first. In this case the test with the ? operator diagnosis is delayed and scheduled as soon as possible. In effect, the use of this logical operator determines the order of scheduling of tests, a task normally reserved to the NOPAL processor. However it is necessary to make an exception, particularly to allow the user to specify order of tests and diagnoses when a series of diagnoses includes messages to the operator that instruct the operator in a step by step fashion in performing a complex task.

There are a number of implications in use of array diagnosis with the above logic operators. First if an array diagnosis is referenced in the logic statement the test must be subscripted and there must be at least one other array statement in the test, as well as a subscript definition statement . Thus the evaluation of array statement instances and selection of array diagnoses instances are synchronized.

In the cases of using a disjunction operator (|,|^) or an independent operator (*), the selection depends only on evaluations within the test. The array test is evaluated for each value of the subscript, from 1 to the upper bound. The statements are evaluated and

the array diagnosis are selected, repeatedly.

In the case of using the conjunctions operators (&,&^), the selection of an array diagnosis is effected by more than one array test. Each of these tests must be evaluated for each value of a common subscript in order to select the corresponding element of the diagnosis. In this case there are restrictions on the tests that are involved, as follows.

1. All the diagnoses in these tests must be of array type. The one exception is scalar diagnoses with logic operators ? (which are selected by another test).

2. All these tests must be subscripted by a common subscript and each include a subscript defining statement, with the same subscript name and upper bound specification.

3. If the upper bound is variable it is specified by an *. An assertion must be then used in each of these tests to define the last element condition, denoted by the variable END_<SUBSCRIPT>.

Note that a diagnosis may be referenced in one test or conjunctively in several tests. A logical statement of a test may reference several diagnoses, except that it may reference only one diagnosis with the ? operator. Otherwise, there would be ambiguity in selection of some diagnoses independently. For example consider the example below.

```
TEST T1(I);
   STATEMENTS
   LOGIC:&D1(I);
TEST T2(I);
  STATEMENTS
  LOGIC: &^D1(I),|D2(I);
TEST T3(I);
  STATEMENTS
  LOGIC:?D2(I);
```

Assume that I is the subscript in all three tests with values from 1 to 5. Tests 1, 2 and 3 are evaluated for each value of I. The diagnosis D2(I) is selected on each evaluation of test T2, while D1(I) is selected on each evaluation of both tests, T1 and T2, if test T1 passes and test T2 fails. Test T3 is evaluated following tests T1 and T2, for each value of I.

4.3 SPECIFICATION OF A DIAGNOSIS

Diagnoses statements are used both for identifying the causes of failures and/or for selecting and editing appropriate messages. As explained above, the diagnoses are selected based on the results of evaluation of one or more tests. A diagnosis statement may be specified anywhere in a module. If it is referenced with a | or a * operator in a logic statement, then it must be unique to the respective test only, except that it may also be referenced with a ? operator in logic statements of other tests. A diagnosis may be referenced with the & operator in logic statements of more than one test.

The syntax of a diagnosis statement is shown in Figure 4.5.

```
1 <DIAGNOSIS STATEMENT>::=<DIAGNOSIS HEADER>[:]<KEYWORD_CLAUSE>

                              [,<KEYWORD CLAUSE>]*;

  2 <DIAGNOSIS HEADER>::=DIAG[NOSIS]<IDENTIFIER>[(<SUBSCRIPT LIST>)]

  2 <KEYWORD_CLAUSE>::=[AFFECTED]COMP[FL]=<FAILURE_LIST>
                      |[OTHER]PARAM[ETERS]=<PARAMETER LIST>
                      |PRINT = <LABEL>
                      |TIME = <CONSTANT>[<UNIT>]
                      |RESP[ONSE] = <RESPONSE_LIST>

      3 <FAILURE LIST>::= <FAILURE FUNCTION>
                (<COMPONENT>[<AND_OR><COMPONENT>]*)
                [<AND_OR><FAILURE_FUNCTION
                (<COMPONENT>[<AND_OR><COMPONENT>]*)]*

          4 <FEATURE FUNCTION>::= <IDENDIFIER>

          4 <COMPONENT>::=<IDENTIFIER>

          4 <AND_OR>  ::= &
                        | |

       3 <PARAMETER LIST>::=(<PARAMETER>[,<PARAMETER>]*)

          4 PARAMETER::=<VARIABLE>
                        |<STRING CONSTANT>
                        |<CHAR STRING>
                        |<NUMBER>
       3 <RESPONSE_LIST>::=(<VARIABLE>[,<VARIABLE>]*)
                        |<DONE>

          4 <DONE>::=YES/NO
                    |PROCEED
```

FIGURE 4.5. Syntax of Diagnoses Statements.

The diagnosis statement consists of a header and a sequence of keyword clauses that specify various parameters or other aspects of the diagnosis. The header consists of the keyword DIAG[NOSIS] followed by a mandatory identifier of the diagnosis. This identifier (followed by subscript(s), if it is an array diagnosis) is referenced in the logic statement of the respective tests which select the diagnosis. An optional : symbol may follow a keyword clause. There are five types of optional keyword clauses. Each starts on the left hand side with one of a keyword: COMP, PARAM, PRINT, TIME or RESPONSE. Then there is an = sign and a list of respective parameters on the right hand side. The clauses are optional and may be specified in an arbitrary order. Their objectives are described below.

An affected component clause with the keyword [AFFECTED] COMP[FL] lists a set of component failures, together with the respective mode of failure. The failure mode is in the form of a failure function, such as OPEN, SHORT etc. It is followed in parenthesis by the identifiers of the affected components. The symbols | and & are used between component names to indicate whether any one of the components may have failed or all the components have failed, respectively. The NOPAL processor uses this information to generate an ATLAS program where the more generic tests are conducted earlier, followed by more specific tests. Namely the test that selects a diagnosis which reports failure of any one of a number of components is a candidate for earlier scheduling than a test which reports failure of any one of a subset of these components. If the generic test does not select the diagnosis (i.e. none of the affected components have failed), then the more specific test can be skipped. On the other hand, if a more generic test reports failure of at least one of the components, then the more specific test is evaluated as well as to diagnose the failure more specifically to a subgroup or an individual component. This produces a more efficient program.

As explained below the failure list may be inserted in a message to the operator.

The PARAM clause lists the parameters that must be inserted in the diagnosis message. Specifying the place in the message, where these parameters are inserted, is described later in section 4.4. Parameters may be values of variables determined in the course of testing, constants or character strings. Since a variable in ATLAS, and therefore in NOPAL, may not have a value of a character string, only fixed character strings may be inserted in a message. Several diagnoses can use the same message, where each inserts different parameters in the message.

The PRINT clause identifies the label of the diagnosis message. Messages are specified separately (discussed later in Section 4.4) as a number of diagnoses may use the same message.

The TIME clause specifies when the diagnosis message is to be printed (or displayed) in relation to the start of the test. In most cases it is not necessary to specify the time. Diagnoses that are independent of the selecting test are issued at the start of the test. Diagnoses that report the outcome of tests are issued at the end of the tests. However in some cases it is necessary to issue a diagnosis in the middle of a test with a certain delay after the test has started. The delay is reported as a constant followed optionally by the unit of time, i.e. sec, min or hr. If no unit of time is specified then it is assumed to be sec.

The RESP clause specifies that the operator is required to respond before continuing with the test. The operator may be required in the message to enter a series of values of respective variables, based on observations, or input certain parameters. The list of variables determined in this manner can be specified in the RESP clause. The

operator should then be required in a diagnosis message to input these values, in the order specified, with a carriage return at the end of each value. Alternatively, the operator may be requested in a message to verify certain conditions and when done press the special keys marked Yes or No on the operator's panel. This is specified by YES/NO on the right hand side of the clause. The operator may be required to perform some tasks, and when done press the PROCEED key on the ATE console. This is indicated by the word PROCEED on the right hand side of the clause.

The examples below illustrate use of keyword clauses:

Consider the case where a test determines if the resistances R1 or R2 are shorted (S) or that R3 and R4 are both open (O). Further, the values of these resistances denoted by variables RES1, RES2, RES3 and RES4, as evaluated in the test, are to be inserted in the diagnosis message. To continue the testing, the operator is required to press the PROCEED key. The specification is as follows

```
DIAG D:  COMP = S(R1|R2)|O(R3&R4),
         PARM = (RES1, RES2, RES3, RES4),
         PRINT = M1,
         RESP = PROCEED;
```

The message M1 used with this diagnosis is specified separately. This is discussed in Section 4.4. In this case the message specification would be as follows

```
MESSAGE M1 : 'EITHER R1 OR R2 ARE SHORTED OR'
             'BOTH R3 AND R4 ARE OPEN. THEIR'
             'RESISTENCES ARE: (P1) ##.OHM'
             '(P2)##.OHM, (P3)##.#MOHM AND'
             '(P4)##.#MOHM, RESPECTIVELY.'
             'PRESS PROCEED IF YOU WANT TO'
             'CONTINUE THE TEST';
```

P1, P2, P3 and P4 refer to the first through fourth parameters in the parameter list; each is followed by format specification.

4.4 SPECIFICATION OF MESSAGES

A message statement may be specified anywhere in the specification and is considered as global to the entire main module. A message may be referenced in the PRINT clause of one or more diagnoses. The referenced message is sent to the operator terminal whenever a diagnosis is selected, and if the printer is activated it is also printed. The message may include the affected components and parameters listed in the keyword clauses of the diagnoses that reference the message.

The syntax of the message statements is shown in Figure 4.6.

```
1  <MESSAGE STATEMENT>::=
       <MESSAGE HEADER>[:][<ALIAS_CLAUSE>]<MESSAGE TEXT>;

   2  <MESSAGE HEADER>::= MESS[AGE]<LABEL>

   2  <ALIAS_CLAUSE>::= ALIAS=<LABEL>

   2  <MESSAGE TEXT>::=[TEXT=]<CHAR STRING>[<CHAR STRING>]*
```

FIGURE 4.6. Syntax of Message Statement.

A message statement starts with a statement header, consisting of the keyword MESS[AGE] followed by a mandatory label of the message. The label of the message is referenced in the PRINT keyword clause of diagnoses. An optional : is followed optionally by an alias clause and finally the text of the message.

The ALIAS clause allows references to a message by more than one name. It consists of the keyword ALIAS on the left hand side, a = symbol and a label on the right hand side. The text of the message may be optionally preceded by TEXT=. The message is specified in quotes.

Insertions in the message can be specified or shown in Table 4.1.

(P<N>)	Insert the Nth parameter from the diagnosis parameter clause
(C)	Insert the component list from the diagnosis affected components clause
[R<n>] "#[#]*.[#]*"	Format definition for a variable that is a parameter. If binary, octal or hexadecimal, then preceded by R n, when n is the base
!A	Sound terminal bell
!L	Line feed
!P	Puge feed
!T	Next two positions
!X	Double width characters
!!	Exclamation mark

TABLE 4.1. Insertions in Message Text.

To insert a list of affected components (from the COMP clause in the diagnosis) or the Nth parameter (from the PARM clause) it is necessary to insert in the appropriate place in the message (C) or (P<N>), respectively. If the parameter is a variable it is necessary to folow the (P<N>) with a format specification using the ATLAS format representation, as follows. The # symbols represent digits. The . symbol represents the position of the decimal point. For DIGITAL data type variables it is necessary to precede the format with B for binary, O for octal and H for hexadecimal.

A line in a message may not exceed 40 characters. If necessary it may include a carriage return at the end of the line. Each line must be separately enclosed in quotes.

This is illustrated in the following. Assume a diagnosis

```
DIAG D: COMP = S(R1|R2)&O(R3|R4)
        PARM =  (RES1, RES2, RES3, RES4)
        PRINT = M;
```

The message is assumed

```
MESSAGE M: 'THE POSSIBLE FAILURES ARE (C)'
           'THE RESISTANCE OF R1 IS (P1)##. OHM'
           'THE RESISTANCE OF R2 IS (P2)##. OHM'
           'THE RESISTANCE OF R3 IS (P3)##.# MOHM'
           'THE RESISTANCE OF R4 IS (P4)##.# MOHM';
```

This message may appear as follows:

```
THE POSSIBLE POSSIBLE FAILURES ARE S(R1|R2)&O(R3|R4)
THE RESISTANCE OF R1 IS 10. OHM
THE RESISTANCE OF R2 IS  9. OHM
THE RESISTANCE OF R3 IS  2.0 MOHM
THE RESISTANCE OF R4 IS 12.5 MOHM
```

CHAPTER 5

UUT RELATED DECLARATIONS

Apart from specifying tests (as described in Section 4), the user of NOPAL is also required to provide information that is independent of the tests. Such information is classified in NOPAL based on whether it is related to the UUT or to the ATE. The information is used by the NOPAL system to check completeness of the specification in that tests reference only declared UUT and ATE entities. The UUT and ATE related information is also reported in the reports which document the specification.

In this section we are concerned with the UUT related information. It consists of two types of declarations. First is the declaration of the objectives of testing defined in terms of the component failures that the program must be able to diagnose. The second type consists of UUT connection points that the test program uses. The component failures are listed in the affected component clauses of the diagnoses (Section 4.3). The UUT connecting points are referenced in the stimuli or measurement conjunction statements (Section 4.1). Each component failure and connection point are declared in a separate statement. The order of the statements is immaterial. Sections 5.1 and 5.2 describe the statements for declaring component failures and connecting points, respectively.

5.1 DIAGNOSABLE COMPONENT FAILURE STATEMENTS

Each statement describes a component failure, which can be diagnosed by the test program, and its attributes. By <u>component failure</u> is meant any subpart or aspects of the UUT that the program recognizes when it fails. The user can declare and name each component failure to suit the particular UUT and the test objectives. Further, a component may fail in a number of ways, e.g. it may be shorted, open or out of

tolerance. The mode of failure is referred to as failure function. The user can freely assign a meaningful identifier to each failure function to suit the particular UUT and the test objective. The combination of a component name and the name of the respective failure function define the failure. This corresponds to the notion of failure in the affected component clause of a diagnosis (Section 4.3).

The syntax of a component failure statement is shown in Figure 5.1.

```
1  <COMPONENT FAILURE STATEMENT>::=
              <COMPONENT HEADER>[:]<COMPONENT NAME>[<KEYWORD CLAUSE>]
              [,<KEYWORD CLAUSE>]*

     2  <COMPONENT HEADER>::= COMP[FL][<LABEL>]

     2  <COMPONENT NAME>::= <IDENTIFIER>

     2  <KEYWORD CLAUSE>::= ALIAS = <COMPONENT NAME>
                           |FAIL = <FAILURE FUNCTION>
                           |INDEX = <UNSIGNED INTEGER>

                           |PROT[ECT]=(<LABEL>[,<LABEL>]*)
                           |COMM[ENT] = <STRING>
```

FIGURE 5.1. Syntax of UUT Component Failure Statement.

The statement consists of a header, component name and optional keyword clauses which provide additional information on the diagnosable failure. The information in the keyword clauses is provided primarily for use in documentation of the program.

The header consists of the keyword COMP[FL], which identifies the type of the statement. An optional label may be used. This label refers not only to the component but to the entire component failure statement. As will be shown, it can be referred to in a PROTECT clause to show the precedence of testing for one component failure before the other.

An appropriate name for each component may be selected by the user to reflect the objectives of the test. For instance a component name may be R7, i.e. a resistance in UUT circuit diagram, where the objective of the test program is to diagnose component failure of R7. Sometimes it is necessary that the user select a name for a component failure that is more abstract, such as STD_FREQ, i.e the standard frequency of a transmitter.

The keyword clauses may be specified in arbitrary order. The ALIAS clause is provided to assign more than one component name to a component. The FAIL clause allows the user to further specify the mode of failure through identifying the failure function. The user can choose suitable names for failure functions to suit the UUT and the objectives of testing. This is similar to its use in the affected

component clause of a diagnosis (Section 4.3). Typical failure functions may be: OPEN, SHORT or OUT_OF_TOLERANCE.

Two clauses can affect the order of scheduling of tests by the NOPAL processor. First, the user may assign an INDEX to a component failure to indicate the likelihood of failure of the type of a component. For instance diodes are more likely to fail than resistors. Therefore it would be more efficient to test for failure of a diode before testing for a failure of a resistor. If the program finds that a diode has failed, the operator can discontinue further testing. To take advantage of this the user may assign a lower integer index to diode failures than the resistor failures.

The PROT(ECT) keyword clause also effects the order of scheduling of tests. The user may be concerned that a test to detect a component failure may damage a component in the circuit. For example, if a resistor that is connected to the collector of a transistor is shorted than applying a voltage to the collector may damage the respective transistor. The user may want to specify that the test for the shorted resistor must precede testing for the transistor failure. The label of the "protecting" component failure is given in the PROTECT clause, i.e., the component failure statement for the transistor references the label of the resistor component failure statement in its PROTECT clause. The NOPAL processor will then schedule the test for the "protecting" component failure before the test for the "protected" component failure.

Finally a COMMENT clause is provided, purely for documentation, where additional information may be provided to document further a diagnosable failure.

Following is an example of a component failure statement

```
COMP 10:  STD_FREQ
          FAIL = OUT_OF_TOL
          INDEX = 1
          PROT = (11,12);
```

STD_FREQ is the standard frequency of a transmitter UUT. It's failure function named OUT_OF_TOL indicates that the transmitting frequency is outside the allowed tolerance limit. The INDEX indicates that this is the highest priority for testing. The PROT clause says that the test for this component failure should not be conducted if component failure 11 and 12 have been diagnosed as having failed.

5.2 UUT CONNECTING POINT STATEMENTS

A declaration statement must be provided for every UUT connecting point referenced in stimuli or measurement conjunction statements in test specifications. A possible approach is to compose statements for all the UUT connection points that are accessible to the ATE. The NOPAL

processor checks that all UUT connecting points referenced in conjunction statements have been declared. It issues an error message for every connecting point that has been referenced but not declared.

The syntax for UUT connecting point declaration statements is shown in Figure 5.2.

```
1  <UUT CONNECTION STATEMENT>::=
        <CONN STATEMENT HEADER>[:]
             <CONN_NAME>[(<REPETITIONS>[,<REPETITIONS]*)]
             [<KEYWORD_CLAUSE>[,<KEYWORD CLAUSE>]*;

    2  <CONN HEADER>::= UUT_P[NT] [<LABEL>]

    2  <CONN_NAME>::= <IDENTIFIER>[,<IDENTIFIER>]

    2  <REPETITIONS>::= <UNSIGNED INTEGER>

    2  <KEYWORD CLAUSE>::=  ALIAS = <IDENTIFIER>
                           |CONN[ECTOR] = <LABEL>[,<LABEL>]
                           |LIMIT = <LIMIT LIST>
                           |COMMENT = <CHAR_STRING>

        3  <LIMIT LIST>::= [<UNIT>]
             [,[<MAX>][,[<MIN>]],[<REFPPNT>]]

             4  <MAX>::= <CONSTANT>

             4  <MIN>::= <CONSTANT>

             4  <REF_PNT>::= <LABEL>
```

FIGURE 5.2. Syntax of UUT Connection Point Declaration Statement.

The UUT connecting point statement consists of a header, the name of the connecting point and optional keyword clauses that provide additional information which is primarily used for documentation purposes.

The header consists of the keyword UUT_P[NT] followed by an optional label. The connection point name may typically be that of the respective connection point name in the circuit diagram of the UUT. As discussed in Section 2, connection points may be scalars or arrays. If they are scalars, namely only one connection point corresponds to the name, then it is not necessary to specify the number of repetitions of the connection point. If the connection point is of an array type, it corresponds to a number of connection points arranged in one or more dimensions. When an array connection point is referenced in a conjunction, it must be followed in parentheses by subscripts for all of the dimensions of the array. It is necesary to provide in the

respective UUT connecting point declaration the number of element connection points in each dimension. The subscript statement in the test must show the same number of repetitions. For instance, consider a vector connecting point TP with 10 element connections. In a conjunction statement the connection point may be referenced as TP(I), where I is a subscript defined as

```
I = SUBS('---',10);
```

In the UUT connecting point declaration the corresponding connecting point is shown as

```
UUT_P:TP(10);
```

The connection points, whether of scalar or array type, must be mapped into physical ATE connection points in a separate connection file using EQUATE ATLAS definitions. The connection file is not deeded during the use of the NOPAL system; only at EQUATE ATLAS compilation time. This is further discussed in Section 8.

The specification of the connection name may be optionally followed by keyword clauses that contain further documentation of the connection point as follows.

An ALIAS keyword clause allows assigning an alternative name to the connection point.

The CONN[ECTOR] keyword clause allows the user to provide information on the type of UUT connector used and/or the pin location on the connector. Both the connector type and pin location may be identified.

The LIMIT clause allows the user to specify limitations on the magnitude of the signal applied to the connecting point. This is provided in the form of a limit list which consists of four elements, separated by commas. The first element is the physical unit of the applied signal, such as V (for volts) or A (for Amperes). Next the maximum allowed magnitude of the signal followed by the minimum allowed magnitude of the signal if any. The last element is the reference point in regard to which the magnitude of the signal is measured. An element may be omitted, indicated by null information between the separating commas.

Finally, additional information may be provided in a COMMENT clause.

The following example illustrates a UUT connection point declaration statement:

```
UUT_P: TP(10)
       ALIAS = INPUT
       CONN = COAXIAL
       LIMIT = (V,100,GND);
```

It means that a UUT connecting point TP, of array type, with 10 element connections, may also be referred to as INPUT. It is a coaxial type of connector. The maximum voltage applied to this connection point is 100 volts to ground.

CHAPTER 6

ATE RELATED DECLARATION STATEMENTS

The user of NOPAL can refer to functions and UUT connection points without being concerned about how the functions are implemented or how the UUT connection points are connected to the ATE. The stimuli and measurement functions and the ATE connections are dependent on the facilities built-in into the ATE and are therefore considered as ATE related. All the functions that are used in the test specifications, including non ATE related functions, must be declared in ATE related statements. This includes: 1) the functions used in conjunction and assertion statements, and 2) the failure functions used in diagnoses and UUT component failure declarations. These ATE related declaration statements contain informational items which are used as a basis for checking the consistency of the specification and are also incorporated in the produced documentation.

The two types of ATE related statements, for declaring functions and for declaring ATE connections, are described in Sections 6.1 and 6.2, respectively.

6.1 FUNCTION DECLARATION STATEMENTS

Functions that are referenced in test statements must also be declared in function declaration statements. Except for failure functions and the functions that are built-in into NOPAL or ATLAS, all other functions must be separately defined. Basic stimuli and measurement functions may be defined directly in the REQUIRE clause of the declaration statement of the function. Otherwise, a function may be defined directly in EQUATE ATLAS or in a secondary NOPAL module, using the NOPAL processor to generate an ATLAS procedure for the function. Defining functions is further discussed in Section 8. All the programs generated based on the main and secondary NOPAL modules and the ATLAS function programs, must be compiled jointly by the EQUATE ATLAS compiler.

Figure 6.1 shows the syntax of an ATE function declaration statement.

```
1     FUNCTION DECLARATION> ::= <STATEMENT HEADER> [:]
            <FUNCTION NAME>  [, <KEYWORD CLAUSE>]* ;
      2     <STATEMENT HEADER> ::= FUNC[TION] [<LABEL>]
      2     <FUNCTION NAME> ::= <IDENTIFIER>
      2     <KEYWORD CLAUSE> ::= ALIAS = <IDENTIFIER>
                  | TYPE = <FUNCTION USAGE>
                  | #PINS = <UNSIGNED INTEGER>
                  | [PARA[METER] = [(<PARAMETER ATTRIBUTES>)]*
                  | VALUE = <DATA TYPE>
                  | REQ[UIRE] = <ATLAS ACTION STATEMENT TEMPLATE>
                  | COMM[ENT] = <CHAR_STRING>
            3     <FUNCTION USAGE> ::= S|M|E|F
            3     <PARAMETER ATTRIBUTES> = <IDENTIFIER>,
                     <SOURCE OR TARGET>, [<DATA TYPE>]
                  4   <SOURCE OR TARGET> ::= S|T
```

FIGURE 6.1. Syntax of Function Declaration Statement.

The function declaration statement consists of a statement header, a function name, and a series of optional keyword clauses. The header consists of the reserved word FUNC[TION] followed by an optional label. The function name is the same that is used in the function references in conjunction or assertion statements, or in diagnosis statements and UUT component failure declarations.

All the keyword clauses are optional. They are used only in certain cases, as described below. The ALIAS keyword clause is used to give an alternative name to the function by which it may be referenced. This allows using different function names for the same function.

The TYPE clause describes the class of usage of the function. It is provided optionally, purely as part of the documentation of the function. There are four function usage classes:

S - Stimuli. The function is used only in stimuli conjunctions.

M - Measurement. The function is used only in measurement conjunctions.

E - Evaluation. The function is used only in assertions and performs a purely computational task.

F - Failure. The function is used only as a failure function in diagnoses and UUT failure declarations.

The #PINS clause is used only when the function type is of S or M ; namely when the function employs stimuli or measurement devices. In these cases the number of connecting points that are used by the stimuli

or measurement devices is specified in this clauses. This must also be the number of test points specified on the left hand side of the conjunction where the function is referenced. If provided, the information on the number of pins used is included in the generated program documentation.

The PARA[METER] (or PARM) keyword clause may be repeated for each argument of the function, ordered by the argument's position. It provides information on the argument's name and data type. The parameters should be named P01, P02, etc. according to the position of the argument in the argument list. The primitive data types are DECIMAL, INTEGER, DIGITAL and BOOLEAN. The default data type is DECIMAL. The user can also refer by name to a tree-like data structure with the data types of FILE, GROUP and RECORD. The use of data structures or arrays as parameters is further discussed in Section 7. In the present NOPAL system a variable may not denote a character string, but a character string may serve as an argument of some built-in functions. The specification of the parameters in this version of NOPAL is purely for documenting the function and no checks are conducted.

The VALUE clause gives information on data type of the returned argument.

The REQUIRE clause is used to define a basic stimuli or measurement function which consists of application of a single device. A function that is defined in the REQUIRE clause needs not be defined elsewhere in a secondary module or directly in EQUATE ATLAS. The definition of the function consists of providing a template of an ATLAS action statement. It must start with an ATLAS action verb. The only verbs allowed, however, are:

APPLY - for applying a signal through use of a stimuli device.

MEASURE - for measuring a signal through use of a meter device.

MONITOR - Same as measure, except the measurement is repeated and displayed on the operator console, until the operator presses the PROCEED key. The last value measured is returned by this function.

DO DIGITAL - for applying a digital stimuli and/or measurement device.

SET_UP - for set up and application of dual channel devices.

The definition of a function in this manner is further discussed below and in Section 8. A set of templates for the EQUATE ATE currently in

use are given in an appendix. Additional templates mya be entered in the respective file (see Section 9).

Finally a COMMENT clause may be used for any further documentation of the function.

