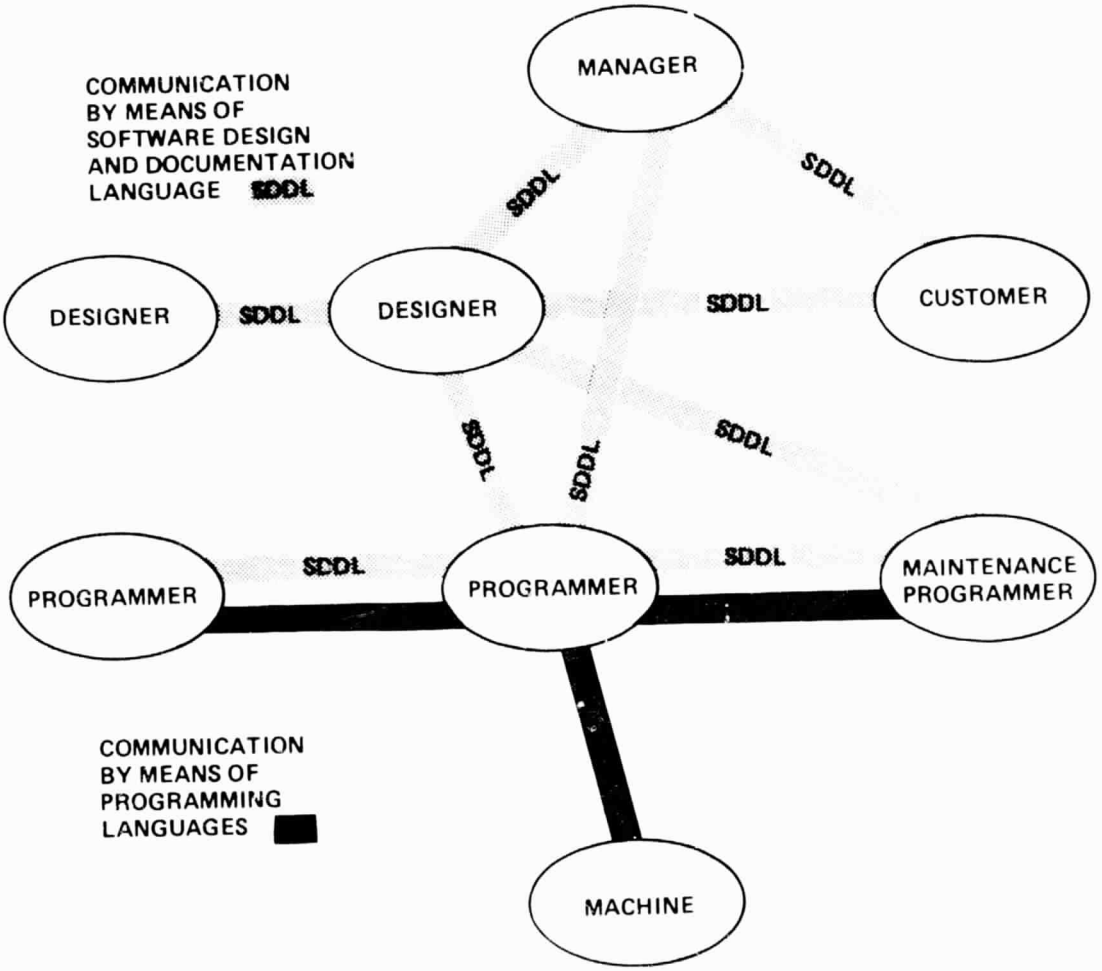
JPL PUBLICATION 77-24

Software Design and Documentation Language

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91103



SOFTWARE DEVELOPMENT TEAM COMMUNICATIONS

JPL PUBLICATION 77-24

Software Design and Documentation Language

Henry Kleine

July 1, 1977

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91103

PREFACE

The work described in this report was performed by the Information Systems Division of the Jet Propulsion Laboratory.

ACKNOWLEDGMENT

Many aspects of the methodology for using SDDL, and enhancements to the language and the processor, evolved from its application to the design of two programs: the Vehicle Economy and Emissions Program (VEEP), and the Solar Array Manufacturing Industry Simulation (SAMIS). The current capabilities, present methodology, successful application, and future prospects of SDDL are, in large measure, due to the many contributions of the members of these design teams. For their many excellent suggestions, critical reviews of this document, critique of new processor capabilities, conscientious application of SDDL to the design tasks, and hours of philosophical discussion of the goals of a software design tool, I wish to express my thanks to Richard V. Morris, Donald A. Heimburger, Marcia A. Metcalfe, Bruce L. Kleine, Robert G. Chamberlain, Steve M. Jacobs, Robert L. Norton, and Gerhard J. Klose.

ABSTRACT

The objective of the Software Design and Documentation Language (SDDL) is to provide an effective communications medium to support the design and documentation of complex software applications. This objective is met by providing (1) a processor which can convert design specifications into an intelligible, informative machine-reproducible document, (2) a design and documentation language with forms and syntax that are simple, unrestrictive, and communicative, and (3) methodology for effective use of the language and processor.

The SDDL processor is written in the SIMSCRIPT II programming language and has been implemented on the UNIVAC 1108 and the IBM 360/370 machines.

CONTENTS

Figure

Tables

SECTION I

INTRODUCTION

The frontispiece is a conceptual view of the software development process. It identifies members of the software development team and shows the many communication links over which information must flow. The team members and the information flow shown in the diagram are a part of every software development project regardless of the number of individuals actually involved. Even when the entire task is done by a single person, it is still essential to have precise, accurate, orderly communication among the various roles the individual performs. With orderly communication, decisions made last month can be acted upon correctly this month, and valid information will be available later when maintenance responsibilities may have to be assumed by others.

The diagram also suggests that a computer programming language is a satisfactory communications medium for only a few links; primarily between programmer and machine, and secondarily among programmers. All other higher-level team communication requires less restrictive, more human-oriented media to be effective.

Historically, software development has suffered because of the lack of an effective communications medium for these high-level links. One may generalize that everyone has experienced some painful results of imprecise and/or incomplete communication in every aspect of life. Programmers suffer immediately when imprecise, incorrect, or incomplete directions are executed by the computer exactly as stated. Managers and customers are affected more seriously because bad communications at the design stage may compound the error by allowing the programming effort, with all its problems, to proceed toward an elusive or erroneous goal.

As long as the communication between members of the software development team remains fuzzy, the misunderstanding will continue and software development costs will be higher than they need be. Software maintenance gets into the act later, when maintenance programmers must deal with poorly written, out-of-date documentation, which, by Murphy's Law, is certain to be inconsistent where it matters.

Effective communication is not sufficient to insure efficient software development, but it is certainly necessary. Therefore, the Software Design and Documentation Language (SDDL) has been developed to satisfy this necessity.

A. SDDL OBJECTIVE

The objective of SDDL is to satisfy the communications requirements of the software design and documentation process. This objective is met by providing

(1) A processor which can convert design specifications into an intelligible, informative, machine-reproducible Software Design Document (SDD).

(2) A program design and documentation language with forms and syntax that are simple, unrestrictive, and communicative.

(3) A methodology for effective use of the language and the processor.

B. SDDL PROCESSOR

The purpose of the SDDL processor is to translate the designer's creative thinking into an effective communications document. The processor must perform as many automatic functions as possible, thereby freeing the designer's energy for the creative design effort.

Some of the automatic functions which the processor, in its current state of development, performs are listed below.

1. Document Formatting

(1) Indentation by structure logic.

(2) Flow lines for accentuating structure escapes.

(3) Flow lines for accentuating module invocation.

(4) Line numbering and/or card sequencing for input deck editing.

(5) Logic error detection.

(6) Special handling for title pages and text segments.

(7) Input and output line continuation.

(8) Line splitting (i.e., printing part of the line so that the last character lines up at the right-hand margin).

2. Software Design Summary Information

(1) Module invocation hierarchy.

(2) Module cross reference (where each module is invoked).

(3) Cross reference tables for selected words or phrases appearing in the document. Selection is controlled by the user.

(4) Table of contents showing all titles and modules, and the location of the tables described above.

(5) Page reference numbers on module invocation statements.

3. Processor Control Capabilities

(1) Page width.

(2) Structure indentation amount.

(3) Page ejection.

(4) Input line numbering sequence.

(5) Keyword specification.

(6) Selection of words for inclusion in the cross reference tables.

(7) Number of right-hand columns for card sequence numbers.

(8) Execution time options for suppressing selected processor features.

C. SDDL OVERVIEW

1. SDDL Syntax

The SDDL syntax consists of keywords, used to invoke design structures, and a collection of directives, which provide the user with control of processor actions such as indentation, page width, start of a new page, etc. Execution time options allow the user to selectively suppress design summary information.