Following is an example of use of an ATE function declaration statement. Assume that a function named FM_GEN has the effect of applying a frequency modulated signal as stimulus to a UUT. It requires four arguments: P01 - the carrier frequency, P02 - the applied power, P03 - the modulating frequency and P04 - the maximum frequency deviation. DIST_ANLY is a function that measures distortion with three parameters: P01 - the distortion in %, P02 - the power level and P03 - the carrier frequency. Below is an example of a stimuli part of a test where the function is used, followed by the measurement part which measures the distortion

```
TEST TN;
STIM;
CONJ: <TP1,GND> = FM_GEN (11.5 MHZ, -27 DBM,20 KHZ, 2 KHZ);
MEAS;
CONJ: <TP2,GND> = DIST_ANLY(DIST %,0 DBM,11,5 MHZ)
                  TARGET:DIST;
```

The declaration of the FM_GEN function is as follows

```
FUNC: FM_GEN
      TYPE = S,
      #PINS = 2,
      PARAM = (P01,S,DEC),
      PARAM = (P02,S,DEC),
      PARAM = (P03,S,DEC),
      PARAM = (P04,S,DEC),
      REQUIRE = APPLY FM_SIGNAL RFD
                CAR_FREQ (P01) MHZ,CAR_POWER (P02) DBM,
                SINE-WAVE, MOD-FREQ(P03) KHZ,
                FREQ_DEV(P04) KHZ
                CNX COAX(CNX01);
```

The REQUIRE clause is based on a template for signal generations in the NOPAL system's library. The available options were selected according to the need of the test. The template is stored in the NOPAL system library in the form of a syntex diagram using EBNF as follows:

```
APPLY {AM_SIGNAL|FM_SIGNAL} {RFA|RFB},
      CAR_FREQ( ) <HZ_DIM>, CAR_POWER( )DBM,
      {EXT,|{SINE-WAVE|SQUARE-WAVE|TRIANGLE-WAVE
      |SAWTOOTH-WAVE, {POS-SLOPE|NEG-SLOPE}
      |COMPLEX-WAVE, STIM( ) |RANDOM-NOISE}
      MOD-FREQ ( ) <HZ-DIM>,}
      {MOD-DEPTH( )PC|MOD-AMPL ( ) PC
      |MOD-INDEX( )|FREQ-DEV( )<HZ-DIM>}
      [,CNX{COAX( )|SELF-TEST}]$
```

The user needs to select the appropriate options to form the REQUIRE clause. Templates include summary information on the legal ATE connection points. The objective is to provide the user of NOPAL sufficient information on defining basic functions through the templates. Further information on the device and connections require referencing the appropriate diagram of the device in the ATLAS manual. The REQUIRE clause is checked by the NOPAL system that it conforms with a respective template stored in the library of the system. Note that parameter numbers (P<n>) and test point numbers (CNX<n>) are inserted in the empty brackets in the template. <n> is a two digit integer. The current template library is listed in an appendix.

Similarly, the DIST_ANLY function if declared as follows.

```
FUNC:   DIST_ANLY
        TYPE = M,
        #PINS = 2,
        PARAM = (P01,T,DEC),
        PARAM = (P02,S,DEC),
        PARAM = (P03,S,DEC),
        VALUE = (DEC),
        REQUIRE = MEASURE(DISTORTION (P01) PC),
                  FM-SIGNAL, POWER MAX (P02) DBM,
                  CAR-FREQ (P03) MHZ
                  CNX HI(CNX01) LO(CNX02);
```

The REQUIRE clause can contain only statements for the action verbs APPLY, MEASURE, MONITOR and DO DIGITAL TEST. In the case of dual channel devices, the APPLY template must be preceded by a SETUP template. The other action verbs in ATLAS are procedural in nature; namely they are dependent on the sequence of statements, and therefore can not be used in the NOPAL nonprocedural language.

The scale factors of the units of the arguments of the function, used in conjunction statements and in the respective REQUIRE clause, must agree.

Additional templates may be added to the library of templates to reflect changes in the ATE. Thus it is easy to custom tailor the function template library to a specific ATE configuration.

6.2 ATE CONNECTION POINT DECLARATION STATEMENT

The choices of ATE connection points that correspond to the UUT connecting points depends on the devices selected for applying stimuli or measurements, on the ATE interface used (dedicated, programmed, and whether switchable) and on a variety of other details of the operation of the ATE. The choice of devices and ATE connection points can be performed separately from the tasks of specifying a test program in NOPAL and the generation of the corresponding ATLAS program. The user can refer in NOPAL to stimuli and measurement functions without knowing how they perform. The details of the functions that utilize stimuli or measurement devices may be defined separately in a NOPAL secondary module or directly in ATLAS. The ATE connection points that correspond to UUT connecting points must be defined in a separate connection file. The programs that define functions, and the connections file, are needed only at the time of use of the ATLAS compiler. Thus the declarations of the ATE connection points are not needed during test program specification and therefore, supplying the ATE connection points declarations is optional. If these statements are included they are incorporated in the generated documentation but no checks are based on these statements. The ATE connection points information maybe useful

for mapping the UUT connection points into the ATE connection points in the connection file. The connection file is described in Section 8.2.

An ATE test point declaration statement essentially provides a name or number of an ATE connection point and the name of the corresponding UUT connecting points. These statements map ATE connections into UUT connection points, the reverse of the connection file. The syntax is shown in Figure 6.2.

```
<ATE CONNECTION DECLARATION STATEMENT> ::=
        <STATEMENT HEADER>[:]<IDENTIFIER>
        [(<SUBSCRIPT RANGE LIST>)][,<KEYWORD CLAUSE>]*;
    2   <STATEMENT HEADER> ::= ATE_P[NT][<LABEL>]
    2   <SUBSCRIPT RANGE LIST> ::= <UNSIGNED INTEGER>
                [,<UNSIGNED INTEGER>]*
    2   <KEYWORD CLAUSE> ::= ALIAS = <IDENTIFIER>
                    | UUT_P[NT] = <UUT CONNECTION LIST>
                    | COMM[ENT] = <CHAR STRING>
           3    <UUT CONNECTION LIST> ::= (<IDENTIFIER>
                    [(<LIST OF INDICES>)][,<IDENTIFIER>
                    [(<LIST OF INDICES>)]]*)
              4     <LIST OF INDICES> ::= <UNSIGNED INTEGER>
                                [,<UNSIGNED INTEGER>]*
```

FIGURE 6.2. Syntax of ATE Point Declaration Statement.

The ATE connecting point declaration statement consists of a header, the identity (name) of the ATE connection points and optional descriptive keyword clauses. The header uses the reserved word ATE_PNT followed by an optional label. The name of the ATE connection point must be provided next. If the ATE connection point is of an array type, then the name of the ATE connecting point is followed by the ranges (i.e the number of element connections) along each dimension of the array.

The optional ALIAS keyword clause provides an alternative name for the ATE connecting point. The UUT_PNT keyword clause identifies the corresponding UUT connection points. If the UUT connecting points are members of an array, then the indices of the point must follow the name in parenthesis.

Finally, additional information may be provided in a COMMENT clause.

Following is an example of an ATE Point declaration.

```
ATE_PNT:PROBE,
        UUT_PNT = (TP1,GND);
```

This declaration statement refers to the PROBE of the ATE which is a two point connection denoted in the test specification as TP1 and GND.

CHAPTER 7

DATA DECLARATION STATEMENTS

In most test program specifications, the user refers only to elementary (i.e. non structured) variables of primitive data types. In these cases it is not necessary to provide data declaration statements at all, and the NOPAL processor determines the data types, dimensionality and ranges of variables based on the use of these variables in statements in the specification. The user may however, optionally provide data declaration statements which override the automatic data type, dimensionality and range determinatin. The NOPAL processor conducts checks on consistency of data type and dimensionality of variables and issues error messages if a conflict is detected. The data types of variables are reported in the cross reference report. Section 7.1 discusses the cases where data declaration statements are optional. In more complex cases, NOPAL allows the declaration and use of data structures. This is discussed in Section 7.2.

7.1 ELEMENTARY VARIABLES WITH PRIMITIVE DATA TYPES

The primitive data types in this version of NOPAL are: DECIMAL, INTEGER, DIGITAL and BOOLEAN. The automatic determination of the data types of a variable is based first on the respective declaration statement, if any. The data type of arguments of a function, including the result argument may be provided optionally in the respective ATE function declaration statement (Section 6.1). Otherwise, DECIMAL data types are selected for variables in arithmetic expressions, DIGITAL data types are selected for variables in digital expressions, and BOOLEAN data types are selected for variables in boolean expressions. These expressions are recognized by the use of respective distinct classes of operators (except in the case that only = and ^= are used, which are common to DECIMAL and DIGITAL). The NOPAL processor "propagates" data types. Namely, once the data type of a variable is determined then the other variables in respective expressions or assertions, where the variable with the determined data type is used, will have the same data

type. Thus variables in an arithmetic assertion have either DECIMAL or INTEGER data type, variables in a digital assertion have DIGITAL data types, and variables in boolean expressions have BOOLEAN data types. The NOPAL DECIMAL, INTEGER, and DIGITAL data types are implemented by cooresponding data types of ATLAS. NOPAL BOOLEAN data type is implemented by ATLAS DIGITAL data type. Any discovered conflicts are reported to the user. Finally, the remaining variables without a data type are assigned the DECIMAL data type. Note that a data type declaration is needed if the user wants to assign an INTEGER data type to some variable. Declared variables are considered global to the entire module. The syntax for elementary (non structured) variable data declaration statements is shown in Figure 7.1.

```
1   <VARIABLE DATA DECLARATION STATEMENT>::=
        <STATEMENT HEADER>[:]<LIST OF VARIABLES>[:]
        <DATA TYPE>[<ARRAY SPEC>];
        2   <STATEMENT HEADER>::=DCL[<LABEL>]
        2   <LIST OF VARIABLES>::=<IDENTIFIER>[,<IDENTIFIER>]:
        2   <DATA TYPE>::=DEC[IMAL]
                        | DIG[ITAL]
                        | INT[EGER]
                        | BOOL[EAN]
        2   <ARRAY SPEC>::=  ARRAY (<RANGE>[,<RANGE>]*)
            3 <RANGE>::= <UNSIGNED INTEGER> | *
```

FIGURE 7.1. Syntax of Data Declaration Statement For Elementary Variables.

The statement starts with a header consisting of the reserved word DCL, followed optionally by a label and :. The variable names are next listed, followed by the assigned data type and array declaration. Thus a number of variables may be declared in a single statement. The allowed primitive data types are DECIMAL, INTEGER, DIGITAL or BOOLEAN. If the variable(s) are of array type, then the ARRAY reserved word is used followed in parenthesis by the ranges of the respective dimensions, i.e. the numbers of elements along the respective dimensions. If the range is unknown (defined by an END_<SUBSCRIPT> variable) then the range is denoted by *.

Examples of elementary variable data declaration statements are:

```
DCL   A, B, X : INTEGER;
DCL   Y, Z : DIGITAL ARRAY (5,7,9);
```

A, B and X are integer scalars. Y and Z are three dimensional arrays of 5 * 7 * 9 elements.

7.2 DATA STRUCTURES AND HIGH LEVEL DATA TYPES

Data declaration statements can be used in NOPAL for declaring tree like data structures, somewhat similar to COBOL or PL1. A data structure may consist of aggregates of variables organized in a specific way. These data structures are useful in communications with external devices that store, consume or produce masses of data. The user of NOPAL is not concerned with input/output activity. Normal communication with the ATE operator is achieved by specifying diagnoses and respective messages and operator responses. However, sometimes it is necessary to access or store data structures from, or to, secondary mass storage devices. Similarly, it may be necessary, especially in digital testing, to send data structures to stimuli device, and/or receive such data structures from a measurement device. An entire data structure may then be referred to by name and operated upon by a function defined for this purpose, i.e. be an argument of a function.

The syntax of a data structure declaration statement is shown in Figure 7.2.

```
1 <DATA STRUCTURE DECLARATION STATEMENT>::=
    <STATEMENT HEADERS>[:]<NODE DECLARATION>
    [<NODE DECLARATION>]*;
   2 <NODE DECLARATION>::=
     <LEVEL NUMBER><NODE VARIABLE><DATA TYPE>[<ARRAY SPEC>]
        3   <NODE VARIABLE>::= <IDENTIFIER>
        3   <LEVEL NUMBER>::= <UNSIGNED INTEGER>
        3   <DATA TYPE>::= <DATA TYPE ROOT NODE>
                          |<DATA TYPE INTERMEDIATE NODE>
                          |<DATA TYPE TERMINAL NODE>
            4   <DATA TYPE ROOT NODE>::= FILE
                                       | GROUP
                                       | RECORD
            4   <DATA TYPE INTERMEDIATE NODE>::= GROUP
                                               | RECORD
            4   <DATA TYPE TERMINAL NODE>::= DECIMAL
                                           | INTEGER
                                           | DIGITAL
                                           | BOOLEAN
```

FIGURE 7.2. Syntax Diagram of Data Structure Declaration Statements.

The notion of a data tree structure is that the entire structure is given a name associated with the root node. Its major constituents are represented by nodes at end of branches emanating from the root node. The nodes on each level are ordered from left to right in accordance with their position in an external device. Additional levels of the tree are used for constituents of each node at the next lower level. The terminal nodes in a tree are the elementary variables.

The data structure declaration statement describes the entire data tree structure. It consists of a statement header similar to that used for elementary variable declaration statements (DCL), followed by a series of declarations of the nodes in the tree structure. The individual node declarations must be ordered starting from the root of the tree and traversing all the nodes, depth first, from left to right. Each node declaration must be preceded by the level number of the node in the tree, with the root node having a level number 1, its immediate constituents level number 2, etc.

An individual node declaration is similar in syntax to the elementary variable declaration statement. Following the level number is a name associated with the node, its data type and an array specification, if an array.

The data types associated with terminal nodes in the tree are the same as those used for elementary variables: DECIMAL, INTEGER, DIGITAL and BOOLEAN.

Three data types, FILE, GROUP and RECORD may be associated with a non terminal node of a tree. Their meaning is as follows. The FILE data type may be associated only with a root node. Its meaning is that the entire data structure is on a secondary storage device, namely it is given as an input file or is to be produced as an output file on a secondary mass storage device, such as disk or tape. The accessing or storing of the data in a file, as needed, is handled by the NOPAL processor and the user need not be concerned with it. The GROUP and RECORD data types may be assigned to any intermediate node in the data tree, i.e. non terminal nodes. They may be assigned to the root node only when the data structure is not an input or output file, namely where the FILE data type is inappropriate. The distinction between between GROUP and RECORD data type is that the latter indicates a structure, consisting of an aggregate of variables, which must be moved as one unit because of the device characteristic. It is similar to RECORD in PL1 and COBOL. The appropriate function where a RECORD structure serves as an argument must be defined separately. This distinction between RECORD and GROUP is not important in the present version of NOPAL because of limitations of ATLAS, where individual variables are referenced in INPUT and WRITE ATLAS statements.

Following is an example of an input/output file declaration statement.

```
DCL   1  F : FILE,
         2  G1 : GROUP,
            3  R : RECORD ARRAY (10),
               4  VAR1 : DIGITAL,
               4  VAR2 : DIGITAL,
         2  G2 : RECORD ARRAY (*),
            3  VAR3 : DECIMAL;
```

This is an input/output file consisting of two substructures, G1 followed by G2. G1 is a scalar. It consists of an array of 10 records, each containing the variables (terminal nodes) VAR1 and VAR2. G2 is a record array of variable number of repetitions. The subscript and range for this dimension must be separately specified. Each G2 record contains a single VAR3 decimal variable.

CHAPTER 8

DEFINITION OF FUNCTIONS AND CONNECTIONS FILE

When specifying a test program, a user may use a symbolic name for a function without being concerned about how the function is implemented. It is sufficient that the user know the name of the function and the meaning of the arguments of the function. Similarly, the user need not be concerned at the time of specifying tests with the physical connections between the UUT and the ATE. It is sufficient to refer to UUT connections by symbolic names. At some time however, it is necessary to define the functions and ATE connection points that are used.

Function programs may be prepared centrally and placed in a library, available to the user community, or they may be prepared for individual test programs, concurrently or after the main test program has been developed. A connection file must be prepared for each test program concurrently or after specifying the test program. These tasks sometimes involve detailed knowledge of the ATE. They can be performed by a specialist in the ATE, but also frequently by a user not knowledgeable in the particular ATE and in the ATLAS test language. This section describes the conventions that need to be followed in these tasks.

8.1 DEFINITION OF FUNCTIONS

As discussed in Section 6.1, basic stimuli and measurement functions may be defined in the REQUIRE clause of the respective function declaration statement. More complex functions can be defined in NOPAL, in a secondary module. It is also possible to define functions directly in ATLAS. The ATLAS function procedures, whether generated by NOPAL or manually written directly in ATLAS, must be placed in an ATLAS file named SYSLIB which is included with the ATLAS program generated from the main NOPAL module in the ATLAS compilation (i.e. the generated main ATLAS program would have a corresponding ATLAS INCLUDE

statement). Due to limitations in EQUATE ATLAS, the user must observe certain naming conventions for variables when either defining a function in ATLAS or in NOPAL. These variable naming conventions are described below.

In defining a function, the user must refer to the variables which correspond to the arguments of a function as P01, P02,... in accordance with their position in the function argument list. The connections must be referred to as CNX01, CNX02, etc. in accordance with the position of the UUT connection in the connection list in the respective conjunction. The list of arguments of a stimuli function should not contain any result arguments. The list of arguments of a measurement function contains input arguments and result arguments. The list of arguments of purely computational functions, used in assertions, have only input arguments. The result argument is not included in the function arguments. It must be referred to by the name RES.

For example, consider the two conjunction statements:

CONJ: <T1,GND> = METER(V1 V,100) TARGET V1;

CONJ: <T1,GND> = METER (>10 V,100);

The first conjunction has two arguments. The first argument is also the result argument V1. It is the value measured by the METER function. It must be referred to as P01 in the definition of METER. The second argument is the constant 100, which is used to indicate the range of values to be measured. It should be denoted by P02. In the second conjunction, the result argument is implicit. It is compared whether it is greater than 10V; this determines whether the test passes or fails. This comparison is handled by the NOPAL system and does not effect the definition of the function. Thus in the above example P01 would correspond to V1, P02 to 100,CNX01 to T1 and CNX02 to GND.

Consider next the example of the assertions:

ASRT: X = FUNC(Y,Z,---) TARGET:X;

ASRT: X > FUNC(Y,Z,---);

In the first definition type assertion, the result argument is not provided in the list of arguments of the function. It defines the value of X. In the second condition type assertion, the result argument is compared if smaller than X to determine test outcome. In both cases the result argument is not included in the list of arguments of the function. In writing the definition of a function the user must refer to the result argument by the name RES. Functions used in assertions must have one result argument.

Futher, the variables corresponding to input and result arguments must be qualified by prefixing the name of the function. Thus the names of the variables and connections that denote arguments in the above

examples are:

METER.P01, METER.P02, METER.CNX01, METER.CNX02.

FUNC.P01, FUNC.P02,---- ,FUNC.RES.

The above rules apply whether the function is defined in a secondary NOPAL module or in ATLAS. There is no need to prefix the name of the variables with the function name when defining a function in a REQUIRE clause.

8.1.1 Definition Of A Function In A Secondary NOPAL Module.

A secondary NOPAL module is similar in organization, syntax and semantics to the main module with the following differences.

A module header statement (see Section 3) is used with the keyword MODULE: followed by a chosen name. For example,

MODULE: SYSLIB

A number of functions can be defined in a secondary module. Each function must start with a function header statement using the syntax of Figure 8.1, and be followed immediately by the statements that define the function.

```
1 <MODFUN HEADER STATEMENT>::=
    MODFUN[:]<IDENTIFIER>[,[INPUT](<INPUT ARGUMENTS>),]
              [,RESULT(<RESULT ARGUMENTS>)]
              [,CONN(<CONNECTION ARGUMENTS>]
  2 <INPUT ARGUMENTS>::=
      <IDENTIFIER>: <DATA TYPE>
      [,<IDENTIFIER>: <DATA TYPE>]*
  2 <RESULT ARGUMENTS>::=
      <IDENTIFIER:><DATA TYPE>
      [,<IDENTIFIER>:<DATA TYPE>]*
  2 <CONNECTION ARGUMENTS> = <IDENTIFIER>[,<IDENTIFIER>]*
    3 <DATA TYPE>::= DEC[IMAL]
                   | INT[EGER]
                   | DIG[ITAL]
                   | BOOL[EAN]
                   | FILE
                   | GROUP
                   | RECORD
```

FIGURE 8.1. Syntax Diagram of a Function Header Statement.

The header statement starts with the MODFUN reserved word, followed by the name of the function, the reserved word INPUT and a list of input arguments, the reserved word RETURNS and a list of result arguments and the reserved word CONN and a list of connection names. Each of the argument names is followed by its data type. The variables denoting arguments and connection points must be named as discussed above. For example:

```
MODFUN: FUNC, INPUT (FUNC.P01:DEC, FUNC.P02:DEC),
                    RESULT (FUNC.RES:DEC);
MODFUN: METER, INPUT (METER.P02:DEC);
                    RESULT(METER.P01:DEC),
                    CONN (METER.CNX01, METER.CNX02);
```

Each MODFUN consists of one test only. The test is similar to tests in the main NOPAL module, and may consist optionally of stimuli and measurement conjunctions, assertions, logic, diagnoses and messages. There may be UUT, ATE and data declaration statements, that are global to the entire module. These may include declarations and definitions of additional functions.

The END statement must be used at the end of the secondary module.

8.1.2 Definition Of Functions Directly In ATLAS.

A function definition maybe an ATLAS Procedure which is included with the main program in the ATLAS compilation. Knowledge of EQUATE ATLAS is needed to compose function procedures directly in EQUATE ATLAS. The version of the EQUATE ATLAS compiler may not handle arguments in the procedure call. Therefore the NOPAL processor generates the appropriate statements for passing the arguments to the function procedure. The user must therefore omit the arguments in the procedure defining statement. The variables that denote the arguments must however, be declared. This is illustrated in the following example of definition of a procedure in ATLAS for an impedance measurement function.

```
DEFINE PROCEDURE 'OHM-METER'$
   DECLARE DECIMAL 'OHM-METER.P01'
                   'OHM-METER.P02'$
MEASURE (RES 'OHM-METER.P01' OHM),IMPEDANCE,
         REF VOLTAGE 'OHM-METER.P02' V,
         CNX HI OHM-METER.CNX01
             LO OHM-METER.CNX02 $
END 'OHM-METER' $
```

This function may be referenced in NOPAL as:

```
MEAS;
CONJ:<T1,GND>::= OHM_METER(R1 OHM,V1 V)TARGET:R1;
```

The code generated by the NOPAL processor to use the function would be:

```
'OHM-METER.P2' = 'V1" $
DEFINE     'OHM-METER.CNX1', T1 $
DEFINE     'OHM-METER.CNX2', GND $
PERFORM 'OHM-METER' $
'R' = 'OHM-METER.P1' $
```

Step numbers 0 through 999 are reserved for use in Atlas function procedures.

Thus as long as the user adheres to the above variable naming convention, the NOPAL processor would generate appropriate code to pass arguments and connections and call the function procedure.

8.2 UUT-ATE CONNECTION FILE

The UUT-ATE connection file uses the ATLAS DEFINE MACRO statements to give each UUT symbolic connection name its equivalent ATE connection name or value. These statements are placed in a file named CNX<SPEC NAME>. <SPEC NAME> consists of the first three characters of the name of the main module. This file is included with the other ATLAS programs in the ATLAS compilation.

In using analog stimuli and measurement devices, an ATE connection point must be identified, individually, for each UUT connection point. It is necessary to compose one DEFINE statement for each UUT connecting point. The syntax of a DEFINE statement for relating the UUT and ATE connecting points is shown in Figure 8.2.

```
1  <UUT-ATE ATLAS DEFINE>::=
          DEFINE<UUT CONNECTION>,<ATE CONNECTION>$
   2  <UUT CONNECTION>::= '<IDENTIFIER>'
   2  <ATE CONNECTION>::= [<LETTER>]*
                          | <ATLAS ARITH EXPR>
```

FIGURE 8.2. Syntax of ATLAS DEFINE Statement for UUT-ATE Connection.

The ATE connection can be expressed as a string of letters specifying a name or a number of an ATE connection. Alternately an arithmetic expression may be expressed using ATLAS syntax with a value that corresponds to the ATE connection. The ATLAS syntax of an arithmetic expression differs from NOPAL in that variable names must be enclosed in quotes. The arithmetic expression is useful to map array connections. For instance, let TP be a UUT connection of array type with 10 elements. The corresponding ATE connections are 221 to 230. The DEFINE statement would be:

```
DEFINE 'TP(I)', 220 + 'I'$
```

Legal ATE connection points are shown in the appendix with the templates for REQUIRE clauses used to define basic stimuli and measurement functions.

The situation is more complicated in using digital stimuli and response. In this case it is necessary, in addition to the above, to declare a vector which stores the ATE connection point values for digital UUT. The connection file must then include also a declaration of the vector (or "list" in ATLAS) and an ATLAS FILL statement which assigns the ATE connection point. For example, consider the case where TP1 and TP2 are two series of connection points for digital stimuli and response which use test points 51-58 and 161-168 respectively. Then the connection file will have to include:

```
DEFINE 'T1', 'STIM_CNX'$

DEFINE 'T2', 'RESP_CNX'$

DECLARE DECIMAL LIST 'STIM_CNX'(8),'RESP_CNX'(8)$

FILL  'STIM_CNX', 'RESP_CNX',
(1)       51          161
(2)       52          162
(3)       53          163
(4)       54          164
(5)       55          165
(6)       56          166
(7)       57          167
(8)       58          168$
```

CHAPTER 9

OPERATING THE NOPAL PROCESSOR

The NOPAL processor is currently operational using the Digital Equipment Corporation VAX 11 series computers, under the VMS operating system. It requires also the use of the PL/I compiler .

To use the processor it is necessary first to assign a disk device for the files. For example, the assignment of a disk device DRO is as follows:

```
$NOPAL:== $DRO:[NOPALBOT]NOPAL
```

This command may also be defined as part of the system call for the NOPAL processor.

The user then composes a specification which serves as the input file to NOPAL.

The NOPAL processor may then be initiated by keying in

```
NOPAL <INPUT FILE NAME>[/<OPTION>]*
```

The options for running the NOPAL processor are shown in Table 9.1. The default options are underlined. The options allow choice of reports, as well as stopping the processor before specified phases.

Option	Description
SOURCE1/NOSOURCE1	Listing of input specification, with numbered lines for later reference.
XREF1/NOREF1	Cross reference report of identifiers and labels and their attributes.
XREF2/NOXREF2	Cross reference reports of tests, failures, connection points and functions.
SOURCE2/NOSOURCE2	Reformatted specification.
SEQ=<N>/NOSEQ	Whether to continue processor through sequencing.
N=1	All flow chart and dependencies reports.
= 2	No internal dependencies reports.
= 3	Only inter-test flow charts and dependencies.
= 4	Omit cycle elimination trace report.
= 5	Show only inter-test order vector.
= 6	No reports.
CODE/NOCODE	Whether to continue processor through code generation phase.
TRACE/NOTRACE	Whether to produce in the ATLAS program ATLAS record statements for results of measurements and tests, to obtain a trace helpful in debugging.
DEBUG=NONE/DEBUG=<X>	Whether to produce messages, useful in debugging the NOPAL processor.
X = SAP	Only in Syntax Analysis Phase.
= SEQ	Only in sequencing Phase.
= CODE	Only in Code Generation Phase.
= ALL	All phases.
LINE = <NNNN>	4 digits starting line numbers of the ATLAS program. 1000 is the default.

TABLE 9.1. Options In Running the NOPAL Processor.

The reports are stored in respective files. Table 9.2 shows the files used by the NOPAL processor. The files are all sequential, except for MACRO.cum. The record format of all the files is variable, except for MACRO.LIB, MACRO.DES and MACRO.com, the files have a maximum record length of 500 bytes. To obtain a printed input, it is necessary to print the respective file.

FILE	CONTENTS
SAPLIST	Source specification listing (SOURCE1) with line numbers.
ERRFIL	Error and warning messages.
XREFRPT	Cross reference reports (XREF1 and XREF2).
SOURCE2	Formatted specification (SOURCE2)
SEQRPT	Sequencing reports.
FLWCHT	Flowchart reports.
ATLAS DCL	Interim file for generated ATLAS program declarations.
ATLAS PROC	Interim file for generated ATLAS program procedures.
ATLAS	Complete generated ATLAS program
USRLIB	User supplied ATLAS function library.
SYSLIB	System Supplied ATLAS function library.
RUNLIB	Runtime ATLAS function library.
MACRO.LIB,MACRO.DES,MACRO.DAT	ATLAS action statement templates.