Input to the SDDL processor consists of a sequence of SDDL statements. An SDDL statement begins and ends with a line (or record) of the input medium, unless continuation is explicitly indicated by placing an ampersand (&) in the last non-blank character of the line. Continued lines are concatenated into a single statement for processing. Any natural language text, including a blank line, is an acceptable SDDL statement. Keywords are recognized only in context, that is, only when they appear as the first word of the input statement.

The user is provided complete control of the choice of keywords by an SDDL directive which allows unlimited addition or deletion of keywords. User control of keyword selection is one of the most important features of SDDL because it allows the designer to command the capabilities of the processor in the way which is best suited to communicating the intent of the design.

A complete description of the SDDL semantics is given in Section II and summarized there in the SDDL Syntax Overview diagrams.

2. SDDL Structures

The basic forms of the language are the module and block structures, and the Module Invocation statement. A design is stated in terms of modules that represent problem abstractions which are complete and independent enough (relative to the level of the design) to be treated as separate problem entities. Modules are the highest-level structure. They may not be nested. Descriptive names are given to the modules, and their interrelationships are stated explicitly by the Module Invocation statements. A Module Invocation statement is the equivalent of the subroutine CALL statement in a programming language.

Blocks are the lower-level structures. They are used to build representations of abstractions which should (relative to the specific design) be a part of and appear with the higher-level abstraction represented by the module. Thus blocks must be nested within modules and may be nested within other blocks to any reasonable (i.e., understandable) depth. Examples of the use of blocks are the representations of Structured Programming concepts such as IF-THEN-ELSE and LOOP-REPEAT.

Both kinds of structures may have up to four parts:

(1) Initiator (required)

(2) Terminator (optional)

(3) Escape (optional)

(4) Substructure (optional)

Structure parts are specified by statements which begin with a keyword that has been defined as the part name. Table 1 displays the SDDL default keywords for both kinds of structures and their corresponding structure parts.

The actions taken by the processor in response to keyword statements are fully explained in Section II and summarized in Figure 1. These actions are quite simple but very effective for communicating design information. Indentation of statements within structures, and flow lines to highlight structure escapes and module invocations provide visual, two-dimensional information display which captures all of the advantages offered by flowcharts without their attendant disadvantages and constraints.

A simple illustration is presented following Figure 1.

Table 1-1. SDDL Control Structure Keywords

	INITIATOR	TERMINATOR	ESCAPE	SUBSTRUCTURE
MODULE	PROGRAM	ENDPROGRAM	EXIT PROGRAM	
	PROCEDURE	ENDPROCEDURE	EXITPROCEDURE	
BLOCK	IF	ENDIF		ELSE ELSEIF
	SELECT	ENDSELECT		CASE
	LOOP	ENDLOOP REPEAT	EXITLOOP CYCLE	
CALL	CALL DO	N/A	N/A	N/A
PROCESSOR CONTROL	#LINENUMBER #EJECT #INDENT #DEFINE #MARK #WIDTH #STRING #SEQUENCE #TERMINATE			
	#TITLE #TEXT	#END		

ACTION TAKEN	STATEMENT TYPE ENCOUNTERED: DEFINITION NUMBER →									
	2.1 MODULE INITIATOR STATEMENT	2.2 BLOCK INITIATOR STATEMENT	2.3 TERMINATOR STATEMENT	2.4 SUBSTRUCTURE STATEMENT	2.5 ESCAPE STATEMENT	2.6 MODULE INVOCATION STATEMENT	1.6 PASSIVE STATEMENT	3.5 TEXT DIRECTIVE	3.7 TITLE DIRECTIVE	4.6 CONTROL DIRECTIVE
Statement entered in table of contents	←							←		
All nested, open structures are closed with error messages	←		←	←				←		
New page started in the output file	←							←		
Indentation level decreased			←	←						
Statement written to output file	←	←	←	←	←	←	←			
Indentation level increased	←	←		←						
Left arrow (escape level indicator) added to the output file					←					
Right arrow (call indicator) added to the output file						←				
Subsequent input lines are diverted to a holding buffer							←	←		
The lines in the holding buffer are written to the output file (boxed in by "*")									←	
Subsequent input lines are diverted back for normal processing									←	
Control parameters of the SDDL processor are altered									←	

Figure 1-1. SDDL Processor Actions

In most of the following examples, the SDDL input statements are shown with the resulting output produced by the processor. In practice, the input source listing is rarely needed. Where the source statements are shown, as in the example below, it should be understood that the line numbering, including the colon, was added and is not part of the input statement.

Example:

As input:

```
 1:PROGRAM EXAMPLE TO DEMONSTRATE THE BASIC SDDL STRUCTURES
 2:(THE LINE ABOVE IS A MODULE INITIATOR STATEMENT WHICH ESTABLISHES
 3:"EXAMPLE" AS THE NAME OF THIS PROGRAM/MODULE)
 4:
 5:IF THIS CONDITION IS TRUE  (BLOCK INITIATOR "IF")
 6:ACT ON THIS STATEMENT  (PASSIVE STATEMENT)
 7:ELSE  (SUBSTRUCTURE STATEMENT FOR "IF")
 8:ACT ON THE FOLLOWING STATEMENTS  (ANOTHER PASSIVE STATEMENT)
 9:
10:LOOP FOR INDEX = 1 TO SOMETHING  (BLOCK INITIATOR "LOOP")
11:   (PASSIVE STATEMENTS CAN BE PLACED ANYWHERE)
12:CALL SUBROUTINE  (MODULE INVOCATION STATEMENT)
13:THE NAME OF THE MODULE INVOKED IN THE PREVIOUS STATEMENT
14:IS "SUBROUTINE"
15:IF THERE IS NOTHING LEFT TO DO  (NESTED BLOCK INITIATOR "IF")
16:EXITLOOP  (ESCAPE STATEMENT "LOOP")
17:ENDIF  (TERMINATOR STATEMENT NESTED "IF")
18:ENDLOOP  (TERMINATOR STATEMENT "LOOP")
19:
20:ENDIF  (TERMINATOR STATEMENT "IF")
21:ENDPROGRAM  (MODULE TERMINATOR STATEMENT "PROGRAM")
22:
23:PROCEDURE SUBROUTINE
24:
25:   NOTE:  A MODULE INITIATOR STATEMENT CAUSES THE START OF A NEW PAGE.
26:
27:SELECT CASE BASED ON SOME CRITERION  (BLOCK INITIATOR "SELECT")
28:
29:CASE 1: CHECK FOR SUBROUTINE ABORT (SUBSTRUCTURE STATEMENT FOR "SELECT")
30:IF THERE IS NO MORE DATA TO BE READ  (BLOCK INITIATOR "IF")
31:EXITPROCEDURE  (ESCAPE STATEMENT "PROCEDURE")
32:ENDIF
33:
34:CASE 2: CHECK FOR SUBROUTINE ERROR (SUBSTRUCTURE STATEMENT FOR "SELECT")
35:IF AN ERROR OCCURS  (BLOCK INITIATOR "IF")
36:PRINT AN ERROR MESSAGE  (PASSIVE STATEMENT)
37:ENDIF
38:
39:CASE 3: INVOKE ANOTHER SUBROUTINE  (SUBSTRUCTURE STATEMENT FOR "SELECT")
40:DO ANOTHER SUBROUTINE  (MODULE INVOCATION STATEMENT)
41:NOTE:  "DO" IS A SYNONYM FOR "CALL"  (PASSIVE STATEMENT)
42:
43:ENDSELECT  (TERMINATOR STATEMENT "SELECT")
44:ENDPROCEDURE (MODULE TERMINATOR STATEMENT "PROCEDURE")
```

As output:

```
                                TABLE OF CONTENTS                                PAGE    1
 PAGE   LINE ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
NUMBER NUMBER     MODULE NAME

   1      1     PROGRAM EXAMPLE TO DEMONSTRATE THE BASIC SDDL STRUCTURES
   2     23     PROCEDURE SUBROUTINE

   3            MODULE REFERENCE TREE

   4            MODULE - CROSS REFERENCE LISTING
```

```
LINE                                                                        PAGE    1
   1 PROGRAM EXAMPLE TO DEMONSTRATE THE BASIC SDDL STRUCTURES
   2    (THE LINE ABOVE IS A MODULE INITIATOR STATEMENT WHICH ESTABLISHES
   3    "EXAMPLE" AS THE NAME OF THIS PROGRAM/MODULE)
   4
   5    IF THIS CONDITION IS TRUE  (BLOCK INITIATOR "IF")
   6       ACT ON THIS STATEMENT  (PASSIVE STATEMENT)
   7    ELSE  (SUBSTRUCTURE STATEMENT FOR "IF")
   8       ACT ON THE FOLLOWING STATEMENTS  (ANOTHER PASSIVE STATEMENT)
   9
  10       LOOP FOR INDEX = 1 TO SOMETHING  (BLOCK INITIATOR "LOOP")
  11             (PASSIVE STATEMENTS CAN BE PLACED ANYWHERE)
  12          CALL SUBROUTINE  (MODULE INVOCATION STATEMENT)-------------->(  2)
  13          THE NAME OF THE MODULE INVOKED IN THE PREVIOUS STATEMENT
  14          IS "SUBROUTINE"
  15          IF THERE IS NOTHING LEFT TO DO  (NESTED BLOCK INITIATOR "IF")
  16       <-----EXITLOOP  (ESCAPE STATEMENT "LOOP")
  17          ENDIF  (TERMINATOR STATEMENT NESTED "IF")
  18       ENDLOOP  (TERMINATOR STATEMENT "LOOP")
  19
  20    ENDIF  (TERMINATOR STATEMENT "IF")
  21 ENDPROGRAM  (MODULE TERMINATOR STATEMENT "PROGRAM")
```

```
LINE                                                                PAGE     2
  23 PROCEDURE SUBROUTINE
  24
  25       NOTE:  A MODULE INITIATOR STATEMENT CAUSES THE START OF A NEW PAGE.
  26
  27    SELECT CASE BASED ON SOME CRITERION  (BLOCK INITIATOR "SELECT")
  28
  29    CASE 1: CHECK FOR SUBROUTINE ABORT (SUBSTRUCTURE STATEMENT FOR "SELECT")
  30       IF THERE IS NO MORE DATA TO BE READ  (BLOCK INITIATOR "IF")
  31 <--------EXITPROCEDURE  (ESCAPE STATEMENT "PROCEDURE")
  32       ENDIF
  33
  34    CASE 2: CHECK FOR SUBROUTINE ERROR (SUBSTRUCTURE STATEMENT FOR "SELECT")
  35       IF AN ERROR OCCURS  (BLOCK INITIATOR "IF")
  36          PRINT AN ERROR MESSAGE  (PASSIVE STATEMENT)
  37       ENDIF
  38
  39    CASE 3: INVOKE ANOTHER SUBROUTINE  (SUBSTRUCTURE STATEMENT FOR "SELECT")
  40       DO ANOTHER SUBROUTINE  (MODULE INVOCATION STATEMENT)---------->(   )
  41       NOTE:  "DO" IS A SYNONYM FOR "CALL"  (PASSIVE STATEMENT)
  42
  43    ENDSELECT  (TERMINATOR STATEMENT "SELECT")
  44 ENDPROCEDURE (MODULE TERMINATOR STATEMENT "PROCEDURE")
```

```
    ..........  MODULE REFERENCE TREE  ......                          PAGE     3
 LN  PAGE
  1     1  EXAMPLE
  2     2  .  SUBROUTINE
  3     .  .  .  ANOTHER
```

```
                                MODULE
                          CROSS REFERENCE LISTING                       PAGE     4
IDENTIFIER++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

ANOTHER
    PAGE   2  PROCEDURE SUBROUTINE
     LINES  40
EXAMPLE
    PAGE   1  PROGRAM EXAMPLE
     LINES  1, 3
SUBROUTINE
    PAGE   1  PROGRAM EXAMPLE
     LINES  12, 14
    PAGE   2  PROCEDURE SUBROUTINE
     LINES  23, 29, 34, 39, 40
```

D. SDDL METHODOLOGY

The following discussion of techniques and styles is intended as a guideline or list of suggestions for using the capabilities of the SDDL language and processor to fullest advantage in striving for the goal of an informative and communicative Software Design Document.

The reader is encouraged to examine these suggestions with a critical eye. Accept what is useful, adapt to your own requirements and taste, and invent new methods, but always keep in mind that the primary purpose of the Software Design Document is to communicate information to other people.

1. Uses of the Software Design Document

Throughout the development of the software design, the SDD always represents the definitive word on the current status of the ongoing, dynamic design development process. It is easily updated and readily accessible, in a familiar, informative, readable form, to all members of the development team. This makes the SDD an effective instrument for reconciling misunderstandings and disagreements in the evolutionary development of design specifications, engineering support concepts, and the software design itself. Using the SDD to analyze the design makes it possible to eliminate many errors which otherwise might not be detected until coding is attempted.

As a project management aid, the SDD is very useful for monitoring progress and for recording task responsibilities.

The SDD has been found to be very effective in its primary role as the specification for coding the design. To date, there is no experience with the use of the SDD for software maintenance, but since the SDD is easily revised, and revisions are automatically cross referenced, the outlook for this purpose is favorable.

2. Representation of Data Structures

A thorough knowledge of the content and organization of its input and output data is an essential prerequisite to understanding a program. For this reason, much attention was focused on developing data structure representations that effectively display data organization and content. SDDL techniques that facilitate achieving this goal include:

- Group the data into appropriate data description modules located in the beginning pages of the SDD.
- Provide descriptive names for variables.
- Use the period (.) (it lies low on the printed line and does not interfere with readability) to connect the words of a descriptive phrase to form a variable name.

- Use the underscore to connect the words of a descriptive phase to form a module name.
- Use the single or double quote mark to identify single word variable names for cross referencing.
- Include information about the data (e.g., units, mode, dimension, etc.) in the data structure module.
- Group all data which describe attributes of a design entity with the entity they describe, and provide an entity name which can be used as a qualifier with the attribute.
- If the program is to be implemented in a language that does not permit the use of descriptive variable names, include the name to be used in the program code in the data structure.
- Define ENTITY (or another suitable word) to be a block initiator keyword to provide automatic indentation. Use the #TERMINATE directive to terminate the block without printing a Termination statement.

  Example:

```
PROGRAM VEHICLE_COMPONENTS DATA STRUCTURE

  ENTITY     ENGINE:
       PCT.PEDAL                    [PCTPED]     PERCENT
       'RPM'                        [ENGRPM]     REV/MIN
       'TORQUE'                     [TORQUE]     FT*LB
        MIN.TORQUE                  [MINTOR]     FT*LB
        MAX.TORQUE                  [MAXTOR]     FT*LB
       'HORSEPOWER'     (VECTOR)   [HPOWER]     HP

ENDPROGRAM VEHICLE_COMPONENTS DATA STRUCTURE

PROGRAM DYNAMIC_SYSTEM_PARAMETERS DATA STRUCTURE
  .
  .
  .
ENDPROGRAM
```

3. Representations of Control/Procedural Structures

The constructs of Structured Programming, such as modules (e.g., PROGRAM - RETURN - ENDPROGRAM), iterations (e.g., LOOP - CYCLE/EXITLOOP - REPEAT), conditionals (e.g., IF - ELSE - ENDIF), and selections (e.g., SELECT - CASE - ENDSELECT) are used in a similar manner for software design. The difference is that for software design, the structures should convey human-oriented, natural language information to the level of precision and completeness necessary to communicate the design, but free of the syntax constraints and detailed information requirements imposed by programming languages.

Example: Module and block structures, high-level statements

```
1     PROGRAM MAIN ROUTINE
2        LOOP UNTIL THERE IS NO MORE DATA
3           READ THE DATA AND CHECK IT
4           IF THE DATA IS BAD OR INCOMPLETE
5        <-----CYCLE TO THE NEXT CASE
6           ELSE
7              CALL DATA_PROCESSING ROUTINE---------> (9)
8           ENDIF
9        REPEAT
10       TERMINATE THE PROGRAM
11    ENDPROGRAM
```

o If the design must specify a list of conditions where all must be tested and acted upon if true (in contrast to the SELECT-CASE-ENDSELECT construct, which finds and executes only the first true condition), a new structure is recommended in place of a sequence of IF-ENDIF structures. Use the #DEFINE directive to establish the following structure:

CHECK - block initiator
ENDCHECKLIST - block terminator
CONDITION - substructure

Example:

As input:

```
 1:#DEFINE BLOCK CHECK, ENDCHECKLIST,, CONDITION
 2:
 3:PROGRAM FOR VACATION PREPARATION
 4:
 5:CHECK AND ACT ON ALL TRUE CONDITIONS IN THE FOLLOWING LIST
 6:
 7:CONDITION: CAR NEEDS TO BE SERVICED
 8:TAKE CAR TO THE SERVICE STATION
 9:GET GAS AND OIL
10:INFLATE TIRES
11:
12:CONDITION: DELIVERIES HAVE TO BE CANCELLED
13:CANCEL NEWSPAPER
14:CANCEL MILK
15:
16:CONDITION: TRIP HAS TO BE PLANNED
17:GET MAPS
18:MAKE HOTEL RESERVATIONS
19:
20:ENDCHECKLIST
21:ENDPROGRAM
```

As output:

```
LINE                                                          PAGE     1
   3 PROGRAM FOR VACATION PREPARATION
   4
   5     CHECK AND ACT ON ALL TRUE CONDITIONS IN THE FOLLOWING LIST
   6
   7     CONDITION: CAR NEEDS TO BE SERVICED
   8        TAKE CAR TO THE SERVICE STATION
   9        GET GAS AND OIL
  10        INFLATE TIRES
  11
  12     CONDITION: DELIVERIES HAVE TO BE CANCELLED
  13        CANCEL NEWSPAPER
  14        CANCEL MILK
  15
  16     CONDITION: TRIP HAS TO BE PLANNED
  17        GET MAPS
  18        MAKE HOTEL RESERVATIONS
  19
  20     ENDCHECKLIST
  21 ENDPROGRAM
```