TABLE 9.2. Files Used by the NOPAL Processor.

APPENDIX I —— TEMPLATES USED TO COMPOSE REQUIRE CLAUSES TO DEFINE STIMULI AND MEASUREMENT FUNCTIONS

I.1 ORGANIZATION OF EACH TEMPLATE ENTRY

(Line 1): NAME COMMENT

NAME —— THE SYMBOLIC NAME OF TEMPLATE

COMMENT —— A BRIEF DESCRIPTION OF TEMPLATE

PAGE NO —— IN ATLAS REFERENCE MANUAL(RCA EQUATE ATLAS PROGRAMMING MANUAL FOR AN/USM-410(XE-3)(V) ELECTRONIC EQUIPMENT TEST STATION REPORT NO. 410XE3. PM SEPTEMBER 1976) WHERE THE TEMPLATES ARE DESCRIBED. NOTE THAT THE NOPAL TEMPLATES USE AN UNDERSCORE INSTEAD OF A HYPHEN - USED IN ATLAS.

(LINE 2,3,....): TEMPLATE

I.2 NOTATIONS USED IN TEMPLATES

Notation	Meaning
{ ......}	Select one
[ ]	Optional
|	Optional delimiter
< ......>	Another template name
( )	Variable/constant to be supplied as argument of function. Alternatively the user may provide a constant numeric value.
!( )	Connection point to be supplied on the left hand side of the conjunction that references the function. Alternatively the user may provide a fixed character string.
$	End mark of the template
Other character strings	Terminal symbols to be selected by user

The templates are stored in three different files:

(1) A sequential file MACRO.DES which is used for documentation.

(2) A sequential file MACRO.LIB consists of alphabetically ordered templates.

(3) An ISAM file MACRO.DAT used to verify REQUIRE function.

It is necessary to perform the following steps to add or modify template.

Step 1: Add or modify the MACRO.DES file.
The above organization and conventions must be observed.

Step 2: The template must be inserted in the appropriate alphabetical order into the MACRO.LIB file.

Step 3: Execute the command control file MACRO.COM by keying in @MACRO
This will create a new ISAM file MACRO.DAT

I.3 TEMPLATES, ORDERED BY CATEGORIES

CATEGORY I —— DIMENSION

ADIM ---- THE LEGAL CURRENT UNIT Pg.3-20
{ A | MA | UA } $

WDIM ---- THE LEGAL POWER UNIT Pg.3-20
{ W | MW | UW}$

VDIM ---- THE LEGAL VOLTAGE UNIT Pg.3-20
{ V | KV | MV | UV } $

SDIM ---- THE LEGAL TIME UNIT Pg.3-20
{SEC| MSEC | USEC| NSEC| PSEC}$

OHMDIM ---- THE LEGAL RESISTANCE UNIT Pg.3-20
{ OHM | KOHM | MOHM}$

HZDIM ---- THE LEGAL FREQUENCY UNIT Pg.3-20
{ HZ | KHZ | MHZ | GHZ}$

OHMDEGHZDIM ---- THE LEGAL IMPEDANCE + ANGLE + FREQUENCY UNIT Pg.3-20
{OHMDEGHZ|KOHMDEGHZ|MOHMDEGHZ}$

CATEGORY II ---- CONNECTION POINTS DESCRIPTIONS

HI/GND ---- CONNECTION POINTS DESCRIPTION Pg.C-11
HI ( ) [GND ( )]$

HI/LO ---- CONNECTION POINTS DESCRIPTION Pg.C-11
{ HI ( ) [LO ( )] | LO ( ) }$

HI/LO/GND ---- CONNECTION POINTS DESCRIPTION Pg.C-11
HI ( ) [LO ( ) | GND ( )]$

REF-HI/GND ---- CONNECTION POINTS DESCRIPTION Pg.C-11
REFHI ( ) [REFGND ( )]$

REF-HI/LO ---- CONNECTION POINTS DESCRIPTION Pg.C-11
REFHI ( ) [REFLO ( )]$

REF-HI/LO/GND ---- CONNECTION POINTS DESCRIPTION Pg.C-11
REFHI ( ) [REFLO ( ) | REFGND ( )]$

CONN-REFHI/GND ---- CONNECTION POINTS DESCRIPTION Pg.C-9
{ CNX { { <HI/GND> | BNC ( ) | PROBE } | { <RF-PROBE> | !( ) } }
[<REF-HI/GND> | REFBNC ( )] | { BUFFERED, CNX D( ) | [UNBUFFERED,] CNX
{ { HI ( ) | BNC ( ) | PROBE } | { <RF-PROBE> | !( ) } } }
[REFHI ( ) | REFBNC ( )] } $

CONN-REFHI/LO/GND ---- CONNECTION POINTS DESCRIPTION Pg.C-9
{ CNX { { <HI/LO/GND> | BNC ( ) | PROBE } | { <RF-PROBE> | !( ) } }
[<REF-HI/LO/GND> | REFBNC ( )] | { BUFFERED, CNX ( ) [REFHI ( ) |
REFBNC ( )] | [UNBUFFERED,] CNX { {<HI/LO> | BNC ( ) | PROBE } |
{ <RF-PROBE> | !( ) } } [<REF-HI/LO> | REFBNC ( )] } } $

CONN-TERMHI/GND ---- CONNECTION POINTS DESCRIPTION Pg.C-10
[TESTEQUIPIMP 50 OHM,] { CNX {<HI/GND> | !( )} | CNX { BNC ( ) | PROBE |
!( )} | [UNBUFFERED,] CNX HI ( ) | CNX <RF-PROBE> | BUFFERED, CNX HI ( ) } $

CONN-TERMHI/LO/GND ---- CONNECTION POINTS DESCRIPTION Pg.C-10
[TESTEQUIPIMP 50 OHM,] { CNX { <HI/LO/GND> | !( ) } | CNX { BNC ( ) | PROBE
| !( )} | [UNBUFFERED,] CNX <HI/LO> | CNX <RF-PROBE> | BUFFERED, CNX HI ( )}$

CONNHI/GND ---- CONNECTION POINTS DESCRIPTION Pg.C-8
{ CNX { <HI/GND> | !( ) } | CNX { BNC ( ) | PROBE | !( ) | <RF-PROBE> }
| { [UNBUFFERED,] | BUFFERED, } CNX HI ( ) } $

CONNHI/LO/GND ---- CONNECTION POINTS DESCRIPTION Pg.C-8
{ CNX { <HI/LO/GND> | !( ) } | CNX { BNC ( ) | PROBE | !( ) | <RF-PROBE> }
| { BUFFERED, CNX HI ( ) | [UNBUFFERED], CNX { HI ( ) LO ( ) | <HI/LO> }}} $

RF-PROBE ---- ACTIVE MEASUREMENT PROBES Pg.C-11
{RFPROBE | RFPROBEX1 | RFPROBEX10 | RFPROBEX100}$

CATEGORY III ---- DIGITAL TEST

Cat. III Digital Test :

```
C101 ---- DIGITAL TEST, STIM ONLY                      Pg.C-34
  DO  DIGITALTEST, {STIMONLY | RECIRCULATE }, STIM (),
  MSGLENGTH ( ) WORDS, WORDLENGTH ( ) BITS [, WORDRATE ( ) <HZDIM>
  [, THRUIMP ( ) <OHMDIM>] ], VOLTAGEONE ( ) <VDIM>, VOLTAGEZERO ( )
  <VDIM> [, CNXSTIM ( ) ] $
C102 ---- DIGITAL TEST, STIM-RESP-SAVE                 Pg.C-35
  DO  DIGITALTEST, STIMRESPSAVE, STIM ( ), RESP ( ),
  MSGLENGTH ( ) WORDS, WORDLENGTH ( ) BITS [, WORDRATE ( ) <HZDIM>
  [, RESPDELAY ( ) <SDIM>] [, THRUIMP ( ) <OHMDIM>] ], VOLTAGEONE
  ( ) <VDIM>, VOLTAGEZERO ( ) <VDIM> { [, VOLTAGEONE MIN ( ) <VDIM>]
  [, VOLTAGEZERO MAX ( ) <VDIM>] | [, VOLTAGEONE MAX ( ) <VDIM>]
  [, VOLTAGEZERO MIN ( ) <VDIM>] } [, CNXSTIM ( ), CNXRESP ( ) ] $
C103 ---- DIGITAL TEST, D-LASAR FUNCTION               Pg.C-36
  DO DIGITALTEST, DLASAR, MSGLENGTH ( ) WORDS, WORDLENGTH ( ) BITS, CHANNEL
  ( ) [, ERRORCHANNEL ( )] [, RESP ( )] [, WORDRATE ( ) <HZDIM> [, RESPDELAY
  ( ) <SDIM>] [, THRUIMP ( ) <OHMDIM> ] ], VOLTAGEONE ( ) <VDIM>,
  VOLTAGEZERO ( ) <VDIM> { [, VOLTAGEONE MIN ( ) <VDIM>] [, VOLTAGEZERO MAX
  ( ) <VDIM>]| [, VOLTAGEONE MAX ( ) <VDIM>] [, VOLTAGEZERO MIN ( ) <VDIM>]}
  [, CNXSTIM ( ), CNXRESP ( ) ] $
```

CATEGORY IV ---- ANALOG STIMULUS

```
C61 ---- DC VOLTAGE AND POWER, THE ALLOWED !( ) ARE:
            DC1, DC2A, DC2B, DC3A, DC3B, DC5,AND DC6        Pg.C-15
  APPLY DCSIGNAL !( ), VOLTAGE ( ) <VDIM> [, VOLTAGE MIN ( )
  <VDIM> ] [, VOLTAGE MAX ( ) <VDIM>] [ [, <THRU-IMP>], CNX <HI/LO>] $
C62 ---- DC STANDARD(VOLTAGE SOURCE AND CURRENT SOURCE)   Pg.C-16
  APPLY DCSIGNAL DCSTD, {VOLTAGE ( ) <VDIM> | CURRENT ( ) <ADIM>}
  [,CURRENT MAX ( ) <ADIM> ] [, VOLTAGE MAX ( ) <VDIM>] [ [, <THRU-IMP>], CNX
  <HI/LO> ] $
C63A ---- AC SINEWAVE WAVEFORM GENERATOR                      Pg.6-11
  APPLY ACSIGNAL, SINEWAVE, FREQ ( ) <HZDIM> {, VOLTAGE |
  , VOLTAGEP | , VOLTAGEPP } ( ) <VDIM> [, DCOFFSET ( ) <VDIM>]
  [ , TESTEQUIPIMP {50| 600} OHM [, UUTIMP ( ) <OHMDIM>] | [, THRUIMP
  ( ) <OHMDIM>] , CNX HI ( ) ]$
C63B ---- AC SQUAREWAVE AND TRIANGULARWAVE WAVEFORM GENERATOR   Pg.6-11
  APPLY ACSIGNAL, [ SQUAREWAVE| TRIANGULARWAVE], FREQ ( ) <HZDIM>
  [, VOLTAGEP | , VOLTAGEPP ] ( ) <VDIM> [, DCOFFSET ( ) <VDIM> ]
  [ , TESTEQUIPIMP {50| 600} OHM [, UUTIMP ( ) <OHMDIM>] | [, THRUIMP
  ( ) <OHMDIM>] , CNX HI ( ) ]$
C63C ---- AC SAWTOOTH-WAVE WAVEFORM GENERATOR                  Pg.6-11
  APPLY ACSIGNAL, SAWTOOTHWAVE, FREQ ( ) <HZDIM> [, VOLTAGEP |
  , VOLTAGEPP ] ( ) <VDIM> [,POSSLOPE |, NEGSLOPE] [, DCOFFSET ( ) <VDIM>]
  [ , TESTEQUIPIMP {50| 600} OHM [, UUTIMP ( ) <OHMDIM>] | [, THRUIMP
  ( ) <OHMDIM>] , CNX HI ( ) ]$
C63D ---- AC RANDOM-NOISE WAVEFORM GENERATOR                   Pg.6-12
  APPLY ACSIGNAL, RANDOMNOISE, FREQ ( ) <HZDIM> , VOLTAGETRMS ( ) <VDIM>
  [, DCOFFSET ( ) <VDIM>][ , TESTEQUIPIMP {50| 600} OHM
  [, UUTIMP ( ) <OHMDIM>] | [, THRUIMP ( ) <OHMDIM>] , CNX HI ( ) ]$
C64 ---- AC STANDARD                                            Pg.C-17
  APPLY ACSIGNAL ACSTD, FREQ ( ) <HZDIM>, VOLTAGE ( ) <VDIM>
  [, VOLTAGE MIN ( ) <VDIM> ] [, VOLTAGE MAX ( ) <VDIM> ] [ [ , <THRU-IMP>]
  , CNX <HI/LO> ] $
C65 ---- PULSE TRAINS AND PULSE BURSTS                          Pg.C-18
```

```
  APPLY PULSEDDC { MP | DP, [DELAY ( ) <SDIM>] } , PERIOD ( )
  <SDIM>, WIDTH ( ) <SDIM> [, BURST ( )] [, RISETIME ( ) <SDIM>, FALLTIME
  ( ) <SDIM> ] [ [, <THRU-IMP>] , CNX HI ( ) ] $
C66 --- SYNCHRO SIGNAL GENERATOR                                Pg.C-19
  APPLY SYNCHRO, ANGLE ( ) DEG [ , CNX [ S1 ( ) ] [ S2 ( )] [ S3 ( ) ]
  [ R1 ( ) ] [ R2 ( ) ] ] $
C67 --- AC POWER                                                Pg.C-19
  APPLY ACSIGNAL ACPWR, FREQ ( ) <HZDIM> $
   CATEGORY V --- ANALOG MEASUREMENTS
C81 --- DC VOLTAGE                                              Pg.C-21
  { MEASURE | MONITOR} (VOLTAGE ( ) <VDIM>),
    DCSIGNAL [ [, VOLTAGE MAX ( ) <VDIM> ] [, VOLTAGE MIN ( ) <VDIM>]
  [, SAMPLEWIDTH ( ) <SDIM> | , RANDOMSAMPLE ] [,DELAY ( ) <SDIM> ]
  [,<VERIFY>] , <CONNHI/LO/GND> ] $
C810 --- TIME INTERVAL( SINGLE AND DUAL CHANNEL)                Pg.C-27
  { MEASURE | MONITOR } (TIME ( ) <SDIM>),
  TIMEINTERVAL [ , {SINGLECHANNEL | DUALCHANNEL }, MAXTIME ( ) <SDIM>
  [, DELAY ( ) <SDIM>] [, BUFFERED |, UNBUFFERED ][,<VERIFY>], START,
  <VOLTAGEP>[, <VOLTAGEP>]* [, ACCOUPLE | , DCCOUPLE] [, TESTEQUIPIMP ( )
  <OHMDIM>], THRESHOLD ( ) <VDIM>, { POSSLOPE | NEGSLOPE}, CNX { <HI/GND>
  | BNC ( ) |PROBE | <RF-PROBE> | !( )}, STOP, THRESHOLD ( ) <VDIM>, {POSSLOPE
  |NEGSLOPE}[, <VOLTAGEP> [, <VOLTAGEP> ]* [, ACCOUPLE |, DCCOUPLE]
  [, TESTEQUIPIMP ( ) <OHMDIM>] , CNX { <REF-HI/GND> | !( )} ]] $
C811 --- HARMONIC DISTORTION                                    Pg.C-27
  { MEASURE | MONITOR } (DISTORTION ( ) PC), ACSIGNAL[, VOLTAGE MAX
  ( ) <VDIM> ,{FREQ | FREQ MAX} ( ) <HZDIM>,{ACCOUPLE| DCCOUPLE}
  [, DELAY ( ) <SDIM>] [,<VERIFY>]<CONN-TERMHI/LO/GND>]$
C813 --- AM AND FM MODULATION                                   Pg.C-29
  { MEASURE | MONITOR}{{(MODFREQ ( ) <HZDIM> |
  (MODAMPL ( ) PC | (VOLTAGETRMS ( ) <VDIM> | (DISTORTION ( ) PC}), AMSIGNAL|
  { (MODFREQ ( ) <HZDIM>|(FREQDEV ( ) <HZDIM>|(DISTORTION ( ) PC}), FMSIGNAL}
  [, POWER MAX ( ) DBM, CARFREQ ( ) <HZDIM> [, DELAY ( ) <SDIM>] [,<VERIFY>],
  {<CONNHI/GND> | CNX RFB}]$
C82 --- DC AND AC CURRENT                                       Pg.C-22
  { MEASURE | MONITOR } (CURRENT ( ) <ADIM>), { DCSIGNAL | ACSIGNAL}
  [ [, DELAY ( ) <SDIM>][,<VERIFY>], CNX {ACPWR | DC1 | DC2A | DC2B
  | DC3A | DC3B | VIA ( ) FREETP ( )} ] $
C83A --- AC VOLTAGE(SINEWAVE RMS)                               Pg.C-22
  { MEASURE | MONITOR } (VOLTAGE ( ) <VDIM>), ACSIGNAL [ [ACCOUPLE]
  [, VOLTAGE MAX ( ) <VDIM> ], FREQ ( ) <HZDIM>
  [, DELAY ( ) <SDIM>][,<VERIFY>]  , <CONNHI/LO/GND> ]$
C83B --- AC VOLTAGE(TRUE RMS)                                   Pg.C-23
  { MEASURE | MONITOR } (VOLTAGETRMS ( ) <VDIM>), ACSIGNAL
  [ , VOLTAGETRMS MAX ( ) <VDIM>, FREQ MAX ( ) <HZDIM>
  [ , ACCOUPLE | , DCCOUPLE ] [, DELAY ( ) <SDIM> ] [,<VERIFY>] ,
  <CONN-TERMHI/LO/GND> ] $
C84 --- PEAK AND PEAK-TO-PEAK VOLTAGE FOR AC SIGNALS            Pg.C-23
  { MEASURE | MONITOR } ( {VOLTAGEPMAX | VOLTAGEPMIN | VOLTAGEPP } ( )
  <VDIM>), ACSIGNAL [ {, VOLTAGEPP MAX ( ) <VDIM>, ACCOUPLE
  | {, VOLTAGEP MAX ( ) <VDIM> | , VOLTAGEP MIN ( ) <VDIM> }
  [ , ACCOUPLE | , DCCOUPLE ] } , FREQ MAX ( ) <HZDIM> [, DELAY ( ) <SDIM> ]
  [,<VERIFY>], <CONN-TERMHI/LO/GND> ] $
```

C86 --- FREQUENCY AND PERIOD Pg.C-25

{ MEASURE | MONITOR} {(FREQ ( ) <HZDIM>), ACSIGNAL [, FREQ MAX ( ) <HZDIM> [, SAMPLEWIDTH ( ) <SDIM>] , <C86NEXT>] | (PERIOD ( ) <SDIM>), ACSIGNAL [, PERIOD MIN ( ) <SDIM> [, PERIODAVERAGE ( ) PERIODS], VOLTAGE MAX ( ) <VDIM> [, ACCOUPLE | , DCCOUPLE] [, THRESHOLD ( ) <VDIM>] [, DELAY ( ) <SDIM>] [, MAXTIME ( ) <SDIM>][,<VERIFY>],<CONN-TERMHI/GND> $

C88 --- PULSE MEASUREMENTS FOR PULSED DC/SIGNALS Pg.C-26

{ MEASURE | MONITOR} ({ VOLTAGEPMAX |VOLTAGEPMIN | VOLTAGEPP | OVERSHOOT | UNDERSHOOT } ( ) <VDIM>), PULSEDDC [{,VOLTAGEPP MAX ( ) <VDIM>, ACCOUPLE | {, VOLTAGEP MAX ( ) <VDIM> | , VOLTAGEP MIN ( ) <VDIM> } [, ACCOUPLE | , DCCOUPLE]}, PERIOD ( ) <SDIM>, DUTYCYCLE ( ) PC [, DELAY ( ) <SDIM>][,<VERIFY>] , <CONN-TERMHI/LO/GND>] $

C89 --- RESISTANCE IMPEDANCE Pg.C-26

{ MEASURE | MONITOR } (RES ( ) <OHMDIM>), IMPEDANCE [[, IMP MAX ( ) <OHMDIM> [, FREQ ( ) <HZDIM>] | , RES MAX ( ) <OHMDIM>] [, REFVOLTAGE ( ) <VDIM>] [, DELAY ( ) <SDIM>][,<VERIFY>],CNX {HI ( ) LO ( ) | BNC ( ) | PROBE | UUTACPWR | RESSTD ( )}] $

CATEGORY VI --- RF STIMULUS AND MEASUREMENTS

C91 ---RF SINE WAVE STIMULUS Pg.C-29

APPLY RFSIGNAL {RFA | RFB}, FREQ ( ) <HZDIM>, POWER ( ) DBM [,CNX {COAX ( ) | SANALYZER | SELF TEST}]$

C92A --- RF MODULATED WAVE STIMULUS(AM AND FM)(DEVICE: RFA, RFB) Pg.C-30

APPLY {AMSIGNAL | FMSIGNAL} {RFA | RFB}, CARFREQ ( ) <HZDIM>, CARPOWER ( ) DBM, {EXT, | {SINEWAVE | SQUAREWAVE | TRIANGULARWAVE | SAWTOOTHWAVE, {POSSLOPE | NEGSLOPE} | COMPLEXWAVE,STIM ( )|RANDOMNOISE}, MODFREQ ( ) <HZDIM>,}{MODDEPTH ( ) PC | MODAMPL ( ) PC | MODINDEX ( ) | FREQDEV ( ) <HZDIM>}[, CNX {COAX ( ) | SELF TEST}]$

C92B --- RF MODULATED WAVE STIMULUS(PAM)(DEVICE: RFA, RFB) Pg.C-30

APPLY PAMSIGNAL {RFA | RFB [, DELAY ( ) <SDIM>]}, CARFREQ ( ) <HZDIM>, CARPOWER ( ) DBM, {EXT | PRF ( ) <HZDIM>, PULSEWIDTH ( ) <SDIM>} [, CNX {COAX ( ) | SELF TEST}]$

C93A --- RF ATTENUATION (DEVICE: ATTNA) Pg.C-31

APPLY RFSIGNAL ATTNA, ATTN ( ) DB [, FREQ MAX ( ) <HZDIM>] [, CNX {ATTNOUTPUT ( ) | SELF TEST}]$

C93B --- RF ATTENUATION (DEVICE: ATTNA) Pg.C-31

APPLY RFSIGNAL ATTNB, ATTN ( ) DB [, CNX SELF TEST]$

C94 --- RF POWER MEASUREMENTS Pg.C-31

{ MEASURE | MONITOR} { (POWER ( ) DBM),RFSIGNAL |(POWER ( ) <WDIM>), RFSIGNAL}[[, POWER MAX ( ) {DBM|<WDIM>}] [, FREQ ( ) <HZDIM>][, ( )] [,<VERIFY>], CNX {COAX ( ) | SANALYZEROSC|RFA}]$

C95 --- LF/HF/RF FREQUENCY MEASUREMENTS Pg.C-32

{ MEASURE | MONITOR} (FREQ ( ) <HZDIM>) RFSIGNAL [,[POWER MAX ( ) DBM,][HF | UHF | PBAND| LBAND | SBNAD | CBAND| XBAND| KUBAND][,( )][,<VERIFY>], CNX {COAX ( )| SANALYZEROSC | RFA}]$

C96 --- RF SIGNAL ANALYSIS (DEVICE: S-ANALYZER) Pg.C-32

RFSIGNALS SANALYZER, RFATTN ( ) DB,{ OUTPUTRF, | { OUTPUTIF | {OUTPUTFMDISCRF |OUTPUTFMDISCLF |OUTPUTLOGAMPRF |OUTPUTLOGAMPLF} , BANDWIDTH ( ) <HZDIM>}, FREQ ( ) <HZDIM>,IFATTN ( ) DB,} CNX {COAX ( )| RFA | SANALYZEROSC}$

CATEGORY VII --- MISCELLANEOUS

THRU-IMP --- SOURCE IMPEDANCE LOCATED AT THE PIU TEST POINT
ASSOCIATED VOLTAGE FIELDS SPECIFY VOLTAGE AT THE ATE

END OF IMPEDANCE. Pg.C-11

THRUIMP ( ) <OHMDIM>$

VERIFY --- CHECK WHETHER THE VALUE FALLS WITHIN THE RANGE Pg.C-7

UL ( ) LL ( )$

VOLTAGEP --- FOR THE EASY OF WRITTEN, SO WE SEPARATE IT FROM C810, Pg.C-27

{VOLTAGEP MIN |VOLTAGEP MAX } ( ) <VDIM>$

APPENDIX II - EXAMPLES

The two examples of spectrum analysis and digital counter-testing in this appendix illustrate the use of variaous features in NOPAL and the reports that are produced. Eight reports are shown for each example, as follows.

	Report Description	Report File
1.	Source NOPAL specification as input by user.	SOURCE1
2.	Reformatted NOPAL specification, with indentation of clauses in assertions and conjunctions, blocked tests and declarations.	SOURCE2
3.	Cross-reference of symbolic names used in the specification, showing where referenced, attributes and data types.	--XREF
4.	Cross-references of tests, connection prints, diagnoses, messages, and functions used in the specification.	--XREF
5.	Flowchart of Main Procedure, showing declarations and order of tests.	FLWCHRT
6.	Dependency matrices and flowcharts, showing sequences of steps of each test and of overall specification.	SEQRPT
7.	Errors and warnings.	ERRFIL
8.	ATLAS program	ATLAS

II.1: Frequency Spectrum Analysis Example

The following example illustrates use of the NOPAL processor for analog testing. The example is taken from the RCA EQUATE-ATLAS Programming Manual, July 1981, page 13-66. It is as follows.

THE UUT IS AN AUDIO SIGNAL GENERATOR THAT PRODUCES A SIGNAL COMPOSED OF THREE COMPONENTS OF EQUAL AMPLITUDES, 1 V PEAK, AT THREE FREQUENCIES: 512 HZ, 640 HZ and 786 HZ. VERIFY THAT:

1) THE EXPECTED AUDIO FREQUENCIES ARE WITHIN +- 5%,

2) THE VOLTAGE AT THESE FREQUENCIES IS 1 +- .3 V PEAK,

3) THERE ARE NO FREQUENCY COMPONENTS EXCEEDING -30 DB OUTSIDE THE THREE AUDIO SIGNAL RANGES.

The reports for this example are shown in Figures II.1 through II.8. The NOPAL specification is shown in source form in Figure II.1 and in reformatted form in Figure II.2. For convenience of presentation, the specification is organized in three tests. Test T1 consists of measurement of the audio frequency spectrum, in 256 spectral line, 5 HZ apart. A function called SPECTRUM is used for the measurement and it defines a vector variable FREQ-SPEC with 256 elements. Test T2 consists of evaluation of the first two conditions above for each of the three audio ranges. It is an array test, performed for each of the three audio ranges, with a subscript J denoting the audio range. It verifies the existance of a signal in each of the expected audio signal frequency ranges, with a voltage amplitude of 1 +- .3 V The diagnoses selected are of array type, with three elements, one for each of the three audio ranges. Test T3 verifies the last condition above, whether there is at least one frequency spectral line exceeding -30 DB outside the expected audio frequency ranges. If found, the diagnosis message reports the lowest such frequency component.

The specification in Figure II.1 includes comments which further explain the objectives of statements and variables.

Test T1 uses a SPECTRUM function which defines the spectral values which form a vector FREQSPEC of 256 (subscript I) elements.

In test T2 the spectral values in the three ranges are extracted from FREQ-SPEC to form a 3x10 matrix AUDIOSPEC, with 10 spectral lines for each of the audio frequency ranges (subscripts J and K). (The NOPAL processor simulates this matrix by a vector in the ATLAS program.) AUDIOSPEC is defined as consisting of the 10 spectral line values in FREQSPEC in each of the three audio ranges only. The lower limit spectral lines of each audio range are denoted by the variable LL which has three elements.

In test T3, the spectral lines outside the expected audio signal range form a variable size vector OUTSPEC, of at most 226 elements (excluding the 30 spectral lines of the three audio ranges). OUTSPEC is defined as consisting of the respective spectral line values in FREQSPEC. The size of this vector is variable and may be limited by the existance of the first (lowest) frequency component exceeding -30 DB. The subscript N is used in referring to elements in OUT_SPEC.

The SPECTRUM function is declared and also defined by use of a REQUIRE clause. As shown it has five parameters: po1--the variable denoting the spectrum value, po2--the maximum value of spectrum element voltage, po3--upper frequency, po4--the distance between spectral lines, and po5--the number of spectral lines. There are also two connection point parameters,

CNX_1, and CNX_2.

These parameters are explained by comments in the source specification.

FIG. II.1 : TEST SPECIFICATION SP001 FOR EXAMPLE 1

```
/* NOPAL TEST SPECIFICATION SOURCE FILE: SP001.SPC */

NOPAL PROCESSOR OPTIONS SPECIFIED:


STMT NO.
--------

    1     NOPAL SPECIFICATION SP001 ;

    2     TEST T1;/*MEASURE SPECTRUM USING SPECTRUM FUNCTION*/

    3     MEAS;
    4         CONJ:  <TP1,TP2> = SPECTRUM(FREQ_SPEC(1)DB,2.9V,1 KHZ,
    4                                  5HZ,256) TARGET: FREQ_SPEC(1);

    5         ASSERT:  I = SUBS('FREQ_SPEC',256) TARGET:  I;
                         /*SUBSCRIPT OF OVERALL SPECTRAL LINES*/

    6     FUNC: SPECTRUM,
    6               TYPE = M,
    6               PARAM = (P01,T,DEC),
    6               PARAM = (P02,S,DEC),
    6               PARAM = (P03,S,DEC),
    6               PARAM = (P04,S,DEC),
    6               PARAM = (P05,S,DEC),
    6               REQUIRE = MEASURE (SPECTRUM (P01) DB),
    6                         AC_SIGNAL,VOLTAGE MAX (P02) V,
    6                         FREQ (P03) KHZ,
    6                         SPEC_LINE_FREQ (P04) HZ,
    6                         SPEC_LINES (P05) ,
    6                         AC_COUPLE,
    6                         CNX HI (CNX1)
    6                         LO (CNX2);
    7     UUT_TP: TP1;
    8     UUT_TP: TP2;
```

FIG. II.1 (CONTD.)

```
 9     TEST T2(J); /* CHECK DB IN THREE AUDIO RANGES 512 HZ, 640 HZ,
                  AND 768 HZ, +- 5% TO HAVE AMPLITUDE 1 +-.3 VOLT*/
10     MEAS;
11         ASSERT:  AUDIO_SPEC(J,K) = FREQ_SPEC(LL(J)+K)
11                                  SOURCE: FREQ_SPEC(LL(J)+K),J,K,LL(J)
11                                  TARGET: AUDIO_SPEC(J,K);

12         ASSERT:  J = SUBS('AUDIO_SPEC,DB1,VOLTS,LL,FREQ',3)
12                    TARGET: J;/*SUBSCRIPT OF AUDIO RANGE*/


13         ASSERT:  K = SUBS('AUDIO_SPEC:2',10)
13                    TARGET: K;/*SUBSCRIPT OF SPECTRAL LINE
                      IN AUDIO RANGES*/
                  /*LOWER LIMIT OF AUDIO RANGE SPECTRAL LINES*/

14         ASSERT:  LL(1) = 97 TARGET: LL(1);

15         ASSERT:  LL(2) = 123 TARGET: LL(2);

16         ASSERT:  LL(3) = 148 TARGET: LL(3);

17         ASSERT:  DB1(J) = SUM(AUDIO_SPEC(J,K),K)
17                        TARGET: DB1(J);

18         ASSERT:  VOLTS(J) = 10**(DB1(J)/20)
18                           TARGET:  VOLTS(J);
                    /*TEST PASSING CONDITION*/

19         ASSERT:  VOLTS(J) = 1 +- .3;

20         ASSERT:  FREQ(1) = 512 TARGET FREQ(1);

21         ASSERT:  FREQ(2) = 640 TARGET FREQ(2);

22         ASSERT:  FREQ(3) = 762 TARGET FREQ(3);

23       LOGIC:  | AUDIO_PASS(J), |^AUDIO_FAIL(J);

24       DIAG AUDIO_PASS: PARAM = ('PASSED',FREQ(J),VOLTS(J)),
24                          PRINT = M;
25       DIAG AUDIO_FAIL: PARAM = ('FAILED',FREQ(J),VOLTS(J)),COMP=FAIL(AUDIO_RANGE),
25                          PRINT = M;

26         MESSAGE M: 'TEST(P01)' 'AT FREQUENCY RANGE(P02)###HZ'
26                    'EFFECTIVE VOLTAGE IS (P03)###.##VOLTS';
27         FUNC: SUM;
```

FIG. II.1 (CONTD.)

```
28      TEST T3; /*CHECK IF AT LEAST ONE SPECTRUM LINE OUTSIDE
                   AUDIO RANGE EXCEEDS -30 DB*/

29      MEAS;

30          ASSERT: IF N<98 THEN OUT_SPEC(N)=FREQ_SPEC(N)
30                  ELSE
30                  IF N>97 & N<114 THEN OUT_SPEC(N)=FREQ_SPEC(N+10)
30                  ELSE
30                  IF N>113 & N<128 THEN OUT_SPEC(N)=FREQ_SPEC(N+20)
30                  ELSE
30                  IF N>127 THEN OUT_SPEC(N)=FREQ_SPEC(N+30)
30                                              TARGET OUT_SPEC(N);

31          ASSERT: N = SUBS('OUT_SPEC',*) TARGET: N;
                   /*SUBSCRIPT OF SPECTRAL LINES OUTSIDE AUDIO RANGES*/

32          ASSERT: IF (OUT_SPEC(N)>-30) | (N=226) THEN END_N = FALSE
32                                            TARGET: END_N;

33          ASSERT: OUT_DB = LAST(OUT_SPEC(N),N)
33                                            TARGET: OUT_DB;

34          ASSERT: OUT_VOLTS = 10**(OUT_DB/20)
34                                  TARGET: OUT_VOLTS;

35          ASSERT: OUT_DB>-30 ; /*TEST PASSING CONDITION*/
36        LOGIC:  | OUT_FAIL, |^ OUT_PASS;

37        DIAG OUT_PASS:  PARAM = ('PASSED',OUT_DB,OUT__VOLTS),
37                        PRINT = M;

38        DIAG OUT_FAIL:  PARAM = ('FAILED',OUT_DB,OUT_VOLTS),
38                        COMP=FAIL(OUT_AUDIO_RANGE),
38                        PRINT = M;
39        COMP_FAIL : AUDIO_RANGE, FAILURE FUNC= FAIL;
40        COMP_FAIL : OUT_AUDIO_RANGE, FAILURE FUNC= FAIL;
41        FUNC: LAST;
42        FUNC: SUBS;
43        FUNC: FALSE;
44        FUNC: FAIL, TYPE= F;

45        END SP001 ;
```

II.2: Digital Counter Example

The following example illustrates the use of NOPAL for digital testing.

THE UUT IS A DIGITAL COUNTER OF INPUT PULSES THAT CAN COUNT FROM 0 TO 255. THE COUNTER HAS ONE INPUT LINE OVER WHICH "ONE" OR "ZERO" SIGNALS MAY BE APPLIED AND AN EIGHT BIT PARALLEL OUTPUT OF THE COUNT. THE TEST REQUIREMENTS ARE AS FOLLOWS:

1) UPON APPLICATION OF A "ONE" TO THE INPUT, THE OUTPUT MUST INCREASE BY B'1'.

2) UPON APPLICATION OF A "ZERO" TO THE INPUT, THE OUTPUT MUST NOT CHANGE.

3) IF THE COUNTER HAS A FAILURE, THE DIAGNOSIS MUST INCLUDE ONE COUNTER VALUE WHERE THE COUNTER FAILS.

The reports for this example are shown in Figure II.9 to II.16. The source specification is shown in Figure II.9. For convenience of presentation there are two tests. The first test, T1, applies a stimulus to the input of the counter consisting of a sequqnce of 256 pairs of "ZERO" and "ONE" and measures the counts for each application of a "ZERO" or a "ONE". This test uses a combined stimuli/measurement function names DIGSRS. TP1 is the input connection. TP2 is a vector of eight output connections. STIM and RESP are the input and output vectors (512 elements) respectively. I is the subscript of these vectors.

Test T2 is also a scalar test. It defines a binary variable COND(J) for each one of the applications of an element of STIM that a "ONE" value. The subscript J has a variable range with an upper limit of 256. For each STIM value of one, if the counter ourput-RESP(2*5) is incremented by B'1' then COND(J) is TRUE, otherwise it is FALSE. The function DIG_ADD is used for binary addition of two operands. It has three operands: the two operands that are added (in binary) and the length of the operands in bits. COND(J) is a vector of variable size. The range of J is terminated on the first occurance of COND(J)=FALSE, or if the counter functions well, on J=256; namely, on finding the first malfunction of the counter, or on completing the check of all 256 count positions. This is specified by the variable END.J. The test passes if the last element of COND(J) is TRUE, and fails otherwise.

FIG. II.2 : REFORMATTED REPORT FOR EXAMPLE 1

```
/* REFORMATTED SPECIFICATION REPORT, FILE: SOURCE2 */


    /************************************************/
    /*                                              */
    /*  NOPAL TEST SPECIFICATION FOR SP001          */
    /*                                              */
    /************************************************/

NOPAL SPECIFICATION SP001;

    /************************************************/
    /*                                              */
    /*  TEST MODULES:      3                        */
    /*                                              */
    /************************************************/

  TEST T1;

     /* NULL STIMULUS */

     MEASUREMENT $M_T1(T1);

        CONJUNCTION $M_W0001($M_T1):
            (<TP1, TP2>  =
            SPECTRUM(FREQ_SPEC(1) DB ,2.9 V ,1 KHZ ,5 HZ ,256))
                    TARGET: FREQ_SPEC(1);

        ASSERTION $M_W0002($M_T1):
            I =  SUBS('FREQ_SPEC',256)
                    TARGET: I
                    SOURCE: SUBS;

     /* NULL LOGIC */

  TEST T2(J) ;

     /* NULL STIMULUS */

     MEASUREMENT $M_T2(T2);

        ASSERTION $M_W0001($M_T2):
            AUDIO_SPEC(J,K) =  FREQ_SPEC(LL(J)+K)
                    TARGET: AUDIO_SPEC(J, K)
                    SOURCE: LL(J), K, J, FREQ_SPEC(LL(J)+K);

        ASSERTION $M_W0002($M_T2):
            J =  SUBS('AUDIO_SPEC,DB1,VOLTS,LL,FREQ',3)
                    TARGET: J
                    SOURCE: SUBS;
```

FIG. II.2 (CONTD.)

```
ASSERTION $M_W0003($M_T2):
    K =  SUBS('AUDIO_SPEC:2',10)
            TARGET: K
            SOURCE: SUBS;

ASSERTION $M_W0004($M_T2):
    LL(1) =  97
            TARGET: LL(1);

ASSERTION $M_W0005($M_T2):
    LL(2) =  123
            TARGET: LL(2);

ASSERTION $M_W0006($M_T2):
    LL(3) =  148
            TARGET: LL(3);

ASSERTION $M_W0007($M_T2):
    DB1(J) =  SUM(AUDIO_SPEC(J,K),K)
            TARGET: DB1(J)
            SOURCE: SUM(AUDIO_SPEC(J,K), K), K, J, AUDIO_SPEC(J, K);

ASSERTION $M_W0008($M_T2):
    VOLTS(J) =  10**(DB1(J)/20)
            TARGET: VOLTS(J)
            SOURCE: J, DB1(J);

ASSERTION $M_W0009($M_T2):
    VOLTS(J) =  1 +- .3
            SOURCE: VOLTS(J), J;

ASSERTION $M_W0010($M_T2):
    FREQ(1) =  512
            TARGET: FREQ(1);

ASSERTION $M_W0011($M_T2):
    FREQ(2) =  640
            TARGET: FREQ(2);

ASSERTION $M_W0012($M_T2):
    FREQ(3) =  762
            TARGET: FREQ(3);

LOGIC $LOGIC0010(T2): |AUDIO_PASS(J) , |^AUDIO_FAIL(J) ;

 DIAGNOSIS AUDIO_PASS:
     OPERATOR MESSAGE:
         OTHER PARAMETERS=('PASSED', FREQ(J), VOLTS(J)),
         PRINT=M;

 DIAGNOSIS AUDIO_FAIL:
     OPERATOR MESSAGE:
         AFFECTED COMPONENTS=FAIL(AUDIO_RANGE),
         OTHER PARAMETERS=('FAILED', FREQ(J), VOLTS(J)),
         PRINT=M;
```

FIG. II.2 (CONTD.)

```
TEST T3;

   /* NULL STIMULUS */

   MEASUREMENT $M_T3(T3);

      ASSERTION $M_W0001($M_T3):
          IF N<98 THEN
              OUT_SPEC(N) =  FREQ_SPEC(N)
          ELSE IF N>97&N<114 THEN
                   OUT_SPEC(N) =  FREQ_SPEC(N+10)
                ELSE IF N>113&N<128 THEN
                         OUT_SPEC(N) =  FREQ_SPEC(N+20)
                      ELSE IF N>127 THEN
                               OUT_SPEC(N) =  FREQ_SPEC(N+30)
                    TARGET: OUT_SPEC(N)
                    SOURCE: N, FREQ_SPEC(N+30), FREQ_SPEC(N+20), FREQ_SPEC(
                            N+10), FREQ_SPEC(N);

      ASSERTION $M_W0002($M_T3):
          N =  SUBS('OUT_SPEC',*)
                    TARGET: N
                    SOURCE: SUBS;

      ASSERTION $M_W0003($M_T3):
          IF (OUT_SPEC(N)>-30)|(N=226) THEN
              END_N =  FALSE
                    TARGET: END_N
                    SOURCE: OUT_SPEC(N), N, FALSE;

      ASSERTION $M_W0004($M_T3):
          OUT_DB =  LAST(OUT_SPEC(N),N)
                    TARGET: OUT_DB
                    SOURCE: OUT_SPEC(N), N, LAST(OUT_SPEC(N), N);

      ASSERTION $M_W0005($M_T3):
          OUT_VOLTS =  10**(OUT_DB/20)
                    TARGET: OUT_VOLTS
                    SOURCE: OUT_DB;

      ASSERTION $M_W0006($M_T3):
          OUT_DB >  -30
                    SOURCE: OUT_DB;

   LOGIC $LOGIC0010(T3): |OUT_FAIL, |^OUT_PASS;

     DIAGNOSIS OUT_FAIL:
         OPERATOR MESSAGE:
             AFFECTED COMPONENTS=FAIL(OUT_AUDIO_RA),
             OTHER PARAMETERS=('FAILED', OUT_DB, OUT_VOLTS),
             PRINT=M;

     DIAGNOSIS OUT_PASS:
         OPERATOR MESSAGE:
             OTHER PARAMETERS=('PASSED', OUT_DB, OUT__VOLTS),
             PRINT=M;
```

FIG. II.2 (CONTD.)

```
/**************************************************/
/*                                                */
/*  MESSAGES                                      */
/*                                                */
/**************************************************/

MESSAGE M:
    TEXT='TEST(P01)', 'AT FREQUENCY RANGE(P02)###HZ', 'EFFECTIVE VOLTAGE IS (
P03)###.##VOLTS';

/**************************************************/
/*                                                */
/*  UUT COMPONENTS/FAILURES                       */
/*                                                */
/**************************************************/

COMP_FAIL 00100: AUDIO_RANGE, FAILURE FUNCTION=FAIL;

COMP_FAIL 00200: OUT_AUDIO_RA, FAILURE FUNCTION=FAIL;

/**************************************************/
/*                                                */
/*  UUT CONNECTION POINTS                         */
/*                                                */
/**************************************************/

UUT_POINT       : TP1;

UUT_POINT       : TP2;

/**************************************************/
/*                                                */
/*  ATE FUNCTIONS                                 */
/*                                                */
/**************************************************/

FUNCTION        : SPECTRUM, FUNCTION TYPE=M, #PINS= 2,
        PARAM#01=(P01, T),
        PARAM#02=(P02, S),
        PARAM#03=(P03, S),
        PARAM#04=(P04, S),
        PARAM#05=(P05, S),
        REQUIRE = MEASURE (SPECTRUM 'P01' DB ), AC-SIGNAL, VOLTAGE MAX 'P02'
V, FREQ 'P03' KHZ, SPEC-LINE-FREQ 'P04' HZ, SPEC-LINES 'P05', AC-COUPLE, CNX HI
'SPECTRUM.CNX01' LO 'SPECTRUM.CNX02';

FUNCTION        : FAIL, FUNCTION TYPE=F;

FUNCTION        : SUM, FUNCTION TYPE=E;

FUNCTION        : LAST, FUNCTION TYPE=E;

FUNCTION        : SUBS, FUNCTION TYPE=E;
```

FIG. II.3 : CROSS REFERENCE AND ATTRIBUTES REPORT FOR EXAMPLE 1

NAME	DEF NO.	ATTRIBUTES AND REFERENCES		DATA TYPE
AUDIO_FAIL	25	DIAGNOSIS LABEL 23		-------
AUDIO_PASS	24	DIAGNOSIS LABEL 23		-------
AUDIO_RANGE	39	COMPONENT ID 25	, WITH FAILURE-FUNCTION: FAIL	-------
AUDIO_SPEC	11	VARIABLE ID 17	, 2-DIMENSIONAL ARRAY	INTEGER
DB1	17	VARIABLE ID 18	, 1-DIMENSIONAL ARRAY	INTEGER
END_N	32	VARIABLE ID	, GLOBAL	DECIMAL
FAIL	44	ATE-FUNCTION ID 25 38 39 40	, F	DECIMAL
FALSE	43	ATE-FUNCTION ID 32	, E	DECIMAL
FREQ	20	VARIABLE ID 24 25	, 1-DIMENSIONAL ARRAY	INTEGER
FREQ	21	VARIABLE ID 24 25	, 1-DIMENSIONAL ARRAY	INTEGER
FREQ	22	VARIABLE ID 24 25	, 1-DIMENSIONAL ARRAY	INTEGER
FREQ_SPEC	4	VARIABLE ID 30 11	, 1-DIMENSIONAL ARRAY, GLOBAL	INTEGER
I	5	VARIABLE ID	, GLOBAL	INTEGER
J	12	VARIABLE ID 11 17 18 19 24 25 23	, GLOBAL	INTEGER
K	13	VARIABLE ID 11 17		INTEGER
LAST	41	ATE-FUNCTION ID 33	, E	INTEGER
LL	14	VARIABLE ID 11	, 1-DIMENSIONAL ARRAY	INTEGER
LL	15	VARIABLE ID 11	, 1-DIMENSIONAL ARRAY	INTEGER
LL	16	VARIABLE ID 11	, 1-DIMENSIONAL ARRAY	INTEGER
M	26	MESSAGE LABEL 24 25 37 38		-------
N	31	VARIABLE ID 30 32 33		INTEGER

FIG. II.3 (CONTD.)

```
OUT_AUDIO_RA      40       COMPONENT ID          , WITH FAILURE-FUNCTION: FAIL
                                                                      -------
                            38
OUT_DB            33       VARIABLE ID                                INTEGER
                            34  35  37  38
OUT_FAIL          38       DIAGNOSIS LABEL                            -------
                            36
OUT_PASS          37       DIAGNOSIS LABEL                            -------
                            36
OUT_SPEC          30       VARIABLE ID           , 1-DIMENSIONAL ARRAY
                                                                      INTEGER
                            32  33
OUT_VOLTS         34       VARIABLE ID                                INTEGER
                            38
SP001              1       SPECIFICATION LABEL                        -------
                            37  45
SPECTRUM           6       ATE-FUNCTION ID       , M                  DECIMAL
                             4
SUBS              42       ATE-FUNCTION ID       , E                  INTEGER
                            31  13  12   5
SUM               27       ATE-FUNCTION ID       , E                  INTEGER
                            17
T1                 2       TEST LABEL                                 -------
                             3
T2                 9       TEST LABEL                                 -------
                            10  23
T3                28       TEST LABEL                                 -------
                            29  36
TP1                7       UUT-POINT ID                               -------
                             4
TP2                8       UUT-POINT ID                               -------
                             4
VOLTS             18       VARIABLE ID           , 1-DIMENSIONAL ARRAY
                                                                      INTEGER
                            19  24  25
```

FIG. II.4 : CROSS REFERENCES -- TESTS VS DIAGNOSES VS MESSAGES VS TEST POINTS VS FUNCTIONS -- FOR EXAMPLE 1

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- DIAGNOSES <=> TEST-MODULES

DIAGNOSES	TEST-MODULES
AUDIO_PASS	T2,
AUDIO_FAIL	T2,
OUT_FAIL	T3,
OUT_PASS	T3,

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- MESSAGES <=> DIAGNOSES <=> TESTS

MESSAGE	DIAGNOSES	TEST-MODULES
M	AUDIO_PASS, AUDIO_FAIL, OUT_PASS, OUT_FAIL,	T2, T3,

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- AFFECTED-COMPONENTS <=> DIAGNOSES <=> TESTS

AFFECTED-COMPONENT	DIAGNOSES	TEST-MODULES
00100: FAIL(AUDIO_RANGE)	AUDIO_FAIL,	T2,
00200: FAIL(OUT_AUDIO_RA)	OUT_FAIL,	T3,

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- UUT-CONNECTING-POINTS <=> TEST-MODULES <=> ATE-CONNECTING-POINTS

UUT-CONNECTING-POINT	TEST-MODULES(S/M)	ATE-CONNECTING-POINTS
TP1	T1(M),	
TP2	T1(M),	

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- ATE-FUNCTIONS <=> TEST-MODULES

ATE-FUNCTION,TYPE	TEST-MODULES(S/M)
SPECTRUM, M	T1(M),
SUM, E	
LAST, E	
SUBS, E	
FALSE, E	

FIG. II.5 : FLOWCHART REPORT FOR NOPAL SPECIFICATION SP001

```
    /***** FLOWCHART FOR GLOBAL PROCEDURE SP001 *****/
      BEGIN SP001;
           DECLARE AS GLOBAL VARIABLES:
                FREQ_SPEC(  256), I, J, END_N,
                OUT__VOLTS;
           DECLARE AS EVAL OR CONTROL FUNCTIONS:
                LAST;
           DECLARE AS STIMULUS FUNCTIONS:
                /* NO STIMULUS FUNCTIONS */
           DECLARE AS MEASUREMENT FUNCTIONS:
                SPECTRUM;
           DECLARE AS FAILURE FUNCTIONS:
                FAIL;
           PERFORM TEST T1;
LOOP OF J ITERATES FROM 1 TO     3
           PERFORM TEST T2;
LOOP OF J ENDS ;
           PERFORM TEST T3;
      END SP001;
```

FIG. II.6 : SEQUENCE REPORT FOR EXAMPLE 1

WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION SP001

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	T1	TEST	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
2	T2	TEST	0	0	0	12	13	0	0	0	0	0	1	0	0	0	0	0
3	T3	TEST	0	0	0	0	0	12	13	0	0	0	0	0	0	1	0	0
4	AUDIO_PASS	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	AUDIO_FAIL	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	OUT_FAIL	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	OUT_PASS	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	FREQ_SPEC	VARIABLE	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
9	SUBS	FUNCTION	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
10	I	VARIABLE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	J	VARIABLE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	SUM	FUNCTION	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	FALSE	FUNCTION	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
14	END_N	VARIABLE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	LAST	FUNCTION	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
16	OUT__VOLTS	VARIABLE	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

SEQUENCE OF PROCESSING FOR NOPAL SPECIFICATION SP001

ORDER VECTOR INDEX	ORDER VECTOR	NAME	TYPE
1	16	OUT__VOLTS	GLOBAL VARIABLE
2	15	LAST	EVAL OR CONTROL FUNCTION
3	13	FALSE	EVAL OR CONTROL FUNCTION
4	12	SUM	EVAL OR CONTROL FUNCTION
5	9	SUBS	EVAL OR CONTROL FUNCTION
6	1	T1	TEST
7	8	FREQ_SPEC	GLOBAL VARIABLE
LOOP OF J ITERATES FROM 1 TO			3
8	2	T2	TEST
9	4	AUDIO_PASS	DIAGNOSIS
10	5	AUDIO_FAIL	DIAGNOSIS
LOOP OF J ENDS ;			
11	3	T3	TEST
12	10	I	GLOBAL VARIABLE
13	6	OUT_FAIL	DIAGNOSIS
14	7	OUT_PASS	DIAGNOSIS
15	14	END_N	GLOBAL VARIABLE
16	11	J	GLOBAL VARIABLE

FIG. II.6 (CONTD.)

```
                    INTRA TEST SEQUENCING FOR MODULE T1
                    ANALYSIS OF THE ADJACENCY MATRIX

                                       1 2 3 4 5
                                       - - - - -
   1  $M_W0001      CONJUNCTION        0 0 1 0 0
   2  $M_W0002      ASSERTION          0 0 0 1 0
   3  FREQ_SPEC     VARIABLE           0 0 0 0 0
   4  I             VARIABLE           0 0 1 0 0
   5  SUBS          FUNCTION           0 1 0 0 0

                    SEQUENCE OF PROCESSING FOR TEST MODULE T1
ORDER
VECT  ORDER   RANK     NAME          TYPE                       TEXT
INDEX VECTOR
  1     1      0     $M_W0001      CONJUNCTION      (<TP1, TP2>  = SPECTRUM(FREQ_SPEC(1) DB ,2.9 V ,1 KHZ ,5 HZ ,256))
                                                    TARGET:  FREQ_SPEC( 1);


  2     5      0     SUBS          FUNCTION


  3     2      1     $M_W0002      ASSERTION        I =  SUBS('FREQ_SPEC',256)
                                                    TARGET:  I
                                                    SOURCE:  SUBS;


  4     4      2     I             VARIABLE         GLOBAL / TARGET /


  LOOP-1 STARTS: SUBSCRIPT I ITERATES FROM 1 TO   256
  5     3      3     FREQ_SPEC     SUBSCRIPT_VA     GLOBAL / TARGET /

  LOOP-1 ENDS;


 LOOP SUMMARY TABLE :
   LOOP-1     FIRST NODE IS    5      LAST NODE IS    5      SUBSCRIPT IS I
```

FIG. II.6 (CONTD.)