- The following forms are recommended for use when the design has progressed to the point where engineering calculations need to be expressed:

Example 1: Equation not yet determined

```
CALCULATE VEHICLE.STATE: DISTANCE.TRAVELLED (TARGETTED)
* GIVEN: VEHICLE.STATE: DISTANCE.TRAVELLED (CURRENT)
*        VEHICLE.STATE.VELOCITY (CURRENT)
*        VEHICLE.STATE.ACCELERATION (TARGETTED)
*        TIME INCREMENT
```

Example 2: Equation included

```
COMPUTE VEHICLE.STATE: DISTANCE.TRAVELLED (TARGETTED) =
        D + V*T + (A/2)*T**2
   D == VEHICLE.STATE: DISTANCE.TRAVELLED (CURRENT)
   V == VEHICLE.STATE: VELOCITY (CURRENT)
   T == TIME.INCREMENT
   A == VEHICLE.STATE: ACCELERATION (TARGETTED)
```

Indentation in the examples above may be imposed by indenting the input statements or by defining COMPUTE to be a Block Initiator keyword.

4. Specification of Module Interfaces

Explicit specification of the data passed between modules and accessed from a global store will eliminate many debugging problems in the coding and integration stages.

- Use the words GIVEN and YIELD to specify parameters transmitted to and returned from a module. Use the word USING to specify global variables accessed.

- List the GIVEN and YIELD parameters with Module Invocation statements.

  Example:

```
NOW CALCULATE_DRIVE_WHEEL_OUTPUT_REQUIRED-----------------------> ( 38)
* GIVEN: VEHICLE.STATE:
*        SCHEDULED.TIME
* YIELD: VEHICLE.STATE: TIRE.RPM, ACCELERATION
*        WHEEL FORCE REQUIRED
*        WHEEL TORQUE REQUIRED
```

In this example, NOW is the Module Invocation keyword. The lines specifying arguments passed to and from the module all begin with an asterisk to emphasize their association with the Invocation statement.

- List USING, GIVEN, and YIELD parameters with Module Initiator statements.

  Example:

```
PROCEDURE TO CALCULATE_DRIVE_WHEEL_OUTPUT_REQUIRED
   ************************************************
   *                                              *
   *  USING; DRIVE.POWER.TRAIN: DATA              *
   *         CHASSIS: DATA                        *
   *  GIVEN; VEHICLE.STATE:                       *
   *         SCHEDULED.TIME                       *
   *  YIELD; VEHICLE.STATE: TIRE.RPM, ACCELERATION *
   *         WHEEL FORCE REQUIRED                 *
   *         WHEEL TORQUE REQUIRED                *
   *                                              *
   ************************************************
```

The parameters in this structure are set off by using the #TEXT - #END directives to enclose them in a box formed by asterisks. In addition to the GIVEN and YIELD arguments, the USING category lists global parameters which are accessed by the module.

5. Inclusion of Management Information in the SDD

Project management information, just as program design, must be kept up to date and accurate. The SDD is the ideal place to maintain

this information, and the language can be used effectively to present the information. Listed below are several Module Initiator statements which have been used effectively in the VEEP and SAMIS programs. These examples are intended to suggest kinds of management information, as indicated by their wording, which might be included in the SDD.

```
PROGRAM OBJECTIVES
PROGRAM REVISIONS MEMORANDA
PROGRAM MEETING CALENDAR & AGENDA
PROGRAM DOCUMENT READING CONVENTIONS
PROGRAM COMPLETION SCHEDULE
```

6. Additional Uses of the Cross Reference Capability

The SDD typically will contain much information, in addition to the names of design parameters, for which it would be useful to have a cross reference. Individual cross reference tables for each type of information can be obtained by associating a different cross reference title with each (see the #MARK directive). Some that have proved to be useful appear below in a sample design, showing the form of the #MARK directive which establishes the cross reference character, and the way in which the data appear in the main body of the SDD. The pound sign (#) has been used in the input to cause some information to be printed at the right-hand margin of the SDD for increased readability.

Example:

As input:

```
1:#MARK REVISIONS % FOOTNOTES [ FILE NAMES $
2:#MARK UPDATE RESPONSIBILITY ?
3:PROGRAM TO PROCESS CUSTOMER DATA # [REF1]
4:READ NAMES FROM CUSTOMERSFILE # %1
5:MATCH NAMES TO CREDIT DATA # ?HK
6:WRITE CREDIT INFO TO CREDITSFILE # %2
7:ENDPROGRAM
```

As output:

```
                                TABLE OF CONTENTS                    PAGE     1
 PAGE    LINE  +-----------------------------------------------------------------+
NUMBER  NUMBER       MODULE NAME

    1       3     PROGRAM TO PROCESS CUSTOMER DATA                      [REF(1)]

    2           REVISIONS - CROSS REFERENCE LISTING

    3           FOOTNOTES - CROSS REFERENCE LISTING

    4           FILE NAMES - CROSS REFERENCE LISTING

    5           UPDATE RESPONSIBILITY - CROSS REFERENCE LISTING



                                                                     PAGE     1
LINE
    3  PROGRAM TO PROCESS CUSTOMER DATA                              [REF(1)]
    4     READ NAMES FROM CUSTOMER$FILE                                   %1
    5     MATCH NAMES TO CREDIT DATA                                    ?HK
    6     WRITE CREDIT INFO TO CREDIT$FILE                                %2
    7  ENDPROGRAM



                                REVISIONS
                          CROSS REFERENCE LISTING                    PAGE     2
IDENTIFIER+-----------------------------------------------------------------+

%1
     PAGE    1   PROGRAM TO PROCESS
      LINES   4
%2
     PAGE    1   PROGRAM TO PROCESS
      LINES   6
```

```
                              FOOTNOTES
                        CROSS REFERENCE LISTING                    PAGE     3
IDENTIFIER++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

CREF1
     PAGE    1  PROGRAM TO PROCESS
      LINES  3



                              FILE NAMES
                        CROSS REFERENCE LISTING                    PAGE     4
IDENTIFIER++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

CREDITSFILE
     PAGE    1  PROGRAM TO PROCESS
      LINES  6
CUSTOMERSFILE
     PAGE    1  PROGRAM TO PROCESS
      LINES  4



                           UPDATE RESPONSIBILITY
                        CROSS REFERENCE LISTING                    PAGE     5
IDENTIFIER++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

?HK
     PAGE    1  PROGRAM TO PROCESS
      LINES  5
```

SECTION II

SDDL USER'S REFERENCE GUIDE

Input to the SDDL processor consists of a sequence of design statements and processor control directives.

Statements and Directives begin and end with a line (or record) of the input medium, unless line continuation is explicitly indicated, as described below. Continued lines are concatenated into a single statement for processing.

A. CONTINUATION OF INPUT LINES

A continuation mark, the ampersand can be used to concatenate several input lines/cards into a single SDDL input statement. The following rules apply to its use:

(1) If the last non-blank character (excluding card sequence numbers -- see #SEQUENCE directive) of an input line is an ampersand, the processor will concatenate the next line of input with the current line to form a single statement.

(2) The ampersand which caused the continuation is removed from the newly formed line, but all other characters, including other ampersands and blanks, are used as they were input to form the new line.

(3) The continuation mark may be used on as many subsequent input lines as desired to form a single SDDL statement or directive out of several input lines.

(4) If the resulting input statement exceeds the allowable output line space, it will be handled as described below.

B. CONTINUATION OF OUTPUT LINES

Occasionally a line of output may be long enough to extend beyond the right-hand page margin. When this occurs, the processor handles the line in the following way:

(1) Beginning at the appropriate indentation level, as many characters (including blanks) of the input line as space permits are printed on the current line.

(2) On the next line of the document, an ampersand is printed one space to the right of the current indentation level, and the remaining characters are printed immediately following the ampersand. Step 2 is repeated as many times as necessary to complete the line.

(3) If the indentation level is such that no characters can be printed on the first line, then step 2 is repeated with output beginning at the left margin instead of at the indentation level.

Example:

As input:

```
1  PRIOR LINE
2  THIS IS AN EXAMPL&
3  E OF A LONG INPUT &
4  LINE & A LONG OUTP&
5  UT LINE
6  NEXT LINE
```

As printed:

```
1  PRIOR LINE
2  THIS IS AN EXAMPLE OF A LONG INPUT LIN
    &E & A LONG OUTPUT LINE
6  NEXT LINE
```

C. SDDL SYNTAX DEFINITION LEVELS

The SDDL syntax definitions are subdivided into five levels. The primitive definitions are presented in Level 0. Secondary definitions based on the primitive definitions are in level 1. Level 2 contains SDDL statement definitions. The SDDL control directives are defined in level 3. Finally, an overview diagram of an SDDL program, based on definitions in levels 2 and 3, is given in level 4. The definitions in levels 1 through 4 are accompanied by flow diagrams which specify the requirements and options of the syntax. To interpret the diagram, trace the flow line from the term being defined to the end of the definition. Boxes which are unavoidable are requirements, boxes which can be bypassed are options, and boxes which can be returned to are repeatables. The contents of a box may refer to another definition or a literal. To differentiate between them, definitions appear in smaller type, with the definition number in the lower right-hand corner, and literals, in larger type, have no accompanying number.