INTRA TEST SEQUENCING FOR MODULE T2

ANALYSIS OF THE ADJACENCY MATRIX

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	$M_W0001	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			1	0	0	0	0	0	0	0	0	0
2	$M_W0002	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			0	0	0	1	0	0	0	0	0	0
3	$M_W0003	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			0	0	1	0	0	0	0	0	0	0
4	$M_W0004	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			0	1	0	0	0	0	0	0	0	0
5	$M_W0005	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			0	1	0	0	0	0	0	0	0	0
6	$M_W0006	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			0	1	0	0	0	0	0	0	0	0
7	$M_W0007	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			0	0	0	0	0	0	1	0	0	0
8	$M_W0008	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			0	0	0	0	0	0	0	0	1	0
9	$M_W0009	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			0	0	0	0	0	0	0	0	0	0
10	$M_W0010	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			0	0	0	0	0	0	0	0	0	1
11	$M_W0011	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			0	0	0	0	0	0	0	0	0	1
12	$M_W0012	ASSERTION	0	0	0	0	0	0	0	0	0	0	0	01717			0	0	0	0	0	0	0	0	0	1
13	AUDIO_PASS	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	AUDIO_FAIL	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	AUDIO_SPEC	VARIABLE	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	LL	VARIABLE	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	K	VARIABLE	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
18	J	VARIABLE	1	0	0	0	0	0	1	1	1	0	0	0	1	1	1	1	0	0	0	0	1	0	1	1
19	FREQ_SPEC	VARIABLE	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	SUBS	FUNCTION	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	DB1	VARIABLE	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	SUM	FUNCTION	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	VOLTS	VARIABLE	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
24	FREQ	VARIABLE	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0

SEQUENCE OF PROCESSING FOR TEST MODULE T2

ORDER VECT INDEX	ORDER VECTOR	RANK	NAME	TYPE	TEXT
1	4	0	$M_W0004	ASSERTION	LL(1) = 97 TARGET: LL(1);
2	5	0	$M_W0005	ASSERTION	LL(2) = 123 TARGET: LL(2);
3	6	0	$M_W0006	ASSERTION	LL(3) = 148 TARGET: LL(3);
4	10	0	$M_W0010	ASSERTION	FREQ(1) = 512 TARGET: FREQ(1);

FIG. II.6 (CONTD.)

```
 5   11   0   $M_W0011     ASSERTION      FREQ(2) =  640
                                          TARGET:  FREQ( 2);

 6   12   0   $M_W0012     ASSERTION      FREQ(3) =  762
                                          TARGET:  FREQ( 3);

 7   19   0   FREQ_SPEC    SUBSCRIPT_VA   GLOBAL / SOURCE /

 8   20   0   SUBS         FUNCTION

 9    2   1   $M_W0002     ASSERTION      J =  SUBS('AUDIO_SPEC,DB1,VOLTS,LL,FREQ',3)
                                          TARGET:. J
                                          SOURCE:  SUBS;

10    3   1   $M_W0003     ASSERTION      K =  SUBS('AUDIO_SPEC:2',10)
                                          TARGET:  K
                                          SOURCE:  SUBS;

11   17   2   K            VARIABLE       LOCAL

12   18   2   J            VARIABLE       LOCAL

13   22   3   SUM          FUNCTION

14   16   4   LL           SUBSCRIPT_VA   LOCAL

15   24   4   FREQ         SUBSCRIPT_VA   LOCAL

 LOOP-1 STARTS: SUBSCRIPT K ITERATES FROM 1 TO    10
16    1   5   $M_W0001     ASSERTION      AUDIO_SPEC(J,K) =  FREQ_SPEC(LL(J)+K)
                                          TARGET:  AUDIO_SPEC( J,  K)
                                          SOURCE:  LL( J),  K,  J,  FREQ_SPEC( LL(J)+K);

17   15   6   AUDIO_SPEC   SUBSCRIPT_VA   LOCAL

18    7   7   $M_W0007     ASSERTION      DB1(J) =  SUM(AUDIO_SPEC(J,K),K)
                                          TARGET:  DB1( J)
                                          SOURCE:  SUM( AUDIO_SPEC(J,K),  K),  K,  J,  AUDIO_SPEC( J,  K);

 LOOP-1 ENDS;
```

FIG. II.6 (CONTD.)

```
19   21    8    DB1          SUBSCRIPT_VA    LOCAL

20    8    9    $M_W0008     ASSERTION       VOLTS(J) =  10**(DB1(J)/20)
                                             TARGET:  VOLTS( J)
                                             SOURCE:  J,  DB1( J);

21   23   10    VOLTS        SUBSCRIPT_VA    LOCAL

22    9   11    $M_W0009     ASSERTION       VOLTS(J) =  1 +- .3
                                             SOURCE:  VOLTS( J),  [illegible];

23   13   12    AUDIO_PASS   DIAGNOSIS       OTHER PARAMETERS = ('PASSED',FREQ(J),VOLTS(J)) PRINT = M;

24   14   12    AUDIO_FAIL   DIAGNOSIS       AFFECTED COMPONENTS = FAIL(AUDIO_RANGE), OTHER PARAMETERS = (
                                             'FAILED',FREQ(J),VOLTS(J)) PRINT = M;


LOOP SUMMARY TABLE :
   LOOP-1     FIRST NODE IS   16      LAST NODE IS   18      SUBSCRIPT IS K
```

FIG. II.6 (CONTD.)

INTRA TEST SEQUENCING FOR MODULE T3
ANALYSIS OF THE ADJACENCY MATRIX

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	$M_W0001	ASSERTION	0	0	0	0	0	01717			1	0	0	0	0	0	0	0	0	0
2	$M_W0002	ASSERTION	0	0	0	0	0	01717			0	1	0	0	0	0	0	0	0	0
3	$M_W0003	ASSERTION	0	0	0	0	0	01717			0	0	0	0	1	0	0	0	0	0
4	$M_W0004	ASSERTION	0	0	0	0	0	01717			0	0	0	0	0	0	1	0	0	0
5	$M_W0005	ASSERTION	0	0	0	0	0	01717			0	0	0	0	0	0	0	0	1	0
6	$M_W0006	ASSERTION	0	0	0	0	0	01717			0	0	0	0	0	0	0	0	0	0
7	OUT_FAIL	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	OUT_PASS	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	OUT_SPEC	VARIABLE	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	N	VARIABLE	1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
11	FREQ_SPEC	VARIABLE	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	SUBS	FUNCTION	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	END_N	VARIABLE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	FALSE	FUNCTION	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	OUT_DB	VARIABLE	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0
16	LAST	FUNCTION	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	OUT_VOLTS	VARIABLE	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
18	OUT__VOLTS	VARIABLE	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

SEQUENCE OF PROCESSING FOR TEST MODULE T3

ORDER VECT INDEX	ORDER VECTOR	RANK	NAME	TYPE	TEXT
1	11	0	FREQ_SPEC	SUBSCRIPT_VA	GLOBAL / SOURCE /
2	12	0	SUBS	FUNCTION	
3	14	0	FALSE	FUNCTION	
4	18	0	OUT__VOLTS	VARIABLE	GLOBAL / SOURCE /
5	2	1	$M_W0002	ASSERTION	N = SUBS('OUT_SPEC',*) TARGET: N SOURCE: SUBS;
6	10	2	N	VARIABLE	LOCAL
7	16	3	LAST	FUNCTION	

FIG. II.6 (CONTD.)

```
LOOP-1 STARTS: SUBSCRIPT N ITERATES FROM 1 TO *
 8    1    4    $M_W0001     ASSERTION       IF N<98 THEN   OUT_SPEC(N)   =  FREQ_SPEC(N)
                                             ELSE IF N>97&N<114 THEN   OUT_SPEC(N)   =  FREQ_SPEC(N+10)
                                             ELSE IF N>113&N<128 THEN   OUT_SPEC(N)   =  FREQ_SPEC(N+20)
                                             ELSE IF N>127 THEN   OUT_SPEC(N)   =  FREQ_SPEC(N+30)
                                             TARGET:  OUT_SPEC( N)
                                             SOURCE:  N,  FREQ_SPEC( N+30),  FREQ_SPEC( N+20),
                                             FREQ_SPEC( N+10), FREQ_SPEC( N);

 9    9    5    OUT_SPEC     SUBSCRIPT_VA    LOCAL

10    3    6    $M_W0003     ASSERTION       IF (OUT_SPEC(N)>-30)|(N=226) THEN   END_N   =  FALSE
                                             TARGET:  END_N
                                             SOURCE:  OUT_SPEC( N),  N,  FALSE;

11    4    6    $M_W0004     ASSERTION       OUT_DB =  LAST(OUT_SPEC(N),N)
                                             TARGET:  OUT_DB
                                             SOURCE:  OUT_SPEC( N),  N,  LAST( OUT_SPEC(N),  N);

LOOP-1 ENDS;

12   13    7    END_N        VARIABLE        GLOBAL / TARGET /

13   15    7    OUT_DB       VARIABLE        LOCAL

14    5    8    $M_W0005     ASSERTION       OUT_VOLTS =  10**(OUT_DB/20)
                                             TARGET:  OUT_VOLTS
                                             SOURCE:  OUT_DB;

15    6    8    $M_W0006     ASSERTION       OUT_DB >  -30
                                             SOURCE:  OUT_DB;

16    8    9    OUT_PASS     DIAGNOSIS       OTHER PARAMETERS = ('PASSED',OUT_DB,OUT__VOLTS) PRINT = M;

17   17    9    OUT_VOLTS    VARIABLE        LOCAL

18    7   10    OUT_FAIL     DIAGNOSIS       AFFECTED COMPONENTS = FAIL(OUT_AUDIO_RA), OTHER PARAMETERS = (
                                             'FAILED',OUT_DB,OUT_VOLTS) PRINT = M;


LOOP SUMMARY TABLE :
  LOOP-1     FIRST NODE IS    8      LAST NODE IS    11      SUBSCRIPT IS N
```

FIG. II.7 : ERROR AND WARNING REPORT FOR EXAMPLE 1

```
 ERROR/WARNING MESSAGES GENERATED DURING NOPAL SYNTAX ANALYSIS:
 **WARNING** LEX(LABEL)1: NAME/INTEGER WAS TOO LONG.  TRUNCATED. IN STATEMENT   38, NEAR TEXT 'OUT_AUDIO_RA'
 **WARNING** LEX(LABEL)1: NAME/INTEGER WAS TOO LONG.  TRUNCATED. IN STATEMENT   40, NEAR TEXT 'OUT_AUDIO_RA'
  *STATISTICS* NO. OF SAP ERRORS =    0  ,   NO. OF WARNINGS =    2  , NO. OF STATEMENTS=   45


ERROR/WARNING MESSAGES GENERATED DURING CROSS-REFERENCE:
**WARNING** XRF3:  FREQ IS USED AS A LOCAL VARIABLE IN TEST T2
            AND ALSO IN DIAGNOSES AUDIO_PASS ,AUDIO_FAIL
**WARNING** XRF3:  OUT_DB IS USED AS A LOCAL VARIABLE IN TEST T3
            AND ALSO IN DIAGNOSES OUT_PASS ,OUT_FAIL
**WARNING** XRF3:  VOLTS IS USED AS A LOCAL VARIABLE IN TEST T2
            AND ALSO IN DIAGNOSES AUDIO_PASS ,AUDIO_FAIL

*STATISTICS*  NO. OF XREF1 ERRORS =    0     NO. OF WARNINGS =    3


 ERROR/WARNING MESSAGES GENERATED DURING SEQUENCING AND CODE GENERATION:

**WARNING** ESQ6: (POSSIBLE INCOMPLETENESS): GLOBAL VAR I DEFINED AS TARGET IN TEST T1; BUT NEVER USED.

**WARNING** ESQ6: (POSSIBLE INCOMPLETENESS): GLOBAL VAR J DEFINED AS TARGET IN TEST T2; BUT NEVER USED.

**WARNING** ESQ9: (POSSIBLE INCOMPLETENESS): TEST MODULE T1 DOES NOT HAVE ANY DIAGNOSIS.


WARNING MESSAGES GENERATED DURING CODE GENERATION
**WARNING** INI1: UUT POINT 'TP2' NOT CONNECTED TO ANY ATE PIN
**WARNING** INI1: UUT POINT 'TP1' NOT CONNECTED TO ANY ATE PIN
**WARNING** FLB3: SYSLIB DOES NOT EXIST  SHOULD BE INCLUDED AT ATLAS COMPILE TIME.
**WARNING** FLB3: RUNLIB DOES NOT EXIST  SHOULD BE INCLUDED AT ATLAS COMPILE TIME.
**WARNING** ISQ8: (POSSIBLE AMBIGUITY): VARIABLE LL OF TEST T2 , IS  DEFINED MORE THAN ONCE IN $M_W0004 ,
                  $M_W0005 ,$M_W0006 .  THEY MUST BE EXCLUSIVE .
**WARNING** ISQ8: (POSSIBLE AMBIGUITY): VARIABLE FREQ OF TEST T2 , IS  DEFINED MORE THAN ONCE IN $M_W0010 ,
                  $M_W0011 ,$M_W0012 .  THEY MUST BE EXCLUSIVE .
**WARNING** SUS8: (INCONSISTENT): IN STATEMENT NUMBER      30
           THE BOUND OF DIMENSION-1 OF VARIABLE FREQ_SPEC  WAS DECLARED TO BE 256 BUT 16001  IS USED HERE.

  *STATISTICS* NO. CODE GENERATION ERRORS =    0, NO. OF WARNINGS=    4

  *STATISTICS* NO. OF SEQUENCING ERRORS=    0, NO. OF WARNINGS=    5
```

FIG. II.8 : ATLAS PROGRAM FOR EXAMPLE 1

```
        BEGIN EQUATE PROGRAM 'SP001' $
C          NAMES                 INDEX
C           *** TEST MODULES ***
C           'T1'                 --  1
C           'T2'                 --  2
C           'T3'                 --  3
C
C           *** DIAGNOSES ***
C           'AUDIO-PASS'         --  1
C           'AUDIO-FAIL'         --  2
C           'OUT-FAIL'           --  3
C           'OUT-PASS'           --  4
C
C           *** AFFECTED COMPONENTS ***
C            FAIL(OUT-AUDIO-RA)          --  1
C            FAIL(AUDIO-RANGE)           --  2
C     $
C    DECLARATIONS OF SYSTEM VARIABLES  $
        DECLARE DIGITAL, LIST,'SYS.DIAG-FLAG'(4) $
        DECLARE DECIMAL, 'SYS.S-TIME' $
        DECLARE DECIMAL, 'SYS.D-TIME' $
        DECLARE DECIMAL, 'SYS.DUMMY' $
        DECLARE DECIMAL, 'SYS.NAME' $
        DECLARE DECIMAL, LIST,'SYS.#TESTS IN CONJ'(4) $
        DECLARE DIGITAL, LIST, 'SYS.TEST-FLAG'(3) $
        DECLARE DIGITAL, 'SYS.FLAG' $
        DECLARE DIGITAL, 'SYS.ASRT-FLAG' $
        DECLARE DECIMAL, 'SYS.TIM', 'SYS.TIME' $
        DECLARE DECIMAL, LIST, 'SYS.CLOCK'(6)  $
        DECLARE DECIMAL, 'SYS.I' $
        DECLARE DIGITAL, 'SYS.Y/N' $
        DECLARE DIGITAL, 'SYS.STATE' $
        DECLARE DIGITAL, 'SYS.SELECT' $
        DECLARE DECIMAL, LIST, 'AFF-COMP.NAME'(1) $
        DECLARE DIGITAL, LIST, 'AFF-COMP.SELECT'(1) $
        DECLARE DECIMAL, LIST, 'AFF-COMP.WHERE'(2) $
        DECLARE DIGITAL, LIST, 'AFF-COMP.STATE'(2) $
        DECLARE DECIMAL, 'AFF-COMP.TEST' $
        DECLARE DECIMAL, 'AFF-COMP.COUNT' $
        DECLARE DIGITAL, 'AFF-COMP.CHANGE' $
C    *** CONSTANTS ***  $
        DEFINE 'SYS.SELECTED' , B'10' $
        DEFINE 'SYS.NOT SELECTED' , B'01' $
        DEFINE 'SYS.NOT TESTED' , B'00' $
        DEFINE 'SYS.TESTED' , B'10' $
        DEFINE 'SYS.SKIPPED' , B'01' $
        DEFINE 'SYS.#DIAGS' , 4 $
        DEFINE 'SYS.#TESTS' , 3 $
        DEFINE 'SYS.TRUE' , B'1' $
        DEFINE 'SYS.FALSE' , B'0' $
        DEFINE 'SYS.DONT KNOW', B'00' $
        DEFINE 'SYS.IS', B'001' $
        DEFINE 'SYS.IS NOT', B'010' $
        DEFINE 'SYS.MAY BE', B'011' $
        DEFINE 'SYS.MAY BE NOT', B'100' $
        DEFINE 'SYS.#COMPONENTS',2 $
```

FIG. II.8 (CONTD.)

```
C     UUT POINT DEFINITIONS      $
C     FOLLOWING DISK FILE SHOULD CONTAIN THE
C     MISSING EQUATE/UUT PIU & DIU ASSIGNMENTS. $
          INCLUDE "CNXSPO" $
C     MACRO DEFINITIONS $
          DEFINE 'PRT.TIME', 'SYS.CLOCK'(1), " ##:",
                 'SYS.CLOCK'(2), "##:", 'SYS.CLOCK'(3), "##" $
C     DECLARATIONS FOR USER DEFINED GLOBAL VARIABLES    $
          DECLARE INTEGER, LIST, 'FREQ-SPEC' (256) $
          DECLARE INTEGER,  'I'  $
          DECLARE INTEGER,  'J'  $
          DECLARE DIGITAL,  'END-N'  $
          DECLARE DECIMAL,  'OUT--VOLTS'  $
C     SYSTEM UTILITY ROUTINES  $
          DEFINE PROCEDURE, 'GET.TIME' $
            READ(TIME 'SYS.CLOCK'(1) ALL), SYS-CLOCK $
            'SYS.TIME' = 3600*'SYS.CLOCK'(1) + 60*'SYS.CLOCK'(2) +
                   'SYS.CLOCK'(3) $
          END 'GET.TIME' $
          DECLARE DECIMAL, 'SPECTRUM.CNX01' $
          DECLARE DECIMAL, 'SPECTRUM.CNX02' $
          DECLARE DECIMAL, 'SYS.DEC.01' $
          DECLARE INTEGER, LIST,'AUDIO-SPEC.2' (3,10) $
          DECLARE INTEGER, LIST,'LL.2' (3) $
          DECLARE INTEGER, 'K.2'  $
          DECLARE INTEGER, 'J.2'  $
          DECLARE INTEGER, LIST,'DB1.2' (3) $
          DECLARE INTEGER, LIST,'VOLTS.2' (3) $
          DECLARE INTEGER, LIST,'FREQ.2' (3) $
          DECLARE DIGITAL, 'END-K.2' $
          DECLARE DECIMAL, 'SYS.DEC.02' $
          DECLARE DECIMAL, 'D001.P02' $
          DECLARE DECIMAL, 'D001.P03' $
          DECLARE DECIMAL, 'D002.P02' $
          DECLARE DECIMAL, 'D002.P03' $
          DECLARE INTEGER, LIST,'OUT-SPEC.3' (2) $
          DECLARE INTEGER, 'N.3'  $
          DECLARE INTEGER, 'OUT-DB.3'  $
          DECLARE INTEGER, 'OUT-VOLTS.3'  $
          DECLARE DIGITAL, 'SYS.DIG.01' $
          DECLARE DIGITAL, 'SYS.DIG.02' $
          DECLARE DIGITAL, 'SYS.DIG.03' $
          DECLARE DIGITAL, 'SYS.DIG.04' $
          DECLARE DIGITAL, 'SYS.DIG.05' $
          DECLARE DIGITAL, 'SYS.DIG.06' $
          DECLARE DIGITAL, 'SYS.DIG.07' $
          DECLARE DIGITAL, 'SYS.DIG.08' $
          DECLARE DIGITAL, 'SYS.DIG.09' $
          DECLARE DIGITAL, 'SYS.DIG.10' $
          DECLARE DIGITAL, 'SYS.DIG.11' $
          DECLARE DIGITAL, 'END-N.3' $
```

FIG. II.8 (CONTD.)

```
        DECLARE DECIMAL, 'D004.P02' $
        DECLARE DECIMAL, 'D004.P03' $
        DECLARE DECIMAL, 'D003.P02' $
        DECLARE DECIMAL, 'D003.P03' $
        INCLUDE "SYSLIB" $
        DEFINE PROCEDURE, 'AFF-COMP.PRINT'$
 300    COMPARE 'SYS.I', LT 2 $ GOTO STEP 305 IF NOGO $
        RECORD 'AFF-COMP.COUNT',"###: FAIL(OUT-AUDIO-RA)"$ GOTO STEP 310 $
 305    RECORD 'AFF-COMP.COUNT',"###: FAIL(AUDIO-RANGE)"$
 310    END 'AFF-COMP.PRINT'$
        INCLUDE "RUNLIB" $
C***************************************************************************$
C***************************************************************************$
C    DIAGNOSES PROCS $
C***************************************************************************$
C***************************************************************************$
 1000   DEFINE PROCEDURE, 'AUDIO-PASS' $
        RECORD "TEST(P01)|LAT FREQUENCY RANGE(P02)###HZ|LEFFECTIVE ",
           "VOLTAGE IS (P03)###.##VOLTS" $
        'SYS.DIAG-FLAG'(1) = 'SYS.SELECTED' $
        END 'AUDIO-PASS' $
C***************************************************************************$
 1100   DEFINE PROCEDURE, 'AUDIO-FAIL' $
        'AFF-COMP.COUNT' = 1 $
        'AFF-COMP.NAME'(1) = 2 $
        'AFF-COMP.SELECT'(1) = 'SYS.MAY BE' $
        PERFORM 'AFF-COMP.UPDATE' $
        RECORD "TEST(P01)|LAT FREQUENCY RANGE(P02)###HZ|LEFFECTIVE ",
           "VOLTAGE IS (P03)###.##VOLTS" $
        'SYS.DIAG-FLAG'(2) = 'SYS.SELECTED' $
        END 'AUDIO-FAIL' $
C***************************************************************************$
 1200   DEFINE PROCEDURE, 'OUT-FAIL' $
        'AFF-COMP.COUNT' = 1 $
        'AFF-COMP.NAME'(1) = 1 $
        'AFF-COMP.SELECT'(1) = 'SYS.MAY BE' $
        PERFORM 'AFF-COMP.UPDATE' $
        RECORD "TEST(P01)|LAT FREQUENCY RANGE(P02)###HZ|LEFFECTIVE ",
           "VOLTAGE IS (P03)###.##VOLTS" $
        'SYS.DIAG-FLAG'(3) = 'SYS.SELECTED' $
        END 'OUT-FAIL' $
C***************************************************************************$
 1300   DEFINE PROCEDURE, 'OUT-PASS' $
        RECORD "TEST(P01)|LAT FREQUENCY RANGE(P02)###HZ|LEFFECTIVE ",
           "VOLTAGE IS (P03)###.##VOLTS" $
        'SYS.DIAG-FLAG'(4) = 'SYS.SELECTED' $
        END 'OUT-PASS' $
C***************************************************************************$
C***************************************************************************$
C    TEST PROCS $
C***************************************************************************$
C***************************************************************************$
```

FIG. II.8 (CONTD.)

```
1400    DEFINE PROCEDURE, 'T1' $
        'SYS.FLAG' = 'SYS.TRUE' $
        'SYS.ASRT-FLAG' = 'SYS.TRUE' $
        'AFF-COMP.TEST' = 1 $
        DEFINE 'SPECTRUM.CNX01', 'TP1' $
        DEFINE 'SPECTRUM.CNX02', 'TP2' $
         MEASURE (SPECTRUM 2.9 DB ), AC-SIGNAL, VOLTAGE MAX 1 V, FREQ 5
           KHZ, SPEC-LINE-FREQ 256 HZ, SPEC-LINES 'FREQ-SPEC'(1)
           AC-COUPLE, CNX HI 'SPECTRUM.CNX01' LO 'SPECTRUM.CNX02' $
1405    'SYS.TEST-FLAG'(1) = 'SYS.TESTED' $
        END 'T1' $
C****************************************************************************$
1500    DEFINE PROCEDURE, 'T2' $
        'SYS.FLAG' = 'SYS.TRUE' $
        'SYS.ASRT-FLAG' = 'SYS.TRUE' $
        'AFF-COMP.TEST' = 2 $
        'LL.2'(1) = 97 $
        'LL.2'(2) = 123 $
        'LL.2'(3) = 148 $
        'FREQ.2'(1) = 512 $
        'FREQ.2'(2) = 640 $
        'FREQ.2'(3) = 762 $
        'END-K.2' = 'SYS.TRUE' $
        FOR 'K.2' = 1 THRU 10 BY 1 THEN $
            'AUDIO-SPEC.2'('J.2','K.2') = 'FREQ-SPEC'(('LL.2'('J.2')+
                'K.2')) $
            'SUM.PRM01' = 'AUDIO-SPEC.2'('J.2','K.2') $
            'SUM.PRM02' = 'K.2' $
1505        PERFORM 'SUM' $
            'SYS.DEC.01' = 'SUM.RES' $
            'DB1.2'('J.2') = 'SYS.DEC.01' $
            COMPARE 'END-K.2', EQ  'SYS.TRUE' $
            GOTO STEP  1510     IF NOGO $
        END FOR $
1510    'VOLTS.2'('J.2') = (10**('DB1.2'('J.2')/ 20)) $
        'SYS.DEC.02' = .3 $
        COMPARE 'VOLTS.2'('J.2'), UL 1 + 'SYS.DEC.02' LL 1 - 'SYS.DEC.02' $
        GOTO STEP  1515     IF GO $
        'SYS.FLAG' = 'SYS.FALSE' $
        'SYS.ASRT-FLAG' = 'SYS.FALSE' $
1515    COMPARE 'SYS.ASRT-FLAG', EQ 'SYS.TRUE' $
        GOTO STEP  1520     IF NOGO $
        'D001.P02' = 'FREQ.2'('J.2') $
        'D001.P03' = 'VOLTS.2'('J.2') $
        PERFORM 'AUDIO-PASS'  $
1520    COMPARE 'SYS.ASRT-FLAG', EQ 'SYS.TRUE' $
        GOTO STEP  1525     IF GO $
        'D002.P02' = 'FREQ.2'('J.2') $
        'D002.P03' = 'VOLTS.2'('J.2') $
        PERFORM 'AUDIO-FAIL'  $
1525    'SYS.TEST-FLAG'(2) = 'SYS.TESTED' $
        END 'T2' $
C****************************************************************************$
1600    DEFINE PROCEDURE, 'T3' $
        'SYS.FLAG' = 'SYS.TRUE' $
        'SYS.ASRT-FLAG' = 'SYS.TRUE' $
        'AFF-COMP.TEST' = 3 $
```

FIG. II.8 (CONTD.)

```
        FOR 'N.3' = 1 THRU 16001 BY 1 THEN $
            COMPARE 'N.3', LT 98 $
            GOTO STEP 1605     IF NOGO $
            'SYS.DIG.01' = 'SYS.TRUE' $
            GOTO STEP 1610     $
1605        'SYS.DIG.01' = 'SYS.FALSE' $
1610        COMPARE 'SYS.DIG.01', EQ 'SYS.TRUE' $
            GOTO STEP 1615     IF NOGO $
            'OUT-SPEC.3'(2) = 'FREQ-SPEC'(2) $
            GOTO STEP 1700     $
1615        COMPARE 'N.3', GT 97 $
            GOTO STEP 1620     IF NOGO $
            'SYS.DIG.02' = 'SYS.TRUE' $
            GOTO STEP 1625     $
1620        'SYS.DIG.02' = 'SYS.FALSE' $
1625        COMPARE 'N.3', LT 114 $
            GOTO STEP 1630     IF NOGO $
            'SYS.DIG.03' = 'SYS.TRUE' $
            GOTO STEP 1635     $
1630        'SYS.DIG.03' = 'SYS.FALSE' $
1635        COMPARE B'1', EQ 'SYS.DIG.02' AND 'SYS.DIG.03' $
            GOTO STEP 1640     IF NOGO $
            'SYS.DIG.04' = 'SYS.TRUE' $
            GOTO STEP 1645     $
1640        'SYS.DIG.04' = 'SYS.FALSE' $
1645        COMPARE 'SYS.DIG.04', EQ 'SYS.TRUE' $
            GOTO STEP 1650     IF NOGO $
            'OUT-SPEC.3'(2) = 'FREQ-SPEC'(('N.3'+ 10)) $
            GOTO STEP 1700     $
1650        COMPARE 'N.3', GT 113 $
            GOTO STEP 1655     IF NOGO $
            'SYS.DIG.05' = 'SYS.TRUE' $
            GOTO STEP 1660     $
1655        'SYS.DIG.05' = 'SYS.FALSE' $
1660        COMPARE 'N.3', LT 128 $
            GOTO STEP 1665     IF NOGO $
            'SYS.DIG.06' = 'SYS.TRUE' $
            GOTO STEP 1670     $
1665        'SYS.DIG.06' = 'SYS.FALSE' $
1670        COMPARE B'1', EQ 'SYS.DIG.05' AND 'SYS.DIG.06' $
            GOTO STEP 1675     IF NOGO $
            'SYS.DIG.07' = 'SYS.TRUE' $
            GOTO STEP 1680     $
1675        'SYS.DIG.07' = 'SYS.FALSE' $
1680        COMPARE 'SYS.DIG.07', EQ 'SYS.TRUE' $
            GOTO STEP 1685     IF NOGO $
            'OUT-SPEC.3'(2) = 'FREQ-SPEC'(('N.3'+ 20)) $
            GOTO STEP 1700     $
1685        COMPARE 'N.3', GT 127 $
            GOTO STEP 1690     IF NOGO $
            'SYS.DIG.08' = 'SYS.TRUE' $
            GOTO STEP 1695     $
```

FIG. II.8 (CONTD.)

```
1690        'SYS.DIG.08' = 'SYS.FALSE' $
1695        COMPARE 'SYS.DIG.08', EQ 'SYS.TRUE' $
            GOTO STEP  1700      IF NOGO $
            'OUT-SPEC.3'(2) = 'FREQ-SPEC'(('N.3'+ 30)) $
            GOTO STEP  1700      $
1700        COMPARE 'OUT-SPEC.3'(2), GT ( 0.000000E+00- 30) $
            GOTO STEP  1705      IF NOGO $
            'SYS.DIG.09' = 'SYS.TRUE' $
            GOTO STEP  1710      $
1705        'SYS.DIG.09' = 'SYS.FALSE' $
1710        COMPARE 'N.3', EQ 226 $
            GOTO STEP  1715      IF NOGO $
            'SYS.DIG.10' = 'SYS.TRUE' $
            GOTO STEP  1720      $
1715        'SYS.DIG.10' = 'SYS.FALSE' $
1720        COMPARE B'1', EQ 'SYS.DIG.09' OR 'SYS.DIG.10' $
            GOTO STEP  1725      IF NOGO $
            'SYS.DIG.11' = 'SYS.TRUE' $
            GOTO STEP  1730      $
1725        'SYS.DIG.11' = 'SYS.FALSE' $
1730        COMPARE 'SYS.DIG.11', EQ 'SYS.TRUE' $
            GOTO STEP  1735      IF NOGO $
            'END-N' = 'SYS.FALSE' $
            GOTO STEP  1735      $
1735        'LAST.PRM01' = 'OUT-SPEC.3'(2) $
            'LAST.PRM02' = 'N.3' $
1740        PERFORM 'LAST' $
            'SYS.DEC.01' = 'LAST.RES' $
            'OUT-DB.3' = 'SYS.DEC.01' $
            'OUT-SPEC.3'(1) = 'OUT-SPEC.3'(2) $
            'FREQ-SPEC.3'(1) = 'FREQ-SPEC.3'(2) $
            COMPARE 'END-N.3', EQ  'SYS.TRUE' $
            GOTO STEP  1745      IF NOGO $
        END FOR $
1745    'OUT-VOLTS.3' = (10**('OUT-DB.3'/ 20)) $
        COMPARE 'OUT-DB.3',  GT ( 0.000000E+00- 30) $
        GOTO STEP  1750      IF GO $
        'SYS.FLAG' = 'SYS.FALSE' $
        'SYS.ASRT-FLAG' = 'SYS.FALSE' $
1750    COMPARE 'SYS.ASRT-FLAG', EQ 'SYS.TRUE' $
        GOTO STEP  1755      IF GO $
        'D004.P02' = 'OUT-DB.3' $
        'D004.P03' = 'OUT--VOLTS' $
        PERFORM 'OUT-PASS'  $
1755    COMPARE 'SYS.ASRT-FLAG', EQ 'SYS.TRUE' $
        GOTO STEP  1760      IF NOGO $
        'D003.P02' = 'OUT-DB.3' $
        'D003.P03' = 'OUT-VOLTS.3' $
        PERFORM 'OUT-FAIL'  $
1760    'SYS.TEST-FLAG'(3) = 'SYS.TESTED' $
        END 'T3' $
C************************************************************************$
C************************************************************************$
```

FIG. II.8 (CONTD.)

```
E  1800    PERFORM 'GET.TIME'  $
           RECORD "!X TESTING UUT: SP001" $
           RECORD  'SYS.CLOCK'(4), "DATE ##/",'SYS.CLOCK'(5), "##/" ,
                    'SYS.CLOCK'(6), "## TIME", 'PRT.TIME' $
           'SYS.TIM' = 'SYS.TIME' $
           FOR 'SYS.I' = 1 THRU 'SYS.#DIAGS' THEN $
              'SYS.DIAG-FLAG'('SYS.I')= 'SYS.NOT SELECTED' $
              'SYS.#TESTS IN CONJ'('SYS.I') = 0 $
           END FOR  $
           FOR 'SYS.I' = 1 THRU 'SYS.#COMPONENTS' THEN $
              'AFF-COMP.STATE'('SYS.I') = 'SYS.DONT KNOW' $
              'AFF-COMP.WHERE'('SYS.I') = 0 $
           END FOR $
           FOR 'SYS.I' = 1 THRU 'SYS.#TESTS' THEN $
               'SYS.TEST-FLAG'('SYS.I') = 'SYS.NOT TESTED' $
           END FOR $
C    BEGINNING OF TESTING       $
C    $
C    CONTROL PRECEDING THE CALL ON THE TEST MODULE 'T1' $
 1900      'SYS.FLAG' = 'SYS.TRUE' $
           PERFORM 'T1' $
C $
           FOR 'J' = 1 THRU '3' THEN $
C    CONTROL PRECEDING THE CALL ON THE TEST MODULE 'T2' $
 2000      'SYS.FLAG' = 'SYS.TRUE' $
           PERFORM 'T2' $
C $
           END FOR $
C    CONTROL PRECEDING THE CALL ON THE TEST MODULE 'T3' $
 2100      'SYS.FLAG' = 'SYS.TRUE' $
           PERFORM 'T3' $
C $
 2200      PERFORM 'GET.TIME' $
           RECORD "FINISHED TESTING AT", 'PRT.TIME' $
           'SYS.TIM' = 'SYS.TIME' - 'SYS.TIM' $
           'SYS.CLOCK'(1) = INT ('SYS.TIM'/3600) $
           'SYS.TIM' = 'SYS.TIM' - 3600*'SYS.CLOCK'(1) $
           'SYS.CLOCK'(2) = INT('SYS.TIM'/60) $
           'SYS.CLOCK'(3) = 'SYS.TIM' - 60*'SYS.CLOCK'(2) $
           RECORD "DURATION " , 'PRT.TIME' $
           REMOVE ALL  $
           RECORD "DO YOU WISH TO SEE THE FINAL FAULT ISOLATION STATE ? ",
              "(Y/N) " $
           WAIT-FOR MANUAL-DATA-GO-NOGO $
           GOTO STEP 2205 IF NOGO $
           PERFORM 'PRINT.DONT KNOW' $
           PERFORM 'PRINT.IS' $
           PERFORM 'PRINT.IS NOT' $
           PERFORM 'PRINT.MAY BE' $
           PERFORM 'PRINT.MAY BE NOT' $
 2205      RECORD "DO YOU WISH TO RERUN THIS PROGRAM? (Y/N)" $
           WAIT-FOR MANUAL-DATA-GO-NOGO $
           GOTO STEP 2210 IF NOGO $
```

FIG. II.9 : TEST SPECIFICATION DIGITAL_TEST FOR EXAMPLE 2

```
/* NOPAL TEST SPECIFICATION SOURCE FILE: DIGIT.SPC */

NOPAL PROCESSOR OPTIONS SPECIFIED:


STMT NO.
--------

   1    NOPAL SPECIFICATION DIGITAL_TEST;

   2     TEST T1; /*MEASURE COUNTER OUTPUT FOR 256 COUNTS*/

   3         MEAS;

   4           CONJ: <T1,T2> = DIG_SRS(STIM(1),RESP(1),512,8,50 KHZ,
   4                                        3.5 V,0.5 V,2 V,1 V)
   4                          TARGET: RESP(1);

   5           ASSERT: IF I=1 THEN STIM(I) = B'0'
   5                           ELSE STIM(I) = "^ STIM(I-1)
   5                           TARGET: STIM(I);

   6           ASSERT: I=SUBS('STIM,RESP',512) TARGET: I;

   7       FUNC: DIG_SRS,  /*DIGITAL STIMULI_RESPONSE_SAVE*/
   7             TYPE = M,
   7             PARAM=(P01,S,DIG),     /*STIMULI VECTOR 1ST ELEMENT*/
   7             PARAM=(P02,T,DIG),     /*RESPONSE VECTOR 1ST ELEMENT*/
   7             PARAM=(P03,S,DEC),     /*MESSAGE LENGTH*/
   7             PARAM=(P04,S,DEC),     /*NUMBER OF BITS/WORD*/
   7             PARAM=(P05,S,DEC),     /*FREQUENCY OF SIGNAL APPLICATION N KHZ*/
   7             PARAM=(P06,S,DEC),     /*ONE LEVEL*/
   7             PARAM=(P07,S,DEC),     /*ZERO LEVEL*/
   7             PARAM=(P08,S,DEC),     /*MIN ONE LEVEL*/
   7             PARAM=(P09,S,DEC),     /*MAX ZERO LEVEL*/
   7             REQUIRE= DO DIGITAL_TEST,
   7                         STIM_RESP_SAVE,
   7                         STIM (P01),
   7                         RESP (P02),
   7                         MSG_LENGTH (P03) WORDS,
   7                         WORD_LENGTH (P04) BITS,
   7                         WORD_RATE (P05) KHZ,
   7                         VOLTAGE_ONE (P06) V,
   7                         VOLTAGE_ZERO (P07) V,
   7                         VOLTAGE_ONE MIN (P08) V,
   7                         VOLTAGE_ZERO MAX (P09) V,
   7                         CNX_STIM (CNX01),
   7                         CNX_RESP (CNX02);
```

FIG. II.9 (CONTD.)

```
 8     TEST T2;

 9         MEAS;

10           ASSERT: IF(RESP(2*J)=DIG_ADD(RESP(2*J-1),H'01',8))
10                   THEN COND(J)=TRUE
10                   ELSE COND(J) = FALSE
10                   TARGET: COND(J);

11           ASSERT: J=SUBS('COND',256) TARGET: J;

12           ASSERT: IF(COND(J)=FALSE) | (J=256) THEN
12                   END_J = TRUE   TARGET:END_J;


13           ASSERT: COUNT = LAST(COND(J),J)
13                   TARGET:COUNT;
                     /*TEST PASSING CONDITION*/

14           ASSERT: COND(J) = TRUE;

15         LOGIC: | COUNT_PASS, |^COUNT_FAIL;

16         DIAG COUNT_PASS: PARAM=('PASSED',COUNT),
16                          PRINT = M;

17         DIAG COUNT_FAIL: PARAM=('FAILED',COUNT),COMP= FAIL(COUNTING),
17                          PRINT = M;

18         MESSAGE M: TEXT = 'TEST(P01),
18                            LAST COUNT(P02) B"########" ';

19     COMP_FAIL : COUNTING, FAILURE FUNCTION=FAIL;
20     FUNC: DIG_ADD,
20           TYPE = E,
20           PARAM = (P01,S,DIG), /*FIRST OPERAND*/
20           PARAM = (P02,S,DIG), /*2ND OPERAND*/
20           PARAM = (P03,S,INT), /*NUMBER OF BITS*/
20           VALUE = DIG;        /*RESULT OF ADDITION*/

21     FUNC: TRUE;
22     FUNC: FALSE;
23     FUNC: SUBS;
24     FUNC: FAIL, TYPE=F;
25     FUNC: LAST;
26     UUT_P: T1;
27     UUT_P: T2;
```

FIG. II.10 : REFORMATTED REPORT FOR EXAMPLE 2

```
/* REFORMATTED SPECIFICATION REPORT, FILE: SOURCE2 */


   /************************************************/
   /*                                              */
   /*  NOPAL TEST SPECIFICATION FOR DIGITAL_TEST   */
   /*                                              */
   /************************************************/

NOPAL SPECIFICATION DIGITAL_TEST;

   /************************************************/
   /*                                              */
   /*  TEST MODULES:     2                         */
   /*                                              */
   /************************************************/


  TEST T1;

     /* NULL STIMULUS */

     MEASUREMENT $M_T1(T1);

        CONJUNCTION $M_W0001($M_T1):
            (<T1, T2>  =
            DIG_SRS(STIM(1),RESP(1),512,8,50 KHZ ,3.5 V ,0.5 V ,2 V ,1 V ))
                    TARGET: RESP(1)
                    SOURCE: STIM(1);

        ASSERTION $M_W0002($M_T1):
            IF I=1 THEN
                STIM(I) =  B''0''
            ELSE
                STIM(I) =  STIM(I-1)
                    TARGET: STIM(I)
                    SOURCE: STIM(I-1), I;

        ASSERTION $M_W0003($M_T1):
            I =  SUBS('STIM,RESP',512)
                    TARGET: I
                    SOURCE: SUBS;

     /* NULL LOGIC */

  TEST T2;

     /* NULL STIMULUS */

     MEASUREMENT $M_T2(T2);
```

FIG. II.10 (CONTD.)

```
    ASSERTION $M_W0001($M_T2):
        IF (RESP(2*J)=DIG_ADD(RESP(2*J-1),H''01'',8)) THEN
            COND(J) =  TRUE
        ELSE
            COND(J) =  FALSE
                TARGET: COND(J)
                SOURCE: TRUE, RESP(2*J-1), RESP(2*J), J, FALSE, DIG_ADD(
                        RESP(2*J-1), H''01'', 8);

    ASSERTION $M_W0002($M_T2):
        J =  SUBS('COND',256)
                TARGET: J
                SOURCE: SUBS;

    ASSERTION $M_W0003($M_T2):
        IF (COND(J)=FALSE)|(J=256) THEN
            END_J =  TRUE
                TARGET: END_J
                SOURCE: TRUE, J, FALSE, COND(J);

    ASSERTION $M_W0004($M_T2):
        COUNT =  LAST(COND(J),J)
                TARGET: COUNT
                SOURCE: LAST(COND(J), J), J, COND(J);

    ASSERTION $M_W0005($M_T2):
        COND(J) =  TRUE
                SOURCE: TRUE, J, COND(J);

  LOGIC $LOGIC0010(T2): |COUNT_PASS, |^COUNT_FAIL;

    DIAGNOSIS COUNT_PASS:
        OPERATOR MESSAGE:
            OTHER PARAMETERS=('PASSED', COUNT),
            PRINT=M;

    DIAGNOSIS COUNT_FAIL:
        OPERATOR MESSAGE:
            AFFECTED COMPONENTS=FAIL(COUNTING),
            OTHER PARAMETERS=('FAILED', COUNT),
            PRINT=M;

 /*************************************************/
 /*                                               */
 /*  MESSAGES                                     */
 /*                                               */
 /*************************************************/

MESSAGE M:
   TEXT=                         LAST COUNT(P02) B"########" ';

 /*************************************************/
```

FIG. II.10 (CONTD.)

```
COMP_FAIL 00100: COUNTING, FAILURE FUNCTION=FAIL;

  /************************************************/
  /*                                              */
  /*  UUT CONNECTION POINTS                       */
  /*                                              */
  /************************************************/


UUT_POINT       : T1;

UUT_POINT       : T2;

  /************************************************/
  /*                                              */
  /*  ATE FUNCTIONS                               */
  /*                                              */
  /************************************************/


FUNCTION        : DIG_SRS, FUNCTION TYPE=M, #PINS= 2,
        PARAM#01=(P01, S),
        PARAM#02=(P02, T),
        PARAM#03=(P03, S),
        PARAM#04=(P04, S),
        PARAM#05=(P05, S),
        PARAM#06=(P06, S),
        PARAM#07=(P07, S),
        PARAM#08=(P08, S),
        PARAM#09=(P09, S),
        REQUIRE = DO DIGITAL-TEST, STIM-RESP-SAVE, STIM 'P01', RESP 'P02', MS-
G-LENGTH 'P03' WORDS, WORD-LENGTH 'P04' BITS, WORD-RATE 'P05' KHZ, VOLTAGE-ONE '
P06' V, VOLTAGE-ZERO 'P07' V, VOLTAGE-ONE MIN 'P08' V, VOLTAGE-ZERO MAX 'P09' V,
 CNX-STIM 'DIG-SRS.CNX01;,

FUNCTION        : FAIL, FUNCTION TYPE=F;

FUNCTION        : DIG_ADD, FUNCTION TYPE=E,
        PARAM#01=(P01, S),
        PARAM#02=(P02, S),
        PARAM#03=(P03, S);

FUNCTION        : TRUE, FUNCTION TYPE=E;

FUNCTION        : FALSE, FUNCTION TYPE=E;

FUNCTION        : SUBS, FUNCTION TYPE=E;

FUNCTION        : LAST, FUNCTION TYPE=E;

END DIGITAL_TEST;
```

FIG. II.11 : CROSS REFERENCE AND ATTRIBUTES REPORT FOR EXAMPLE 2

NAME	DEF NO.	ATTRIBUTES AND REFERENCES		DATA TYPE
COND	10	VARIABLE ID 12 13 14	, 1-DIMENSIONAL ARRAY	DECIMAL
COUNT	13	VARIABLE ID 16 17		DECIMAL
COUNTING	19	COMPONENT ID 17	, WITH FAILURE-FUNCTION: FAIL	-------
COUNT_FAIL	17	DIAGNOSIS LABEL 15		-------
COUNT_PASS	16	DIAGNOSIS LABEL 15		-------
DIGITAL_TEST	1	SPECIFICATION LABEL		-------
DIG_ADD	20	ATE-FUNCTION ID 10	, E	DECIMAL
DIG_SRS	7	ATE-FUNCTION ID 4	, M	DECIMAL
END_J	12	VARIABLE ID	, GLOBAL	DECIMAL
FAIL	24	ATE-FUNCTION ID 17 19	, F	DECIMAL
FALSE	22	ATE-FUNCTION ID 12 10	, E	DECIMAL
I	6	VARIABLE ID 5		INTEGER
J	11	VARIABLE ID 10 12 13 14		INTEGER
LAST	25	ATE-FUNCTION ID 13	, E	DECIMAL
M	18	MESSAGE LABEL 16 17		-------
RESP	4	VARIABLE ID 10	, 1-DIMENSIONAL ARRAY, GLOBAL	DECIMAL
STIM	5	VARIABLE ID 4	, 1-DIMENSIONAL ARRAY	DIGITAL
SUBS	23	ATE-FUNCTION ID 11 6	, E	INTEGER
T1	2	TEST LABEL 3		-------
T1	26	UUT-POINT ID 4		-------
T2	27	UUT-POINT ID 4		-------
T2	8	TEST LABEL 9 15		-------
TRUE	21	ATE-FUNCTION ID 14 12 10	, E	DECIMAL

FIG. II.12 : CROSS REFERENCES -- TESTS VS DIAGNOSES VS MESSAGES VS TEST POINTS VS FUNCTIONS -- FOR EXAMPLE 2

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- DIAGNOSES <=> TEST-MODULES

DIAGNOSES	TEST-MODULES
COUNT_PASS	T2,
COUNT_FAIL	T2,

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- MESSAGES <=> DIAGNOSES <=> TESTS

MESSAGE	DIAGNOSES	TEST-MODULES
M	COUNT_PASS, COUNT_FAIL,	T2,

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- AFFECTED-COMPONENTS <=> DIAGNOSES <=> TESTS

AFFECTED-COMPONENT	DIAGNOSES	TEST-MODULES
00100: FAIL(COUNTING)	COUNT_FAIL,	T2,

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- UUT-CONNECTING-POINTS <=> TEST-MODULES <=> ATE-CONNECTING-POINTS

UUT-CONNECTING-POINT	TEST-MODULES(S/M)	ATE-CONNECTING-POINTS
T1	T1(M),	
T2	T1(M),	

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- ATE-FUNCTIONS <=> TEST-MODULES

ATE-FUNCTION,TYPE	TEST-MODULES(S/M)
DIG_SRS, M	T1(M),
DIG_ADD, E	
TRUE, E	
FALSE, E	
SUBS, E	
LAST, E	

FIG. II.13 : FLOWCHART REPORT FOR NOPAL SPECIFICATION DIGITAL_TEST

```
/***** FLOWCHART FOR GLOBAL PROCEDURE DIGITAL_TEST *****/
  BEGIN DIGITAL_TEST;
       DECLARE AS GLOBAL VARIABLES:
            RESP(  512), END_J;
       DECLARE AS EVAL OR CONTROL FUNCTIONS:
            DIG_ADD, LAST;
       DECLARE AS STIMULUS FUNCTIONS:
            /* NO STIMULUS FUNCTIONS */
       DECLARE AS MEASUREMENT FUNCTIONS:
            DIG_SRS;
       DECLARE AS FAILURE FUNCTIONS:
            FAIL;
       PERFORM TEST T1;
       PERFORM TEST T2;
  END DIGITAL_TEST;
```

FIG. II.14 : SEQUENCE REPORT FOR EXAMPLE 2

WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION DIGITAL_TEST

			1	2	3	4	5	6	7	8	9	10	11
1	T1	TEST	0	0	0	0	1	0	0	0	0	0	0
2	T2	TEST	0	0	12	13	0	0	0	0	0	1	0
3	COUNT_PASS	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0
4	COUNT_FAIL	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0
5	RESP	VARIABLE	0	1	0	0	0	0	0	0	0	0	0
6	SUBS	FUNCTION	1	1	0	0	0	0	0	0	0	0	0
7	TRUE	FUNCTION	0	1	0	0	0	0	0	0	0	0	0
8	FALSE	FUNCTION	0	1	0	0	0	0	0	0	0	0	0
9	DIG_ADD	FUNCTION	0	1	0	0	0	0	0	0	0	0	0
10	END_J	VARIABLE	0	0	0	0	0	0	0	0	0	0	0
11	LAST	FUNCTION	0	1	0	0	0	0	0	0	0	0	0

SEQUENCE OF PROCESSING FOR NOPAL SPECIFICATION DIGITAL_TEST

ORDER VECTOR INDEX	ORDER VECTOR	NAME	TYPE
1	11	LAST	EVAL OR CONTROL FUNCTION
2	6	SUBS	EVAL OR CONTROL FUNCTION
3	9	DIG_ADD	EVAL OR CONTROL FUNCTION
4	8	FALSE	EVAL OR CONTROL FUNCTION
5	7	TRUE	EVAL OR CONTROL FUNCTION
6	1	T1	TEST
7	5	RESP	GLOBAL VARIABLE
8	2	T2	TEST
9	3	COUNT_PASS	DIAGNOSIS
10	4	COUNT_FAIL	DIAGNOSIS
11	10	END_J	GLOBAL VARIABLE

FIG. II.14 (CONTD.)

INTRA TEST SEQUENCING FOR MODULE T1
ANALYSIS OF THE ADJACENCY MATRIX

			1	2	3	4	5	6	7
1	$M_W0001	CONJUNCTION	0	0	0	1	0	0	0
2	$M_W0002	ASSERTION	0	0	0	0	1	0	0
3	$M_W0003	ASSERTION	0	0	0	0	0	1	0
4	RESP	VARIABLE	0	0	0	0	0	0	0
5	STIM	VARIABLE	1	0	0	0	0	0	0
6	I	VARIABLE	0	1	0	1	1	0	0
7	SUBS	FUNCTION	0	0	1	0	0	0	0

SEQUENCE OF PROCESSING FOR TEST MODULE T1

ORDER VECT INDEX	ORDER VECTOR	RANK	NAME	TYPE	TEXT
1	7	0	SUBS	FUNCTION	
2	3	1	$M_W0003	ASSERTION	I = SUBS('STIM,RESP',512) TARGET: I SOURCE: SUBS;
3	6	2	I	VARIABLE	LOCAL
LOOP-1 STARTS: SUBSCRIPT I ITERATES FROM 1 TO 512					
4	2	3	$M_W0002	ASSERTION	IF I=1 THEN STIM(I) = B''O'' ELSE STIM(I) = STIM(I-1) TARGET: STIM(I) SOURCE: STIM(I-1), I;
5	5	4	STIM	SUBSCRIPT_VA	LOCAL
LOOP-1 ENDS;					
6	1	5	$M_W0001	CONJUNCTION	(<T1, T2> = DIG_SRS(STIM(1),RESP(1),512,8,50 KHZ ,3.5 V ,0.5 V ,2 V ,1 V)) TARGET: RESP(1) SOURCE: STIM(1);
LOOP-2 STARTS: SUBSCRIPT I ITERATES FROM 1 TO 512					
7	4	6	RESP	SUBSCRIPT_VA	GLOBAL / TARGET /
LOOP-2 ENDS;					

FIG. II.14 (CONTD.)