1. Primitive Definitions (Level 0)

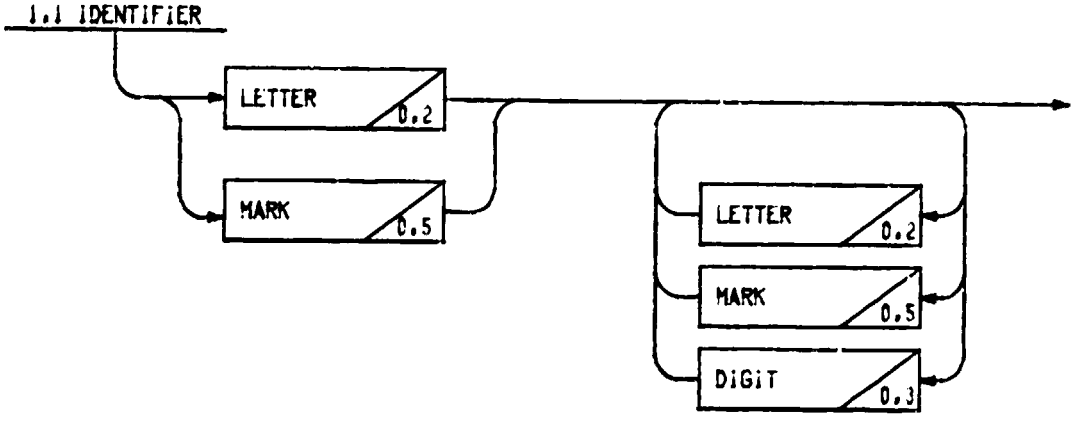
The following description and discussion of SDDL is based on the short list of primitive definitions shown in Table 2. Note especially that the definition of a letter includes the pound sign in addition to the alphabet. Also note that initially no MARK characters are defined. As will be explained later in the discussion of the #MARK directive, any punctuation may be converted to a MARK by user specification.

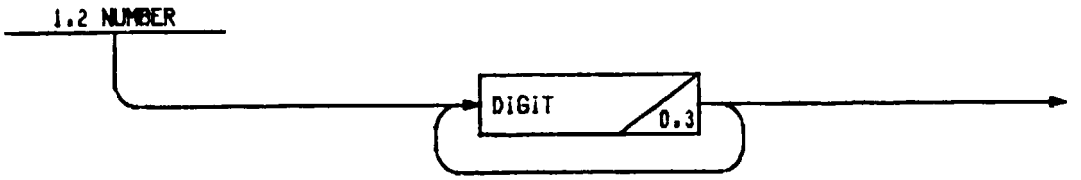
Table 2-1. SDDL Primitive Definitions

Definition Number	Name	Description
0.1	character set	The entire set of allowable characters (including the blank).
0.2	letter	The alphabet (A-Z) and the pound sign (#).
0.3	digit	The digits (0-9).
0.4	punctuation	The characters remaining after letter, digit, and the blank have been deleted from the entire character set.
0.5	mark	Any punctuation which has been converted by a control directive. (Initially, this is the empty set.)
0.6	e.o.s.	The end of an input statement or directive, determined by the end-of-line/record indicator (e.g., carriage return) of an input line without a continuation mark.

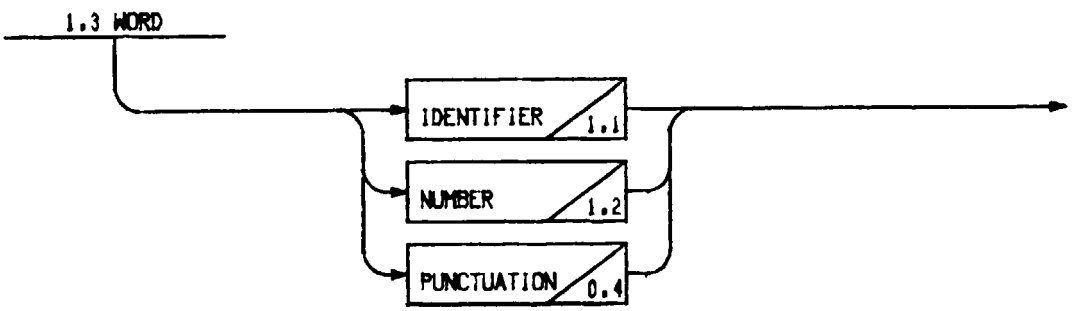
2. Secondary Definitions (Level 1)

The definitions of identifier, number, and word shown below are based on the SDDL primitive definitions shown in Table 2.





Note that a number may not have a decimal point. This constraint only affects SDDL control directives and has no impact on the design statements which appear in the SDD.



As shown above, a word can be an identifier, a number, or punctuation; in short, any token or object definable under the preceding definitions of the language. As in natural languages, the space or blank is a very important part of the syntax which is needed for delimiting or separating words.

Example:

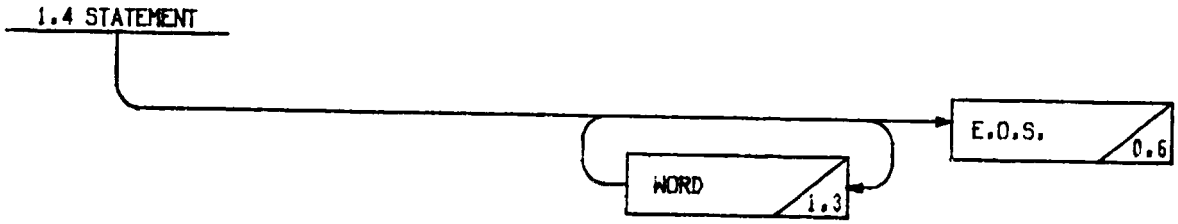
ABC123 X Y#Z?E 12 4W

Lexical analysis of the above line yields the following words:

ABC123	(identifier)
X	(identifier)
Y#Z	(identifier)
?	(punctuation)
E	(identifier)
12	(number)
4	(number)
W	(identifier)

If ? had previously been converted to a mark, the result would yield the following words:

ABC123	(identifier)
X	(identifier)
Y#Z?E	(identifier)
12	(number)
4	(number)
W	(identifier)



A statement, as shown in the diagram above, consists of a sequence (including the null case) of words.

1.5 KEYWORD

The SDDL processor is keyword-driven. A keyword is an indentifier which has been predefined to be the name of a structure part (initiator, terminator, escape, substructure), a Module Invocation word, or a control directive. Keywords are recognized only in context, i.e., only when they appear as the first word, though not necessarily starting in the first column, of the statement or directive.

The primary function (in the sense that it precedes and supports everything else) of the processor is to reproduce the input statements

on the SDD output file in a manner which enhances the reader's capabililty to understand the resulting document with the least effort. This is accomplished by indentation of statements within structures, and superimposition of flow lines to highlight structure escapes and module invocations. The actions taken by the processor in response to specific statement types are described below.

1.6 PASSIVE STATEMENT

A Passive statement is any statement which does not begin with a keyword. Passive statements may be used to convey any design information as desired but they do not have any special meaning to the processor as do the Keyword statements.

Passive statements are processed as follows:

(1) Since Passive statements must be imbedded within a module structure, if one does not already exist, the processor supplies a module, with an error message.

(2) The entire statement is scanned for the appearance of any identifiers which have been designated for inclusion in the cross reference tables. The means for designating identifiers for inclusion in the cross reference tables is explained under the discussion of the #MARK and the #STRING directives.

(3) The input line number (i.e., the number corresponding to the statement's sequential location in the input medium) is written at the left margin.

(4) The entire statement including all blanks is copied to the SDD output file beginning at the current point of indentation.

(5) If the statement contains a pound sign, the portion of the statement which follows will all be right shifted so that the last non-blank character lines up at the right margin. The pound sign itself is replaced with a space. This feature has many important applications which are examined under the discussion of the #MARK directive.