INTRA TEST SEQUENCING FOR MODULE T2
ANALYSIS OF THE ADJACENCY MATRIX

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	$M_W0001	ASSERTION	0	0	0	0	01717			1	0	0	0	0	0	0	0	0	0
2	$M_W0002	ASSERTION	0	0	0	0	01717			0	0	0	1	0	0	0	0	0	0
3	$M_W0003	ASSERTION	0	0	0	0	01717			0	0	0	0	0	0	0	1	0	0
4	$M_W0004	ASSERTION	0	0	0	0	01717			0	0	0	0	0	0	0	0	1	0
5	$M_W0005	ASSERTION	0	0	0	0	01717			0	0	0	0	0	0	0	0	0	0
6	COUNT_PASS	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	COUNT_FAIL	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	COND	VARIABLE	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9	TRUE	FUNCTION	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
10	RESP	VARIABLE	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	J	VARIABLE	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
12	FALSE	FUNCTION	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	DIG_ADD	FUNCTION	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	SUBS	FUNCTION	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	END_J	VARIABLE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	COUNT	VARIABLE	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
17	LAST	FUNCTION	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

SEQUENCE OF PROCESSING FOR TEST MODULE T2

ORDER VECT INDEX	ORDER VECTOR	RANK	NAME	TYPE	TEXT
1	9	0	TRUE	FUNCTION	
2	10	0	RESP	SUBSCRIPT_VA	GLOBAL / SOURCE /
3	12	0	FALSE	FUNCTION	
4	13	0	DIG_ADD	FUNCTION	
5	14	0	SUBS	FUNCTION	
6	2	1	$M_W0002	ASSERTION	J = SUBS('COND',256) TARGET: J SOURCE: SUBS;

FIG. II.14 (CONTD.)

```
 7    11     2     J              VARIABLE        LOCAL


 8    17     3     LAST           FUNCTION


LOOP-1 STARTS: SUBSCRIPT J ITERATES FROM 1 TO   256
 9     1     4     $M_W0001       ASSERTION       IF (RESP(2*J)=DIG_ADD(RESP(2*J-1),H''01'',8)) THEN   COND(J)   =  TRUE
                                                  ELSE   COND(J)   =  FALSE
                                                  TARGET:  COND( J)
                                                  SOURCE:  TRUE,  RESP( 2*J-1),  RESP( 2*J),  J,  FALSE,  DIG_ADD(
                                                  RESP(2*J-1),  H''01'',  8);


10     8     5     COND           SUBSCRIPT_VA    LOCAL


11     3     6     $M_W0003       ASSERTION       IF (COND(J)=FALSE)|(J=256) THEN   END_J   =  TRUE
                                                  TARGET:  END_J
                                                  SOURCE:  TRUE,  J,  FALSE,  COND( J);


12     4     6     $M_W0004       ASSERTION       COUNT =  LAST(COND(J),J)
                                                  TARGET:  COUNT
                                                  SOURCE:  LAST( COND(J),  J),  J,  COND( J);


13     5     6     $M_W0005       ASSERTION       COND(J) =  TRUE
                                                  SOURCE:  TRUE,  J,  COND( J);

LOOP-1 ENDS;

14    15     7     END_J          VARIABLE        GLOBAL / TARGET /


15    16     7     COUNT          VARIABLE        LOCAL


16     6     8     COUNT_PASS     DIAGNOSIS       OTHER PARAMETERS = ('PASSED',COUNT) PRINT = M;


17     7     8     COUNT_FAIL     DIAGNOSIS       AFFECTED COMPONENTS = FAIL(COUNTING), OTHER PARAMETERS = ('FAILED',
                                                  COUNT) PRINT = M;



LOOP SUMMARY TABLE :
   LOOP-1     FIRST NODE IS    9      LAST NODE IS   13      SUBSCRIPT IS J
```

FIG. II.15 : ERROR AND WARNING REPORT FOR EXAMPLE 2

```
 ERROR/WARNING MESSAGES GENERATED DURING NOPAL SYNTAX ANALYSIS:
  *STATISTICS* NO. OF SAP ERRORS =    0  ,   NO. OF WARNINGS =    0  , NO. OF STATEMENTS=   27


ERROR/WARNING MESSAGES GENERATED DURING CROSS-REFERENCE:
**WARNING** XRF3:  COUNT IS USED AS A LOCAL VARIABLE IN TEST T2
            AND ALSO IN DIAGNOSES COUNT_PASS ,COUNT_FAIL

*STATISTICS*  NO. OF XREF1 ERRORS =    0     NO. OF WARNINGS =    1


 ERROR/WARNING MESSAGES GENERATED DURING SEQUENCING AND CODE GENERATION:


**WARNING** ESQ9: (POSSIBLE INCOMPLETENESS): TEST MODULE T1 DOES NOT HAVE ANY DIAGNOSIS.


WARNING MESSAGES GENERATED DURING CODE GENERATION
**WARNING** INI1: UUT POINT 'T2' NOT CONNECTED TO ANY ATE PIN
**WARNING** INI1: UUT POINT 'T1' NOT CONNECTED TO ANY ATE PIN
**WARNING** FLB3: SYSLIB DOES NOT EXIST  SHOULD BE INCLUDED AT ATLAS COMPILE TIME.
**WARNING** FLB3: RUNLIB DOES NOT EXIST  SHOULD BE INCLUDED AT ATLAS COMPILE TIME.

**WARNING** CYC1: (CIRCULAR DEP): THE FOLLOWING GROUP OF ITEMS IN TEST T1 ARE CIRCULARLY DEFINED:
                 $M_W0002 (ROW    2), STIM (ROW    5), $M_W0002 (ROW    2)
**WARNING** CYC3: CYCLE MENTIONED ABOVE HAS BEEN ELIMINATED
  BY CHANGING ENTRY  5, 2) FROM 21 TO 0 IN ADJACENCY MATRIX.

  *STATISTICS* NO. CODE GENERATION ERRORS =    0, NO. OF WARNINGS=    4

  *STATISTICS* NO. OF SEQUENCING ERRORS=    0, NO. OF WARNINGS=    3
```

FIG. II.16 : ATLAS PROGRAM FOR EXAMPLE 2

```
          BEGIN EQUATE PROGRAM 'DIGITAL-TEST' $
C            NAMES                     INDEX
C             *** TEST MODULES ***
C             'T1'                --  1
C             'T2'                --  2
C
C             *** DIAGNOSES ***
C             'COUNT-PASS'        --  1
C             'COUNT-FAIL'        --  2
C
C             *** AFFECTED COMPONENTS ***
C              FAIL(COUNTING)              --  1
C     $
C     DECLARATIONS OF SYSTEM VARIABLES  $
          DECLARE DIGITAL, LIST,'SYS.DIAG-FLAG'(2) $
          DECLARE DECIMAL, 'SYS.S-TIME' $
          DECLARE DECIMAL, 'SYS.D-TIME' $
          DECLARE DECIMAL, 'SYS.DUMMY' $
          DECLARE DECIMAL, 'SYS.NAME' $
          DECLARE DECIMAL, LIST,'SYS.#TESTS IN CONJ'(2) $
          DECLARE DIGITAL, LIST, 'SYS.TEST-FLAG'(2) $
          DECLARE DIGITAL, 'SYS.FLAG' $
          DECLARE DIGITAL, 'SYS.ASRT-FLAG' $
          DECLARE DECIMAL, 'SYS.TIM', 'SYS.TIME' $
          DECLARE DECIMAL, LIST, 'SYS.CLOCK'(6)  $
          DECLARE DECIMAL, 'SYS.I' $
          DECLARE DIGITAL, 'SYS.Y/N' $
          DECLARE DIGITAL, 'SYS.STATE' $
          DECLARE DIGITAL, 'SYS.SELECT' $
          DECLARE DECIMAL, LIST, 'AFF-COMP.NAME'(1) $
          DECLARE DIGITAL, LIST, 'AFF-COMP.SELECT'(1) $
          DECLARE DECIMAL, LIST, 'AFF-COMP.WHERE'(1) $
          DECLARE DIGITAL, LIST, 'AFF-COMP.STATE'(1) $
          DECLARE DECIMAL, 'AFF-COMP.TEST' $
          DECLARE DECIMAL, 'AFF-COMP.COUNT' $
          DECLARE DIGITAL, 'AFF-COMP.CHANGE' $
C     *** CONSTANTS ***  $
          DEFINE 'SYS.SELECTED' , B'10' $
          DEFINE 'SYS.NOT SELECTED' , B'01' $
          DEFINE 'SYS.NOT TESTED' , B'00' $
          DEFINE 'SYS.TESTED' , B'10' $
          DEFINE 'SYS.SKIPPED' , B'01' $
          DEFINE 'SYS.#DIAGS' , 2 $
          DEFINE 'SYS.#TESTS' , 2 $
          DEFINE 'SYS.TRUE' , B'1' $
          DEFINE 'SYS.FALSE' , B'0' $
          DEFINE 'SYS.DONT KNOW', B'00' $
          DEFINE 'SYS.IS', B'001' $
          DEFINE 'SYS.IS NOT', B'010' $
          DEFINE 'SYS.MAY BE', B'011' $
          DEFINE 'SYS.MAY BE NOT', B'100' $
          DEFINE 'SYS.#COMPONENTS',1 $
```

FIG. II.16 (CONTD.)

```
C     UUT POINT DEFINITIONS      $
C     FOLLOWING DISK FILE SHOULD CONTAIN THE
C     MISSING EQUATE/UUT PIU & DIU ASSIGNMENTS. $
          INCLUDE "CNXDIG" $
C     MACRO DEFINITIONS $
          DEFINE 'PRT.TIME', 'SYS.CLOCK'(1), " ##:",
                 'SYS.CLOCK'(2), "##:", 'SYS.CLOCK'(3), "##" $
C     DECLARATIONS FOR USER DEFINED GLOBAL VARIABLES    $
          DECLARE DECIMAL, LIST, 'RESP' (512) $
          DECLARE DIGITAL,  'END-J'  $
C     SYSTEM UTILITY ROUTINES  $
          DEFINE PROCEDURE, 'GET.TIME' $
            READ(TIME 'SYS.CLOCK'(1) ALL), SYS-CLOCK $
            'SYS.TIME' = 3600*'SYS.CLOCK'(1) + 60*'SYS.CLOCK'(2) +
                   'SYS.CLOCK'(3) $
          END 'GET.TIME' $
          DECLARE DECIMAL, 'DIG-SRS.CNX01' $
          DECLARE DECIMAL, 'DIG-SRS.CNX02' $
          DECLARE DIGITAL, LIST,'STIM.1' (512) $
          DECLARE INTEGER, 'I.1'  $
          DECLARE DIGITAL, 'SYS.DIG.01' $
          DECLARE DIGITAL, 'END-I.1' $
          DECLARE DECIMAL, 'SYS.DEC.01' $
          DECLARE DECIMAL, LIST,'COND.2' (256) $
          DECLARE INTEGER, 'J.2'  $
          DECLARE DECIMAL, 'COUNT.2'  $
          DECLARE DIGITAL, 'SYS.DIG.02' $
          DECLARE DIGITAL, 'SYS.DIG.03' $
          DECLARE DIGITAL, 'SYS.DIG.04' $
          DECLARE DECIMAL, 'SYS.DEC.02' $
          DECLARE DIGITAL, 'END-J.2' $
          DECLARE DECIMAL, 'D001.P02' $
          DECLARE DECIMAL, 'D002.P02' $
          INCLUDE "SYSLIB" $
          DEFINE PROCEDURE, 'AFF-COMP.PRINT'$
 300      RECORD 'AFF-COMP.COUNT',"###: FAIL(COUNTING)"$
 305      END 'AFF-COMP.PRINT'$
          INCLUDE "RUNLIB" $
C***************************************************************************$
C***************************************************************************$
C     DIAGNOSES PROCS $
C***************************************************************************$
C***************************************************************************$
 1000     DEFINE PROCEDURE, 'COUNT-PASS' $
          RECORD "                        LAST COUNT(P02) B''########'' " $
          'SYS.DIAG-FLAG'(1) = 'SYS.SELECTED' $
          END 'COUNT-PASS' $
C***************************************************************************$
 1100     DEFINE PROCEDURE, 'COUNT-FAIL' $
          'AFF-COMP.COUNT' = 1 $
          'AFF-COMP.NAME'(1) = 1 $
          'AFF-COMP.SELECT'(1) = 'SYS.MAY BE' $
```

FIG. II.16 (CONTD.)

```
       PERFORM 'AFF-COMP.UPDATE' $
       RECORD "                         LAST COUNT(P02) B''########'' " $
       'SYS.DIAG-FLAG'(2) = 'SYS.SELECTED' $
       END 'COUNT-FAIL' $
C***************************************************************************$
C***************************************************************************$
C    TEST PROCS $
C***************************************************************************$
C***************************************************************************$
 1200   DEFINE PROCEDURE, 'T1' $
        'SYS.FLAG' = 'SYS.TRUE' $
        'SYS.ASRT-FLAG' = 'SYS.TRUE' $
        'AFF-COMP.TEST' = 1 $
        'END-I.1' = 'SYS.TRUE' $
        FOR 'I.1' = 1 THRU 512 BY 1 THEN $
            COMPARE 'I.1', EQ 1 $
            GOTO STEP  1205      IF NOGO $
            'SYS.DIG.01' = 'SYS.TRUE' $
            GOTO STEP  1210      $
 1205       'SYS.DIG.01' = 'SYS.FALSE' $
 1210       COMPARE 'SYS.DIG.01', EQ 'SYS.TRUE' $
            GOTO STEP  1215      IF NOGO $
            'STIM.1'('I.1') = B'0' $
            GOTO STEP  1220      $
 1215       'STIM.1'('I.1') = 'STIM.1'(('I.1'- 1)) $
 1220       COMPARE 'END-I.1', EQ  'SYS.TRUE' $
            GOTO STEP  1225      IF NOGO $
        END FOR $
 1225   DEFINE 'DIG-SRS.CNX01', 'T1' $
        DEFINE 'DIG-SRS.CNX02', 'T2' $
         DO DIGITAL-TEST, STIM-RESP-SAVE, STIM 'STIM.1'(1)  RESP 512
           MSG-LENGTH 8 WORDS, WORD-LENGTH 50 BITS, WORD-RATE 3.5 KHZ,
           VOLTAGE-ONE 0.5 V, VOLTAGE-ZERO 2 V, VOLTAGE-ONE MIN 1 V,
           VOLTAGE-ZERO MAX 'RESP'(1) V, CNX-STIM 'DIG-SRS.CNX01',
           CNX-RESP 'DIG-SRS.CNX02' $
 1230   'SYS.TEST-FLAG'(1) = 'SYS.TESTED' $
        END 'T1' $
C***************************************************************************$
 1300   DEFINE PROCEDURE, 'T2' $
        'SYS.FLAG' = 'SYS.TRUE' $
        'SYS.ASRT-FLAG' = 'SYS.TRUE' $
        'AFF-COMP.TEST' = 2 $
        'END-J.2' = 'SYS.TRUE' $
        FOR 'J.2' = 1 THRU 256 BY 1 THEN $
            'DIG-ADD.PRM01' = 'RESP'(((2* 'J.2')- 1)) $
            'DIG-ADD.PRM02' = X'01' $
            'DIG-ADD.PRM03' = 8 $
 1305       PERFORM 'DIG-ADD' $
            'SYS.DEC.01' = 'DIG-ADD.RES' $
            COMPARE 'RESP'((2* 'J.2')), EQ 'SYS.DEC.01' $
            GOTO STEP  1310      IF NOGO $
            'SYS.DIG.01' = 'SYS.TRUE' $
```

FIG. II.16 (CONTD.)

```
 1310         'SYS.DIG.01' = 'SYS.FALSE' $
 1315         COMPARE 'SYS.DIG.01', EQ 'SYS.TRUE' $
              GOTO STEP  1320      IF NOGO $
              'COND.2'('J.2') = 'SYS.TRUE' $
              GOTO STEP  1325      $
 1320         'COND.2'('J.2') = 'SYS.FALSE' $
 1325         COMPARE 'COND.2'('J.2'), EQ 'SYS.FALSE' $
              GOTO STEP  1330      IF NOGO $
              'SYS.DIG.02' = 'SYS.TRUE' $
              GOTO STEP  1335      $
 1330         'SYS.DIG.02' = 'SYS.FALSE' $
 1335         COMPARE 'J.2', EQ 256 $
              GOTO STEP  1340      IF NOGO $
              'SYS.DIG.03' = 'SYS.TRUE' $
              GOTO STEP  1345      $
 1340         'SYS.DIG.03' = 'SYS.FALSE' $
 1345         COMPARE B'1', EQ 'SYS.DIG.02' OR 'SYS.DIG.03' $
              GOTO STEP  1350      IF NOGO $
              'SYS.DIG.04' = 'SYS.TRUE' $
              GOTO STEP  1355      $
 1350         'SYS.DIG.04' = 'SYS.FALSE' $
 1355         COMPARE 'SYS.DIG.04', EQ 'SYS.TRUE' $
              GOTO STEP  1360      IF NOGO $
              'END-J' = 'SYS.TRUE' $
              GOTO STEP  1360      $
 1360         'LAST.PRM01' = 'COND.2'('J.2') $
              'LAST.PRM02' = 'J.2' $
 1365         PERFORM 'LAST' $
              'SYS.DEC.02' = 'LAST.RES' $
              'COUNT.2' = 'SYS.DEC.02' $
              COMPARE 'COND.2'('J.2'),  EQ 'SYS.TRUE' $
              GOTO STEP  1370      IF GO $
              'SYS.FLAG' = 'SYS.FALSE' $
              'SYS.ASRT-FLAG' = 'SYS.FALSE' $
 1370         COMPARE 'END-J.2', EQ  'SYS.TRUE' $
              GOTO STEP  1375      IF NOGO $
          END FOR $
 1375     COMPARE 'SYS.ASRT-FLAG', EQ 'SYS.TRUE' $
          GOTO STEP  1380      IF NOGO $
          'D001.P02' = 'COUNT.2' $
          PERFORM 'COUNT-PASS'  $
 1380     COMPARE 'SYS.ASRT-FLAG', EQ 'SYS.TRUE' $
          GOTO STEP  1385      IF GO $
          'D002.P02' = 'COUNT.2' $
          PERFORM 'COUNT-FAIL'  $
 1385     'SYS.TEST-FLAG'(2) = 'SYS.TESTED' $
          END 'T2' $
C*****************************************************************************$
C*****************************************************************************$
C    SYSTEM VARIABLE INITIALIZATION AND FIRST ENTRY POINT    $
C*****************************************************************************$
C*****************************************************************************$
```

FIG. II.16 (CONTD.)

```
E  1400    PERFORM 'GET.TIME'  $
          RECORD "!X TESTING UUT: DIGITAL-TEST" $
          RECORD  'SYS.CLOCK'(4), "DATE ##/",'SYS.CLOCK'(5), "##/" ,
                  'SYS.CLOCK'(6), "## TIME", 'PRT.TIME' $
          'SYS.TIM' = 'SYS.TIME' $
          FOR 'SYS.I' = 1 THRU 'SYS.#DIAGS' THEN $
             'SYS.DIAG-FLAG'('SYS.I')= 'SYS.NOT SELECTED' $
             'SYS.#TESTS IN CONJ'('SYS.I') = 0 $
          END FOR  $
          FOR 'SYS.I' = 1 THRU 'SYS.#COMPONENTS' THEN $
             'AFF-COMP.STATE'('SYS.I') = 'SYS.DONT KNOW' $
             'AFF-COMP.WHERE'('SYS.I') = 0 $
          END FOR $
          FOR 'SYS.I' = 1 THRU 'SYS.#TESTS' THEN $
              'SYS.TEST-FLAG'('SYS.I') = 'SYS.NOT TESTED' $
          END FOR $
C    BEGINNING OF TESTING      $
C    $
C    CONTROL PRECEDING THE CALL ON THE TEST MODULE 'T1' $
 1500     'SYS.FLAG' = 'SYS.TRUE' $
          PERFORM 'T1' $
C $
C    CONTROL PRECEDING THE CALL ON THE TEST MODULE 'T2' $
 1600     'SYS.FLAG' = 'SYS.TRUE' $
          PERFORM 'T2' $
C $
 1700     PERFORM 'GET.TIME' $
          RECORD "FINISHED TESTING AT", 'PRT.TIME' $
          'SYS.TIM' = 'SYS.TIME' - 'SYS.TIM' $
          'SYS.CLOCK'(1) = INT ('SYS.TIM'/3600) $
          'SYS.TIM' = 'SYS.TIM' - 3600*'SYS.CLOCK'(1) $
          'SYS.CLOCK'(2) = INT('SYS.TIM'/60) $
          'SYS.CLOCK'(3) = 'SYS.TIM' - 60*'SYS.CLOCK'(2) $
          RECORD "DURATION " , 'PRT.TIME' $
          REMOVE ALL  $
          RECORD "DO YOU WISH TO SEE THE FINAL FAULT ISOLATION STATE ? ",
             "(Y/N) " $
          WAIT-FOR MANUAL-DATA-GO-NOGO $
          GOTO STEP 1705 IF NOGO $
          PERFORM 'PRINT.DONT KNOW' $
          PERFORM 'PRINT.IS' $
          PERFORM 'PRINT.IS NOT' $
          PERFORM 'PRINT.MAY BE' $
          PERFORM 'PRINT.MAY BE NOT' $
 1705     RECORD "DO YOU WISH TO RERUN THIS PROGRAM? (Y/N)" $
          WAIT-FOR MANUAL-DATA-GO-NOGO $
          GOTO STEP 1710 IF NOGO $
          RECORD "!P" $
          GOTO STEP  1400    $
 1710     RECORD "TERMINATE EQUATE PROGRAM 'DIGITAL-TEST' !P" $
 1715     FINISH $
          TERMINATE EQUATE PROGRAM 'DIGITAL-TEST' $
```

APPENDIX III - ERROR AND WARNING MESSAGES IN THE NOPAL SYSTEM

III.1. INDEX TABLE:

NAME OF PROGRAM	ABBREVIATION	COMMENT
ATLASIO	AIO #	ATLAS I/O HANDLING
CDETEST	CDT #	GEN CODE FOR TESTS
CHECKER	CKR #	DATA TYPE CHECKING
CYCLES	CYC #	CYCLES DETECTING
EXTSCH	ESH #	EXTERNAL SCHEDULER
EXTSEQ	ESQ #	EXTERNAL SEQUENCING
FAILMAN	FLM #	SYNTAX ERROR CHECK
FUNCLIB	FLB #	USER FUNC LIB HANDLER
FUNTAB	FTB #	USER ATE FUN HANDLER
GENMSG	GMG #	CODE GEN FOR MSG
GENWAVE	GWV #	CODE GEN FOR WAVEFORM
INITIAL	INI #	CODE GEN INIT ATLAS
INTSEQ	ISQ #	INTERNAL SEQUENCING
ISPARM	ISP #	FUNC PARA HANDLING
LEX	LEX #	LEXICAL ANALYZER
MONITOR	MNT #	THE MAIN PROGRAM
REDUSAG	RDS #	REDUCTION ON SUB USE
RETREVS	RTV #	RETRIEVE ASS. MEMORY
SAP	SAP #	SYNTAX ANALYZER
SCHEDLR	SCD #	INTERNAL SCHEDULER
SETAREX	SAR #	SYNTAX ANAL (ARITH)
STLOG	STL #	STORE LOGIC
STSPEC	STS #	STORE SPECIFICATION
SUBANAL	SBL #	SUB USE ANALYSIS
SUBUSAG	SUS #	SUB USE CHECKING
STSTIM	STM #	STORE STIMULUS
XREF1	XRF #	CROSS REFERENCE 1
XREF2	XRT #	CROSS REFERENCE 2

Note:
The "#" sign above represents the corresponding numbers in respective error messages.

III.2. ERROR AND WARNING MESSAGES:

Note: The "^" sign will be used to indicate the place where the error or warning arises. For example if the declared data type of variable A is digital and it conflicts with the use of A, which depends on decimal variables B and C, then this is shown as

```
A = B + C .
^
```

"**WARNING** AIO1: NO BLANKS OUTSIDE SINGLE AND DOUBLE QUOTES IN: 'atlas stmt#'"

There is no blanks in the indicated position of the generated ATLAS statement.

"**WARNING** AIO2: UNABLE TO TRUNCATE PROPERLY. NO BLANKS. AFTER ATLAS STMT='atlas stmt #'"

This message indicates that the length of the generated ATLAS

statement is too long and there is no blank in the statement making the truncation improperly done.

"**SYSTEM IMPLEMENTATION ERROR** AIO3: PRINT BUFFER OVERFLOW!(TEXT > 1000)"

"**ERROR** CDT1: ATLAS SUPPORTS ONLY 1 DIM."

"**ERROR** CKR1: VARIABLE: " variable name " IN BOOLEAN EXPRESSION WITH TYPE NOT "BOOLEAN"; AT LINE: stmt #"

This only occures in the case when the assertion contains a variable as an

argument of a Boolean operator, i.e. "&", "|" or "^", but the data type of

the variable is not of Boolean.

EX.

```
DCL A: INT;
 ...
ASRT: IF A | B THEN C=120..;
```

The error message will be "**ERROR** VARIABLE: "A" IN... ".

"**ERROR** CKR2: BOOLEAN OPERATOR "^" HAS ITS ARGUMENT OF TYPE NOT "BOOLEAN".AT LINE: stmt #"

EX.

```
DCL A: DIGTAL;
 ...^
ASRT: IF ^A THEN B=0...; ...
         ^
```

"**ERROR** CKR3: LHS OF A BOOLEAN OPERATOR HAS DATA TYPE NOT "BOOLEAN";

```
  AT LINE:  stmt #"
EX.
  DCL A: DECIMAL;
      ^
  DCL B: BOOLEAN;
   ...
  ASRT: IF A & B THEN ...; ...
             ^
```

This is the similar error as above.

"**ERROR** CKR4: RHS OF A BOOLEAN OPERATOR HAS DATA TYPE NOT "BOOLEAN"; AT LINE: stmt #"

EX.

```
DCL A: BOOLEAN;
DCL B: INTEGER;
... ^
ASRT: IF A & B THEN ...; ...
               ^
```

This is self-explaninable.

"**ERROR** CKR5: LHS OF A RELATIONAL OPERATOR HAS DATA TYPE "BOOLEAN" OR

" DIGITAL"; AT LINE: stmt #"

EX.

```
DCL A: BOOLEAN(OR DIGITAL);
     ^
DCL B: INT;
 ...
ASRT: IF A <= B THEN ...; ...
         ^
```

Similar as above.

"**ERROR** CKR6: RHS OF A RALATIONAL OPERATOR HAS DATA TYPE "BOOLEAN" OR

"DIGITAL"; AT LINE: stmt #"

EX.

```
DCL A: INTEGER;
DCL B: BOOLEAN(OR DIGITAL);
 ...
ASRT: IF A > B THEN ...; ...
```

Note that there is no magnitute comparison facilities available between digital and integer or between digital and decimal numbers in NOPAL system.

"**ERROR** CKR7: LHS AND RHS HAVE INCOMPATIBLE DATA TYPES IN A BOOLEAN

EXPRESSION."

EX.

```
DCL A: INT;
DCL B: BOOL;
 ...
ASRT: IF A = B THEN ...; ...
 [ OR IF A ^=B THEN ...; ...]
```

Note that this message occurs only when the equal or not-equal sign is

used and it has different type on both sides.

"**ERROR** CKR8: LHS OF A RELATIONAL OPERATOR HAS ILLEGAL TYPE IN AN

ASSERTION; AT LINE: stmt #"

EX.
```
  DCL A: DIG;
  ...
  ASRT: A < 100...; ...
```

This is the error only in the use of relational assertion.

"**ERROR** CKR9: RHS OF A RALATIONAL OPERATOR HAS ILLEGAL TYPE IN AN

ASSERTION; AT LINE: stmt #"

EX.
```
  DCL A: INT;
  DCL B: DIG;
  ... ^
  ASRT: A >= B + 3 ...; ...
              ^
```

Here the error is obvious, i.e., the declaration of B is not compatible with the use of B. The operation "+" is only defined on the oprands of type "INT" or "DEC".

"**ERROR** CKR10: RHS AND LHS HAVE INCOMPATIBLE DATA TYPES IN TERMS OF AN OPERATOR: 'the corresponding operator' IN AN ASSERTION; AT LINE: stmt #"

EX.
```
  DCL A: INT;
  DCL B: DIG;
  ...
  ASRT: A < B ...; ...
        ^   ^
```

Note that this error is different from the above two. It is a general error detecting message reporting the unequality of data type in an asserion in terms of an relational operator. The ATLAS compiler supports only the basic binary operations but not the binary-decimal automatic conversion.

"**ERROR** CKR11: (UNUSED)

"**ERROR** CKR12: A DIGITAL OPERATOR HAS LHS OR RHS OF TYPE NOT DIGITAL;

AT LINE: stmt #"

```
EX.
  DCL A: INT;
      ^
  DCL B: DIG;
  ...
  ASRT: ABC = A "& B ...; ...
                ^
```

where the operator "& is one of the digital operators meaning "LOGICAL AND".

"**ERROR** CKR13: A 'ROTATE' OR 'SHUFT' OPERATOR HAS LHS NOT OF TYPE "DIGITAL"; AT LINE: stmt #"

```
EX.
  DCL A: INT;
  ... ^
  ASRT: ABC= A "* 2 ... ; ...
             ^
```

where "* is the digital operator "ROTATE" .

"**ERROR** CKR14: A 'ROTATE' OR 'SHIFT' OPERATOR HAS RHS NOT OF TYPE "INTEGER'; AT LINE: stmt #"

```
EX.
  DCL A: DIG;
  DCL B: DIG;
  ... ^
  ASRT: ABC = A "> B ...; ...
                   ^
```

"**ERROR** CKR15: AN ARITHMETIC OPERATOR HAS LHS OF TYPE "DIGITAL" OR "BOOLEAN"; AT LINE: stmt #"

```
EX.
  DCL A: BOOL; (or DCL A: DIG;)
  ... ^                ^
  ASRT: ABC = A + 120...; ...
              ^
```

"**ERROR** CKR16: AN ARITHMETIC OPERATOR HAS RHS OF TYPE "DIGITAL" OR "BOOLEAN"; AT LINE: stmt #"

EX.
Similar as above but RHS intead of LHS.

"**ERROR** CYC1: (CIRCULAR DEF): THE FOLLOWING GROP OF ITERMS IN

{NOPAL SPECIFICATION | TEST} 'name' ARE CIRCULARLY DEFINED:

... a list of item names ... "

"**ERROR** CYC2: CYCLE MENTIONED ABOVE COULD NOT BE ELIMINATED."

"**WARNING** CYC3: CYCLE MENTIONED ABOVE HAS BEEN ELIMINATED

BY CHANGING ENTRY ('entry #1','entry #2') FROM "1"

[W(IP,IS)] TO 0 IN ADJACENCY MATRIX."

"**WARNING** ESH1: CYCLE CONTAINS THE FOLLOWING ELEMENTS:

... a list of elements ... "

"**WARNING** ESH2: A RANGE CONFLICT IN NODE 'node name'

BETWEEN THE ALREADY ASSIGNED RANGE:'range #' AND THE NEWLY IMPLIED

RANGE:'new range #'"

"**ERROR** ESH3: NO RANGE DETERMINED FOR LOOP VARIABLES AT LEVEL-'level #' AT CYCLE-

... a list of nodes ..."

"**ERROR** ESH4: NO CANDIDATE SUBSCRIPT IN CYCLE-

... a list of nodes ..."

"**ERROR** ESH5: A CYCLE IS DETECTED-

... a list of nodes ..."

"** ERROR ** ESQ1: TEST 'name' AND DIAGNOSIS 'name' HAVE CONFLICTING

SUBSCRIPTS."

"**ERROR** ESQ2: (AMBIGUITY): IN STATEMENT NUMBER 'stmt #' GLOBAL VARIABLE' var name'

HAS BEEN USED MORE THAN ONCE WITH DIFFERENT DIMENSIONS."

"**ERROR** ESQ3: (INCOMPLETENESS): GLOBAL VARIABLE 'var name' IS USED AS SOURCE IN TEST 'test name'

BUT ITS TARGET DEFINITION NEVER GIVEN ELSEWHERE."

"**WARNING** ESQ4: (POSSIBLE AMBIGUITY): GLOBAL VAR 'var name' DEFINED AS TARGET MORE THAN ONCE IN: {TEST | DIAGNOSES } 'stmt#'

THEY MUST BE UNDER MUTUALLY EXCLUSIVE CONDITION."

"**ERROR** ESQ5: (AMBIGUITY): VARIABLE 'var name' HAS MORE THAN ONE

DATA DECLARATION."

"**WARNING** ESQ6: (POSSIBLE INCOMPLETENESS): GLOBAL VAR 'var name' DEFINED AS TARGET IN {TEST | DIAGNOSIS} 'name'; BUT NEVER USED."

"**ERROR** ESQ7: GLOBAL VARIABLE 'var name' IS AN ARRAY BUT ONE

DIMENSION IS UNDEFINED."

"**ERROR** ESQ8: 'name' MUST USE SUBSCRIPT 'sub name' IN DIMENSION 'dimension #'."

"**WARNING** ESQ9: (POSSIBLE INCOMPLETENESS): TEST MODULE 'name' DOES

NOT HAVE ANY DIAGNOSIS."

"**ERROR** ESQ10: (AMBIGUITY): TWO LOGICAL OPERATORS CONNECT TEST

MODULE 'name' WITH DIAGNOSIS 'name'.

"**ERROR** ESQ11: MORE THAN ONE TEST SELECT THE SAME DIAGNOSIS WITH / OR * ."

"**ERROR** ESQ12: (INCONSISTENCY): TWO OR MORE TESTS:

... list of names ... "

COME AFTER THE DIAGNOSIS 'name' VIA LOGICAL OPERATOR AFTER OR AFTER-NOT."

"**WARNING** ESQ13: (POSSIBLE INCONSISTENCY): BOTH INTERACTIVENESS AND COMPONENT PROTECTION RELATIONSHIPS EXIST BETWEEN DIAGNOSIS 'name' AND TEST MODULE 'name'; ONLY THE INTERACTIVE NESS RELATIONSHIP IS RETAINED."

/******"**ERROR** ESQ12: INTERACTIVENESS RELATIONSHIP BETWEEN DIAGNOSIS

'name' AND TEST 'name' CANNOT BE MAINTAINED." */

/* "**ERROR** ESQ13: ORDER VECTOR FOR NOPAL SPECIFICATION 'name' COULD

NOT BE GENERATED." */

"**ERROR** FLM1: FAIL STACK UNDERFLOW. COMPILATION DISCONTINUED."

The failure stack is filled. Process is forced to be stopped.

"**SYSTEM IMPLEMENTATION ERROR** FLM1: FAIL STACK UNDERFLOW."

"**WARNING** FLB1: THE FUNCTION 'fun name' IS NOT IN FUNCTION LIBRARY."

"**WARNING** FLB2: 'file name' CONTAINING THE MISSING FUNCTIONS SHOULD BE INCLUDED AT ATLAS COMPILE TIME."

"**WARNING** FLB3: 'file name' DOES NOT EXIST SHOULD BE INCLUDED AT ATLAS COMPILE TIME."

"**WARNING** FLB4: THE FUNCTION 'fun name' HAS BEEN MULTIPLY DEFINED."

"**ERROR** FTB1: -- ILLEGAL FUNCTION DEFINITION. FUN: the ATE function text."

This error message tells you that the ATE function you have specified in your NOPAL program is not matched by the ATE function liberary. You better check your function with the definitions in the liberary.

"**WARNING** GMG1: IN STMT NO. 'stmt #' "#" REQUIRED IMMEDIATELY AFTER PARAMETER IN MESSAGE."

"**ERROR** GWV1: NUMBER OF ARGUMENTS INVALID IN BUILT IN FUN 'fun name' AFTER 'atlas stmt #'."

This message tells that the user has specified a built-in function with unmatched # of arguments with the one that supported by the

NOPAL system.

"**ERROR** GWV2: OP IN FUNC 'fun name' IN PARM 'par naem(s)'

IN TEST MODULE NO. 'test name' SHOULD BE '='. ASSUMED."

The system corrects the incorrect use of an operator with indicated

function in the indicated TEST. An '=' operator is assumed.

"**ERROR** GWV3: ILLEGAL USE OF SUBSCRIPT EXPRS FOR VIRTUAL SUBSCRIPT:

'name'."

This message tells the illegal use of subscript in

NOPAL system. The correct use of subscript is of the form : I or I-1.

"**WARNING** GWV4: UNITS IN FUNCTION 'fun name' PARAMETER DEFINITION

AND USE DO NOT MATCH."

This tells the unmatched use of UNIT(s) in the indicated function

between use and delaration of the function.

"**WARNING** GWV5: UNITS IN FUNCTION 'fun name' PARAMETER USE NOT

RECOGNIZED."

This tells the missing of the UNIT parameter in a use of

certain function (or unrecognized by NOPAL system).

"**SYSTEM IMPLEMENTAION ERROR** GWV6: REQUIRE ARG STACK OVERFLOW!"

"**WARNING** INI1: UUT POINT 'name' NOT CONNECTED TO ANY ATE PIN."

This indicates the incomplete specification of connection points.

(The NOPAL system will include the corresponding specification

automatically.)

"**SYSTEM IMPLEMENTATION ERROR** ISP1: VARID NOT FOUND.'var name'."

"**ERROR** ISQ1: RETURNED FROM INTSEQ BECAUSE SUBSCRIPTED VARIABLES ARE USED INCONSISTENTLY IN TEST 'name'."

This indicates the incorrect use of subscripts within the named test.

"**ERROR** ISQ2: RETURNED FROM INTSEQ BECAUSE A CYCLE EXISTS IN TEST 'name'."

"**ERROR** ISQ3: IN TEST 'name' VARIABLE 'name' OCCURS IN MORE THAN

ONE DATA DCL."

"**WARNING** ISQ4: IN TEST 'name' PTR VARIABLE 'name' IS GIVEN A VALUE BUT CORRESPONDING RECORD IS NOT USED."

"**ERROR** ISQ5: INVALID SUBSCRIPT VARIABLE 'name' IN STATEMENT 'stmt #'."

"**ERROR** ISQ6: IN TEST 'name' VARIABLE 'var naem' USES DIFFERENT

NUMBER OF SUBSCRIPTS."

"**ERROR** ISQ7: IN TEST 'name' VARIABLE 'var name' OCCURS BOTH AS

SIMPLE AND ARRAY VARIABLE."

"**WARNING** ISQ8: (POSSIBLE AMBIGUITY): VARIABLE 'name' OF

TEST 'name', IS DEFINED MORE THAN ONCE IN 'list of names'.

THEY MUST BE EXCLUSIVE."

"**ERROR** ISQ9: (AMBIGUITY): IN ASSERTION 'name'

OF TEST 'name', THERE ARE TWO OR MORE TARGET VARIABLES: 'list of names'."

"**ERROR** ISQ10: (AMBIGUITY): EXPRESSION 'exp'

PRECEDING THE '=' IN ASSERTION 'name' OF TEST 'name' DOES NOT

MATCH THE TARGET VARIABLE 'list of names'."

"**WARNING** ISQ11: (INCONSISTENCY): IN ASSERTION 'name' OF

TEST 'name', A VARIABLE IS DECLARED AS TARGET;

BUT THE RELATION OPERATOR IS NOT AN EQUAL SIGN.

REPLACED BY AN EQUAL SIGN."

"**WARNING** LEX1: INVALID BIT STRING. CHARACTER STRING ASSUMED. IN STATEMENT 'stmt#', NEAR TEXT 'the nearest context saved'."

This tells the illegal declaration of a bit string.

"**WARNING** LEX2: IN STATEMENT 'stmt#', NEAR TEXT 'the nearest context saved'; ';' OF LAST STATEMENT MISSING. INSERTED."

EX.

```
ASRT: A=.....   with no semicolon(;)
                ^
```

The missing ";" was inserted.

"**WARNING** LEX3: IN STATEMENT 'stmt#', NEAR TEXT 'the nearest context saved' A NUMBER IS TOO LONG. TRUNCATED."

EX.

```
... A=11111111111024567899 ...
                         ^
```

The above number in the specification has length greater than 16.

"**WARNING** LEX4: IN STATEMENT 'stmt#', NEAR TEXT 'the nearest context saved'; EXPONENT DIGITS AFTER 'E' MISSING IN A NUMBER. 00 ASSUMED."

EX.

```
A = 1 ** 2E;
           ^
```

This is supposed to be "A = 1 ** 2E00;" or "A = 1 ** 2E01;" or something else.
The system assumes "00" follows the "E" in this case.

"**WARNING** LEX(LABEL)1: NAME/INTEGER WAS TOO LONG. TRUNCATED. IN STATEMENT
'stmt ',
NEAR TEXT 'the nearest text saved'"

"**WARNING** LEX(LABEL)2: ILLEGAL OPERATOR RESPONSE KEYWORD. YES/NO ASSUMED."

"**WARNING** LEX(STSPEC)1: END-LABEL DIDN'T MATCH SPEC-NAME. SPEC-NAME
ASSUMED."

"**WARNING** LEX(STSTIM)1: ID DECLARED NOT FOUND. ITS DECLARATION IGNORED."

"**WARNING** LEX(STSTIM)2: RELATIONAL OP WAS NOT '=' IN A TRIPLET. '='
ASSUMED."

"**WARNING** LEX(UUTCMPF)1: MIN-LIMIT GREATER THAN MAX-LIMIT.
MIN-LIMIT SET TO -1E35."

"**SYSTEM IMPLEMENTATION ERROR** MNT1: SAPIN INPUT RECORD SIZE EXCEEDS 80."

"**ERROR** RDS1: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #'
--- REDUCTION FUNCTION IS NOT USED IN AN ASSERTION."

"**ERROR** RDS2: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #'
REDUCTION STATEMENT IS IMPROPERLY USED."

"**ERROR** RDS3: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #'
TOO MANY OR TOO FEW ARGUMENTS IN THE REDUCTION FUNCTION."

"**ERROR** RDS4: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #'
THE SECOND ARGUMENT 'name' OF THE RHS OF REDUCTION
FUNCTION IS NOT DECLARED AS A FREE SUBSCRIPT."

"**ERROR** RDS5: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #'
THE FIRST ARGUMENT 'name' OF THE REDUCTION FUNCTION
IS NOT A SUBSCRIPTED VARIABLE."

"**ERROR** RDS6: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #',
THE 'number'-TH SUBSCRIPT 'name' OF VARIABLE
'var name' IS NOT A SIMPLE VARIABLE OR CONSTANT."

"**ERROR** RDS7: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #'
MORE THAN ONE DIMENSION MAY BE REDUCED IN A REDUCTION FUNCTION."

"**ERROR** RDS8: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #' THE SUBSCRIPT
BEING REDUCED 'name' IS NOT FOUND IN THE FIRST ARGUMENT 'name'."

"**ERROR** RDS9: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #' THE LHS 'name'
OF A REDUCTION FUNCTION IS NOT A SUBSCRIPTED VARIABLE OR SCALAR."

"**ERROR** RDS10: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #',
THE 'number'-TH SUBSCRIPT 'name' OF LHS OF
A REDUCTION FUNCTION IS NOT A SIMPLE VARIABLE OR CONSTANT."

"**ERROR** RDS11: (INCONSISTENT):IN STATEMENT NUMBER 'stmt #'
SUBSCRIPT 'name' IN LEFT-HAND SIDE OF A REDUCTION
DEOES NOT APPEAR IN RIGHT-HAND SIDE."

"**ERROR** RDS12: (INCONSISTENT):IN STATEMENT NUMBER 'stmt #'
THE DIMENSIONALITY OF LHS IS NOT ONE LESS THAN THAT OF RHS."

"**ERROR** RDS13: (IMPLEMENTATION): IN STATEMENT 'stmt #' THIS REDUCTION
STATEMENT CANNOT BE FOUND IN DICTIONARY TABLE 'DICT'."

"**SYSTEM IMPLEMENTATION ERROR** RTV1: MAX NO. OF RETRIEVALS EXCEEDED."

"**SYSTEM IMPLEMENTATION ERROR** XRT1: MAX. OF REF. ENTRIES PER ROW EXCEEDED
IN XREF2."

"**ERROR** SAP1: MISSING/INVALID ARGUMENT OF FUNCTION CALL OR SUBSCRIPT OF A
SUBSCRIPTED VARIABLE"

"**ERROR** SAP2: MISSING COLON ':'"

"**ERROR** SAP3: MISSING RELATIONAL OPERATOR IN AN ASSERTION"

"**ERROR** SAP4: MISSING/INVALID ARITHMETIC EXPRESSION"

"**ERROR** SAP5: MISSING RIGHT PARENTHESIS ')' "

"**ERROR** SAP6: MISSING RELATIONAL OPERATOR IN A BOOLEAN EXPRESSION"

"**ERROR** SAP7: MISSING/INVALID BOOLEAN EXPRESSION"

"**ERROR** SAP8: MISSING LEFT PARENTHESIS '(' AFTER NEGATION SIGN"

"**ERROR** SAP9: MISSING/INVALID CONNECTOR ID IN CONNDIMEX"

"**ERROR** SAP10: MISSING RIGHT TRIANGULAR BRACKET '>' IN CONNDIMEX"

"**ERROR** SAP11: MISSING/INVALID AFFECTED COMPONENT"

"**ERROR** SAP12: '&' AND '|' OPERATORS MIXED"

"**ERROR** SAP13: MISSING RELATIONAL OPERATOR IN A CONJUNCTION"

"**ERROR** SAP14: MISSING VARIABLE ID IN DECLARATION"

"**ERROR** SAP15: MISSING 'MESSAGE' AFTER 'OPERATOR' IN KEYWORD DIAGNOSIS DEFINITION"

"**ERROR** SAP16: MISSING COLON ':' AFTER 'OPERATOR MESSAGE' IN KEYWORD DIAGNOSIS DEFINITION"

"**ERROR** SAP17: MISSING DIAGNOSIS LABEL"

"**ERROR** SAP18: MISSING DATA TYPE"

"**ERROR** SAP19: MISSING/INVALID FUNCTION DIMENSION EXPRESSION"

"**ERROR** SAP20: MISSING RIGHT PARENTHESIS IN FUNCTION DIMENSION EXPRESSION"

"**ERROR** SAP21: MISSING/INVALID FUNCTION TYPE"

"**ERROR** SAP22: MISSING/INVALID ASSERTION"

"**ERROR** SAP23: MISSING THEN IN AN IF-CLAUSE"

"**ERROR** SAP24: MISSING EQUAL SIGN '=' AFTER A KEYWORD"

"**ERROR** SAP25: MISSING/INVALID LOGICAL OPERATOR"

"**ERROR** SAP26: MISSING DIAGNOSIS LABEL AFTER LOGICAL OPERATOR"

"**ERROR** SAP27: MISSING MESSAGE LABEL"

"**ERROR** SAP28: MISSING EQUAL SIGN '=' AFTER 'ALIAS'"

"**ERROR** SAP29: MISSING MESSAGE SYNONYM AFTER '='"

"**ERROR** SAP30: MISSING COMMA ',' AFTER MESSAGE SYNONYM"

"**ERROR** SAP31: MISSING/INVALID MESSAGE ARGUMENT IN OTHER PARAMETERS OF A DIAGNOSIS"

"**ERROR** SAP32: MISSING/INVALID OPERATOR RESPONSE VARIABLE ID"

"**ERROR** SAP33: MISSING/INVALID BACK-REFERENCE STIM/MEAS LABEL"

"**ERROR** SAP34: INVALID SPECIFICATION STATEMENT"

"**ERROR** SAP34: MISSING/INVALID KEYWORD"

"**ERROR** SAP36: MISSING/INVALID TEXT AFTER '='"

"**ERROR** SAP37: MISSING IDENTIFIER"

"**ERROR** SAP38: MISSING PARENT LABEL AFTER '('"

"**ERROR** SAP39: MISSING/INVALID MESSAGE TEXT"

"**ERROR** SCD1: THERE IS A CYCLE IN THE GRAPH. "

"**ERROR** SCD2: NODES ARE NOT SCHEDULABLE."

"**WARNING** SCD3: SUBSCRIPT 'name' CONTAINING IO MUST BE SPLIT."

"**SYS ERROR** SCD4: ILLEGAL CASE."

"**SYS ERROR** SCD5: TOPOLOS: FOR NODE 'a list of nodes' PARENT COUNT < 0"

"**SYS ERROR** SCD6: DELETE: ELEM NOT FOUND.'element #'"

"**SYS ERROR** SCD7: ENDITER: ITERATION VARIABLE NOT FOUND.
'# of iteration'."

"**WARNING** SAR1: LOGICAL CONNECTIVE (logical connective) APPEARED IN ARITHEXP. AT LINE: stmt #. THE LOGICAL CONNECTIVE WERE IGNORED."

EX.

```
A= B & C;
     ^
```

This is a sytactical error meaning that the logical connective(s) appeared in an arithmetic expression.

"**WARNING** SAR2: ILLEGAL OPERATOR "^" LEADING ARITHEXP AT LINE: stmt #. THE "^" WAS IGNORED."

EX.

```
A = ^B;...
```

Similar as above, the logical symbol "^" is discarded.

"**ERROR** SAR3: ILLEGAL HEX CONSTANT; ' IS MISSING AT LINE: stmt #."

EX.

```
A = H'120AB;
           ^
```

meaning that the digital constant is not properly surrounded by single quote "'".

"**ERROR** SAR4: ILLEGAL HEX CONSTANT; CONTENT NOT ALLOWED;
AT LINE:stmt #."

EX.
```
A = H'01N';
        ^
```

This is an obvious mistake. The content of a hexdecimal can only be in the

character set from "0" to "9" and "A" to "F".

"**ERROR** SAR5: ILLEGAL OCT CONSTANT; ' IS MISSING AT LINE:stmt #."

EX.
```
A = O'012;
         ^
```

"**ERROR** SAR6: ILLEGAL OCT CONSTANT; CONTENT NOT ALLOWED;

AT LINE:stmt #."

EX.
```
A = O'0091';
        ^
```

Remenber that the content of an octal constant can only be from "0" to "7".

"**ERROR** SAR7: ILLEGAL BINARY CONSTANT; ' IS MISSING AT LINE:stmt #."

EX.
```
A = B'010;
         ^
```

"**ERROR** SAR8: ILLEGAL BINARY CONSTANT; CONTENT NOT ALLOWED; AT LINE:stmt # ."

EX.
```
A = B'120';
       ^
```

This is an obvious mistake. The contents allowed in a binary constant is either "0" or "1".

"**SYSTEM IMPLEMENTATION ERROR** STL1: TOO MANY SUBSCRIPTS."

"**SYSTEM IMPLEMENTATION ERROR** STM1: NO. OF SUBSCRIPTS EXCEEDED."

"**SYSTEM IMPLEMENTATION ERROR** STM2: CONJ. TRIPLETS EXCEEDS 15 ('number')."

"**SYSTEM IMPLEMENTATION ERROR** STM3: EXPR STACK OVERFLOW."

"**SYSTEM IMPLEMENTATION ERROR** STS1: Too many subscripts."

"**WARNING** SBL1: (INCOMPLETE): IN TEST 'name' DIMENSION-'number' OF VARIABLE 'var name' IS NOT DECLARED, BOUND UNDECIDED."

"**ERROR** SBL2: (INCONSISTENT): IN TEST 'name' THE NUMBER OF DIMENSIONS FOR VARIABLE 'var name' IS INCONSISTENT DEFINED."

"**ERROR** SBL3: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #' SUBSCRIPT DECLARATION IS NOT A SIMPLE ASSERTION."

"**ERROR** SBL4: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #' VARIABLE 'var name' IS MULTIPLY DECLARED AS FREE SUBSCRIPT."

"**ERROR** SBL5: (INCOMPLETE): IN STATEMENT NUMBER 'stmt #' THE PARENT LIST OF SUBSCRIPT DECLARATION IS MISSING OR NOT ENCLOSED IN QUOTES."

"**ERROR** SBL6: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #' EXCESSIVE BLANK APPEARS IN THE PARENT LIST."

"**ERROR** SBL7: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #' IMPROPER USE OF COMMA IN THE PARENT LIST."

"**ERROR** SBL8: (INCOMPLETE): IN STATEMENT NUMBER 'stmt #' NULL PARENT LIST IS USED."

"**ERROR** SBL9(AMBIGUOUS): IN STATEMENT NUMBER 'stmt #' THE DIMENSION NUMBER IS A NOT A POSITIVE INTEGER."

"**ERROR** SBL10: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #' IMPROPER USE OF

COLON."

"**ERROR** SBL11: (INCOMPLETE): IN STATEMENT NUMBER 'stmt #' SUBSCRIPT DECLARATION IS NOT CORRECT, CHECK USE."

ERROR SBL12: (INCONSISTENT): IN STATEMENT NUMBER 'stmt #' THE UPPER BOUND IS NOT A POSITIVE INTEGER OR *."

"**ERROR** SBL13: (INCONSISTENT): IN STATEMENT NUMBER 'stmt #' THE UPPER STATEMENT NUMBER 'stmt #' VARIABLE 'var name' IS NOT USED AS A SUBSCRIPTED VARIABLE IN THE TEST."

"**ERROR** SBL14: (INCONSISTENT): IN STATEMENT NUMBER 'stmt #' DIMENSION 'numbar' OF VARIALBE 'var name' IS DOUBLY DEFINED."

"**WARNING** SBL15: (AMBIGUOUS): IN STATEMENT NUMBER 'stmt #' IN TEST 'name' PARENT NAME 'var name' HAS NEVER APPEARED IN THE TEST."

"**WARNING** SUS1: (POSSIBLE INCONSISTENCY): IN STATEMENT NUMBER 'stmt $' THE FREE SUBSCRIPT 'name' IN SOURCE VARIABLE (OR CONNECTOR) DOES NOT APPEAR AS FREE SUBSCRIPT IN THE TARGET VARIABLE."

"**ERROR** SUS2: (INCONSISTENT): IN STATEMENT NUMBER 'stmt #' THE FREE SUBSCRIPT 'name' IN SOURCE VARIABLE (OR CONNECTOR) DOES NOT APPEAR AS FREE SUBSCRIPT IN THE TARGET VARIABLE."

"**ERROR** SUS3: (INCONSISTENT): IN STATEMENT NUMBER 'stmt #' TARGET VARIABLES DO NOT USE THE SAME FREE SUBSCRIPTS."

"**ERROR** SUS4: (INCONSISTENT): IN STATEMENT NUMBER 'stmt #' SUBSCRIPT IN VARIABLE OR CONNECTOR 'name' DOES NOT FOLLOW THE CORRECT SYNTAX."

"**ERROR** SUS5: (INCONSISTENT): IN STATEMENT NUMBER 'stmt #' SUBSCRIPT IN CONNECTOR 'name' DOES NOT FOLLOW THE CORRECT SYNTAX."

"**WARNING** SUS6: (POSSIBLE INCONSISTENT): IN STATEMENT NUMBER 'stmt #', 'number'-TH SUBSCRIPT OF VARIABLE 'var name' IS A SUBSCRIPTED VARIABLE OR A NON-FREE SUBSCRIPT; RANGE TEST IS NOT PERFORMED."

"**ERROR** SUS7: (INCOMPLETE): IN STATEMENT NUMBER 'stmt #' VARIABLE 'var name' IS NOT FOUND IN THE VARIABLE TABLE OR DIMENSION-'numbar', IS NOT DEFINED."

"**ERROR** SUS8: (INCONSISTENT): IN STATEMENT NUMBER 'stmt #' THE BOUND OF DIMENSION-'number' OF VARIABLE 'name' WAS DECLARED TO BE 'number1' BUT 'number2' IS USED HERE.

"**SYSTEM IMPLEMENTION ERROR** SUS9: FROM SUBUSAG. FAILED TO FIND WAVEFORM OF 'stmt #' OR VARIABLE 'var name' IN ADJACENCY MATRIX."

"**SYSTEM IMPLEMENTATION ERROR** STM1: NO. OF SUBSCRIPTS EXCEEDED."

"**SYSTEM IMPLEMENTATION ERROR** STM2: CONJ. TRIPLETS EXCEEDS 15 (STRIPT)."

"**SYSTEM IMPLEMENTATION ERROR** STM3: EXPR STACK OVERFLOW."

"**ERROR** XRF1: 'name' OCCURS BOTH AS A PARM OF MODFUN AND IN GLOBAL DCL."

"**ERROR** XRF2: 'name' HAS CONFLICTING DIMENSIONS: '# of dimension 1' IN STMT 'stmt#1' AND '# of demension 2' IN STMT 'stmt #2'.'"

"**WARNING** XRF3: 'name' IS USED AS A LOCAL VARIABLE IN TEST 'test name' AND ALSO IN DIAGNOSES 'diag. name'"

"**WARNING** XRF4: 'stmt#1', 'name', MULTIPLY DEFINED. THE STMNT DELETED. FIRST DEFN. IN STMNT stmt#2."

"**WARNING** XRF5: IN STMNT 'stmt#', 'name' HAD MULTIPLE CONJUNCTION. DELETED."

"**ERROR** XRF6: IN STMNT 'stmt#', 'name' CONJUNCTION BACK REF. COULD NOT BE RESOLVED."

"**ERROR** XRF7: IN STMNT 'stmt#', 'name' NEVER DEFINED. INVALID REFERENCE."

"**ERROR** XRF8: IN STMNT 'stmt#', 'name' CONJUNCTION BACK REFERENCES FORM A LOOP."

"**WARNING** XRF9: IN STMNT 'stmt#', 'name' DIAGNOSIS NEVER REFERENCED."

NOTE:

The error messages with the form "**SYSTEM IMPLEMENTATION ERROR**..." are run-time prompts of NOPAL system indicating the abnormal conditions in processing. Similarly the messages with the form "**SYS ERROR**..." have the same function but in the error file [ERRFIL.DAT]. Please inform appropriate personel about those matter and ask for help. Those are most likely to be system bugs.