Example:

As input (input line=1) :

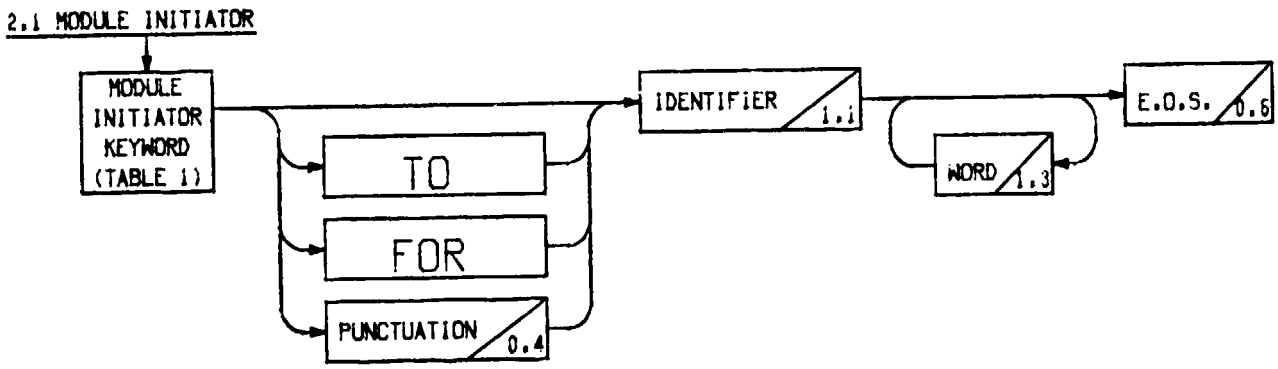
```
ADD 1 # COUNT CASES
```

As output:

```
LINE                                                    PAGE 1

        PROGRAM STATEMENT SUPPLIED BY PROCESSOR

1       ADD 1                                   COUNT CASES
```

3. Keyword Statement Definitions (Level 2)

This section describes the Keyword statements which drive the processor formatting actions. The actions are summarized in Figure 1.



Example:

PROGRAM TO READ THE PROGRAM INPUT

(1) The keyword PROGRAM is recognized as a Module Initiator.

(2) The optional noise word TO (FOR or punctuation are alternative noise words) is ignored.

(3) The next identifier, READ, is established as the module name and recorded for future cross referencing. The remaining

words, including the second appearance of PROGRAM, are all passive (i.e., they are handled as though they were part of a Passive statement).

(4) Since a module is the highest-level structure and may not be nested within other structures, the processor terminates any open structures (i.e., structures which have been initiated but left unterminated) with appropriate error messages.

(5) The entire Module Initiator statement is entered into the SDD table of contents.

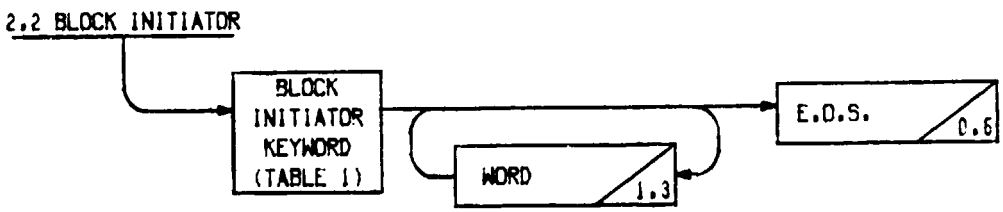
(6) The module structure is entered into a push-down (last-in, first-out) structure stack for later matching with subsequent statements specifying other parts of the structure.

(7) A new page of the SDD is started with appropriate heading.

(8) The indentation point is set to level zero (just to the right of the location of the input line number field).

(9) The statement is written to the SDD output file in the manner described above for Passive Statements.

(10) The indentation is increased one level by moving the indentation point the required number (default = 3) of spaces to the right.



Example:

LOOP UNTIL FILES A, B & C HAVE BEEN READ

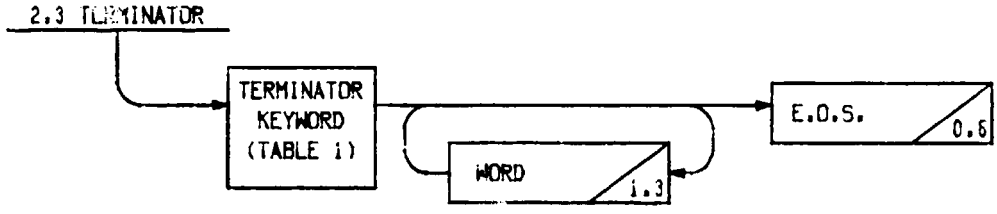
(1) The keyword LOOP is recognized as a Block Initiator keyword.

(2) Since blocks must be nested within modules, if an open module does not already exist, the processor supplies a module with an error message.

(3) The block structure is placed on the structure stack, as described above in step 6 of the Module Initiator statement.

(4) The statement is written to the SDD output file, as described above for Passive statements.

(5) Indentation is increased one level (see step 10 for the Module Initiator statement).



Example:

ENDPROGRAM TO READ INPUT

(1) The identifier ENDPROGRAM is recognized as a Terminator keyword.

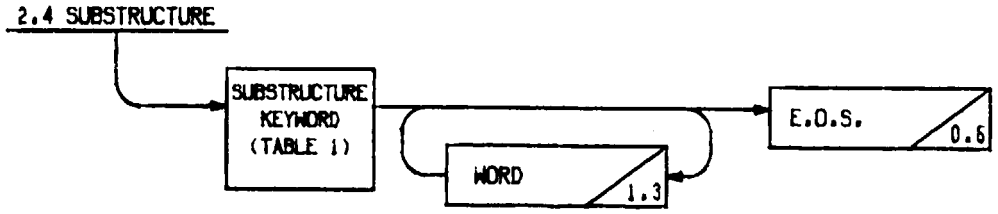
(2) The structure stack is searched for a matching Structure Initiator. If none is found, the statement is processed as a Passive statement and is followed by an error message. No further action is taken.

(3) If a matching structure is found, all intervening open structures are terminated with error messages;

(4) The structure to be terminated is removed from the top of the structure stack;

(5) Indentation is decreased one level (shifted left) to match the indentation of the Structure Initiator statement.

(6) The statement is written to the SDD output file in the manner of a Passive statement.



Example:

ELSE TRY ANOTHER ALTERNATIVE

(1) The identifier ELSE is recognized as a Substructure keyword.

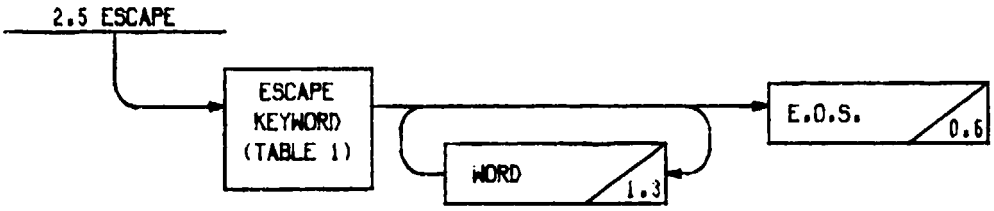
(2) The structure stack is searched for a matching Structure Initiator. If none is found, the statement is processed as a Passive statement and followed with an error message. No further action is taken.

(3) If a matching structure is found, all intervening, open structures are terminated with error messages.

(4) Indentation is decreased one level (shifted left) to match the indentation of the Structure Initiator statement.

(5) The statement is written like a Passive statement.

(6) Indentation is increased one level (shifted right), as when the structure had just been initiated, in effect re-initiating the structure.



Example:

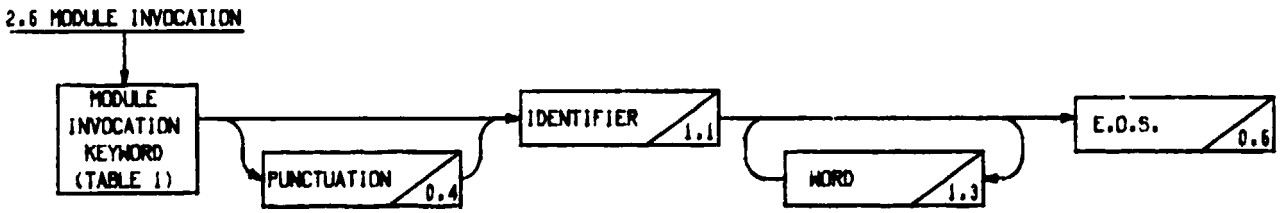
EXITLOOP IF DELTA < EPSILON

(1) The identifier EXITLOOP is recognized as an Escape keyword.

(2) The statement is written to the SDD in the manner described for the Passive statement.

(3) The structure stack is searched for a matching Structure Initiator. If none is found, an error message is added to the SDD output file.

(4) If a matching structure is found, the escape statement is completed by the addition of a flow line (left arrow) extending from the current indentation level to the indentation level of the Structure Initiator statement.



Example:

CALL : INITIALIZATION ROUTINE

(1) The identifier CALL is recognized as a Module Invocation keyword.

(2) The optional punctuation, :, is ignored.

(3) The identifier INITIALIZATION is established as the name of the module to be invoked and recorded for module cross referencing.

(4) The statement is written to the SDD in the manner described for a Passive statement.

(5) The output line is augmented by a flow line (right arrow) extending from the rightmost non-blank character of the statement to within five columns of the right-hand margin.

(6) The last five columns of the output line are filled in with parentheses enclosing the page number of the module referenced by the Module Invocation statement.

The processor actions for SDDL statements described above are summarized in Figure 1. The following example illustrates the statements as they might be combined in a simple design:

Example:

As input:

```
 1:PROGRAM TO SUMMARIZE DATA
 2:CALL INITIALIZE
 3:LOOP UNTIL ALL NUMBERS HAVE BEEN READ
 4:READ A VALUE
 5:CALL ERRORCHECK
 6:IF THE ERRORCHECK INDICATES AN ERROR
 7:PRINT THE FOLLOWING MESSAGE
 8:    "SOMETHING'S WRONG"
 9:CYCLE BACK FOR ANOTHER ITERATION
10:ELSE
11:SUM VALUES & SQUARED VALUES
12:INCREMENT COUNTER
13:ENDIF
14:REPEAT
15:DISPLAY MEAN AND STANDARD DEVIATION
16:ENDPROGRAM
17:PROCEDURE TO INITIALIZE
18:    VARIABLE                          INITIAL VALUE
19:    SUM                                    0.0 #REAL
20:    SUM OF SQUARES                         0.0 #REAL
21:    COUNT                                  0 #INTEGER
22:    LOWER BOUND                            0 #REAL
23:    UPPER BOUND                          100.0 #REAL
24:PROCEDURE FOR ERRORCHECK
25:INITIALIZE ERRORCHECK TO INDICATE AN ERROR
26:IF LOWER BOUND < VALUE
27:IF VALUE < UPPER BOUND
28:RESET ERRORCHECK TO INDICATE NO ERROR
29:EXITLOOP
30:ENDLOOP
```

As output:

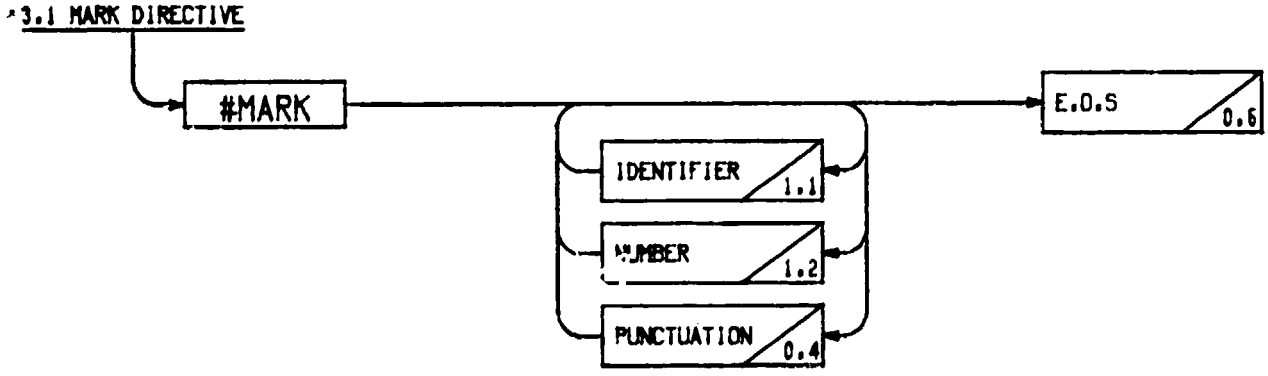
```
                                TABLE OF CONTENTS                       PAGE     1
 PAGE   LINE
NUMBER NUMBER     MODULE NAME

    1       1     PROGRAM TO SUMMARIZE DATA
    2      17     PROCEDURE TO INITIALIZE
    3      24     PROCEDURE FOR ERRORCHECK

    4           MODULE REFERENCE TREE

    5           MODULE - CROSS REFERENCE LISTING


LINE
   1 PROGRAM TO SUMMARIZE DATA                                          PAGE     1
   2    CALL INITIALIZE------------------------------------------------>(  2)
   3    LOOP UNTIL ALL NUMBERS HAVE BEEN READ
   4       READ A VALUE
   5       CALL ERRORCHECK------------------------------------------->(  3)
   6       IF THE ERRORCHECK INDICATES AN ERROR
   7          PRINT THE FOLLOWING MESSAGE
   8             "SOMETHING'S WRONG"
   9    <------CYCLE BACK FOR ANOTHER ITERATION
  10       ELSE
  11          SUM VALUES & SQUARED VALUES
  12          INCREMENT COUNTER
  13       ENDIF
  14    REPEAT
  15    DISPLAY MEAN AND STANDARD DEVIATION
  16 ENDPROGRAM


LINE
  17 PROCEDURE TO INITIALIZE                                            PAGE     2
  18       VARIABLE                          INITIAL VALUE
  19       SUM                                     0.0                    REAL
  20       SUM OF SQUARES                          0.0                    REAL
  21       COUNT                                   0                   INTEGER
  22       LOWER BOUND                             0                      REAL
  23       UPPER BOUND                           100.0                    REAL
    ENDPROCEDURE - STMT SUPPLIED BY PROCESSOR
```

```
LINE                                                                PAGE     3
  24 PROCEDURE FOR ERRORCHECK
  25    INITIALIZE ERRORCHECK TO INDICATE AN ERROR
  26    IF LOWER BOUND < VALUE
  27       IF VALUE < UPPER BOUND
  28          RESET ERRORCHECK TO INDICATE NO ERROR
  29          EXITLOOP
*****         *** ERROR *** INCORRECT MODULE ESCAPE WORD
  30          ENDLOOP
*****         *** ERROR *** INCORRECT MODULE TERMINATOR
         ENDIF - STMT SUPPLIED BY PROCESSOR
      ENDIF - STMT SUPPLIED BY PROCESSOR
   ENDPROCEDURE - STMT SUPPLIED BY PROCESSOR



   ************  MODULE REFERENCE TREE  ******          PAGE     4
 LN  PAGE
  1     1  SUMMARIZE
  2     2  . INITIALIZE
  3     3  . ERRORCHECK



                          MODULE
                  CROSS REFERENCE LISTING                PAGE     5
IDENTIFIER****************************************************************

ERRORCHECK
    PAGE   1  PROGRAM TO SUMMARIZE
     LINES  5, 6
    PAGE   3  PROCEDURE FOR ERRORCHECK
     LINES  24, 25, 28
INITIALIZE
    PAGE   1  PROGRAM TO SUMMARIZE
     LINES  2
    PAGE   2  PROCEDURE TO INITIALIZE
     LINES  17
    PAGE   3  PROCEDURE FOR ERRORCHECK
     LINES  25
SUMMARIZE
    PAGE   1  PROGRAM TO SUMMARIZE
     LINES  1
```

4. Control Directives (Level 3)

Control directives allow the user to set processor control specifications (e.g., page width, indentation) and to cause some immediate actions to be taken (e.g., page eject). Control directives are read, interpreted, and acted upon by the processor. They are not written to the SDD output file and hence are not seen in the final document. Control specifications set by directives are put into effect as soon as they are interpreted and remain in effect for all subsequent input, or until overridden by another directive. Directives can be used to set and reset processor control specifications as often as desired. The SDDL control directives are defined and described on the following pages. The sequence of presentation is intended to avoid lookahead caused by definitions based on terms defined on subsequent pages.

Control directive keywords all begin with the pound sign character. They are preset (see Table 2) and must not be altered. The user must be careful not to define a new meaning for a control directive keyword (see #DEFINE directive) since it will cause the preset definition to be overridden.



Selection of words or identifiers for cross referencing is controlled by the #MARK and the #STRING directives. When using the #MARK directive, the designer specifies a list of punctuation which the processor will subsequently treat in the following manner:

(1) All punctuation appearing in the statement is converted into a MARK (syntax definition 0.5), i.e., those characters which are used to form identifiers. They can then be used as connectors to build a single identifier out of separate words.

Example:

```
#MARK .
EVERY.GOOD.BOY DOES FINE
```

(2) Every identifier which includes a MARK, such as in EVERY.GOOD.BOY in the example above, is included in a cross reference listing produced at the end of the design document.

Titles for the cross reference listings may be supplied by placing any string of characters (except punctuation) prior to the punctuation to be converted. If no title is supplied prior to the first punctuation in the directive, a blank title is assumed.

The SDDL processor provides individual cross reference listings for each unique title found in the #MARK and/or #STRING directives. Identifiers containing MARKs which were specified with identical titles are merged into a single cross reference listing. Titles are considered to be identical if, after deleting leading and following blanks, they are an exact, character-by-character match, including internal (between word) blanks. Identifiers which contain marks associated with several unique titles will appear in each appropriate cross reference. These conventions are exemplified below, and an additional, more comprehensive example is given following the #STRING directive.

Example:

```
#MARK     ?! DATA ITEMS %  REVISIONS $

#MARK     ; DATA ITEMS .:
```

The MARKs specified in the above example are associated with the titles as follows:

CROSS REFERENCE LISTING

? ! ;

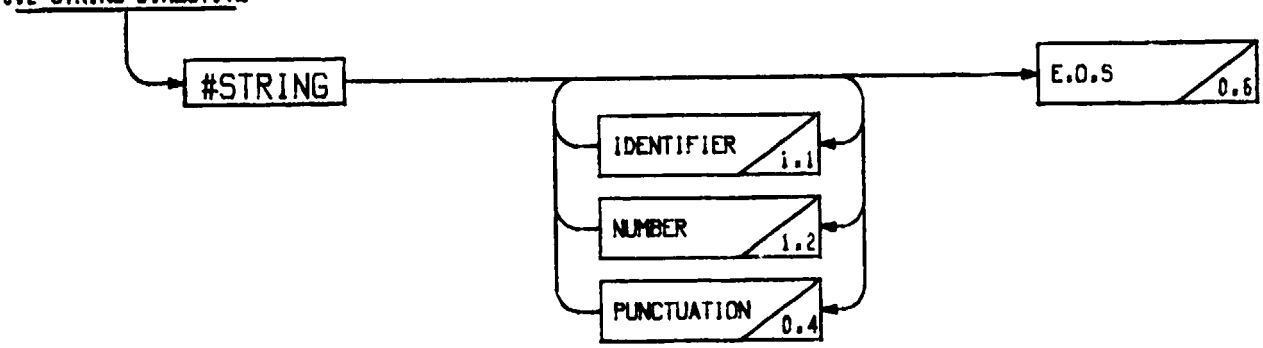
DATA ITEMS
CROSS REFERENCE LISTING

% . :

REVISIONS
CROSS REFERENCE LISTING

$

3.2 STRING DIRECTIVE



This directive allows the user to specify one or more punctuation marks to be used as string delimiters. The purpose of enclosing text within string delimiters is to have it included in a cross reference table at the end of the document. The following rules govern the use of this feature.

(1) Several MARKs may be specified as string delimiters but no distinction is made between left (opening) or right (closing) delimiters

Example:

```
#STRING ()
1     SAMPLE STATEMENT (STRING   ONE(
2     ) STRING TWO (NOT A STRING) STRING ABC)
```

In the above example, the following text segments are defined and will be cross referenced:

"STRING ONE" "STRING TWO" "STRING ABC"

(2) Preceding and following blanks are excluded from the string, but interior blanks are included.

Example:

```
#STRING '
LINE 1          '  ABC D'
LINE 2          'ABC D  '
LINE 3          'ABC   D'
```

The strings in LINE 1 and LINE 2 are the same because they match exactly after preceding and following blanks are stripped off. The string in LINE 3 does not match the others because it does not have the same number of spaces between ABC and D. Each unique string, where uniqueness is defined by rules 1 and 2, becomes a single entry in the cross reference.

(3) If the closing delimiter is omitted, the string is terminated by the end of the input statement.

Example:

```
#STRING '
LINE 1 'ABC' AND 'DEF G
```

Strings ABC and DEF G are recognized.

(4) If the text enclosed in string delimiters consists of a single identifier, regardless of preceding or following blanks, it is recognized as described above, but in addition, the processor will thereafter recognize and cross reference the named identifier whether it appears with or without delimiters.

Example:

```
#STRING "
LINE 1          "VEHICLE "
LINE 2          VEHICLE AND VEHICLE
```

In the above example, VEHICLE is recognized and the cross reference will show that it was found once in LINE 1 and twice in LINE 2.

(5) A title for the cross referencing of text strings may be supplied by including any characters except punctuation between the #STRING keyword and the first MARK to be converted to a string delimiter.

The title (including the null case) supplied with the #STRING directive is compared with the titles supplied with the #MARK directives for merging of the cross reference listings. When several #STRING or #MARK directives with conflicting title specifications are used, the rule followed is that the last usage overrides all prior usage.

Example:

As input:

```
1:*MARK 7;  DATA ITEMS %     REVISIONS  $
2:*MARK DATA ITEMS .;
3:*STRING DATA ITEMS "
4:PROGRAM TO READ DATA AND "CHECK" IT
5:READ VEHICLE:  ,  MAX.RPM , $POWER , "AND WHAT EVER ELSE THERE IS "
6:IF ANY VALUES ARE UNKNOWN?  OR UNTESTED?
7:CHECK THE DATA;;  FOR DOUBTFUL.STUFF?  $1
8:ENDIF
9:AN ADDITIONAL CHECK MAY BE NEEDED HERE
10:ENDPROGRAM
```

As output:

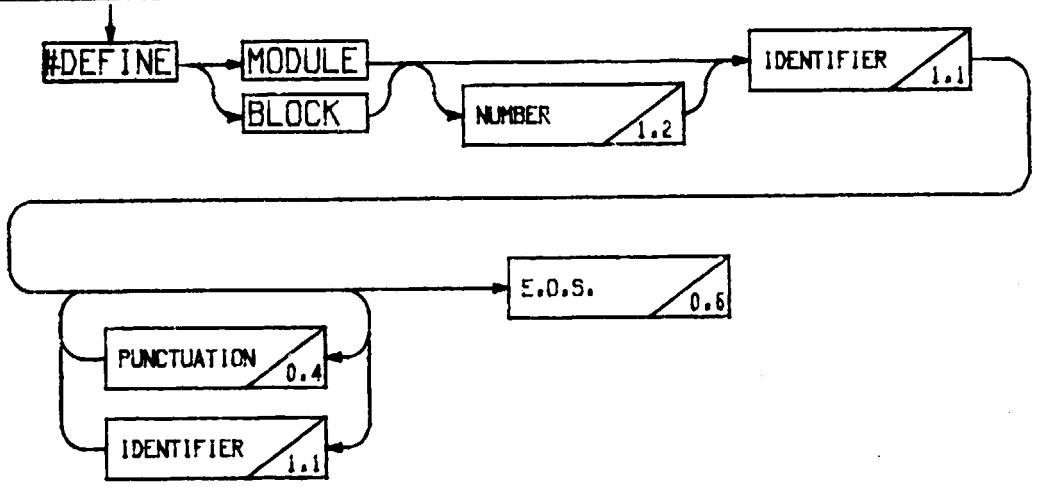
```
                                TABLE OF CONTENTS                PAGE    1
 PAGE   LINE .....+.....+.....+.....+.....+.....+.....+.....+.....+.....+
NUMBER NUMBER     MODULE NAME

   1       4    PROGRAM TO READ DATA AND "CHECK" IT

   2            MODULE REFERENCE TREE

   3            MODULE - CROSS REFERENCE LISTING

   4            DATA ITEMS - CROSS REFERENCE LISTING

   5            REVISIONS - CROSS REFERENCE LISTING

   6            CROSS REFERENCE LISTING


                                                                 PAGE    1
LINE
   4 PROGRAM TO READ DATA AND "CHECK" IT
   5    READ VEHICLE:  ,  MAX.RPM , $POWER , "AND WHAT EVER ELSE THE
         &RE IS "
   6    IF ANY VALUES ARE UNKNOWN?  OR UNTESTED?
   7       CHECK THE DATA;;  FOR DOUBTFUL.STUFF?  $1
   8    ENDIF
   9    AN ADDITIONAL CHECK MAY BE NEEDED HERE
  10 ENDPROGRAM
```

```
                                DATA ITEMS
                          CROSS REFERENCE LISTING                    PAGE    4
IDENTIFIER++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

$POWER
     PAGE    1  PROGRAM TO READ
      LINES  5
AND WHAT EVER ELSE THERE IS
     PAGE    1  PROGRAM TO READ
      LINES  5
CHECK
     PAGE    1  PROGRAM TO READ
      LINES  4, 7, 9
DOUBTFUL.STUFF?
     PAGE    1  PROGRAM TO READ
      LINES  7
MAX.RPM
     PAGE    1  PROGRAM TO READ
      LINES  5
VEHICLE:
     PAGE    1  PROGRAM TO READ
      LINES  5



                                REVISIONS
                          CROSS REFERENCE LISTING                    PAGE    5
IDENTIFIER++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

$1
     PAGE    1  PROGRAM TO READ
      LINES  7



                          CROSS REFERENCE LISTING                    PAGE    6
IDENTIFIER++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

DATA!!
     PAGE    1  PROGRAM TO READ
      LINES  7
DOUBTFUL.STUFF?
     PAGE    1  PROGRAM TO READ
      LINES  7
UNKNOWN?
     PAGE    1  PROGRAM TO READ
      LINES  6
UNTESTED?
     PAGE    1  PROGRAM TO READ
      LINES  6
```

The #DEFINE directive is used to specify new or to delete old SDDL keywords. To select the desired action, one of the four words shown below must follow the SDDL keyword, #DEFINE.

MODULE BLOCK CALL NULL

3.3 DEFINE DIRECTIVE (MODULE, BLOCK)



The word MODULE or BLOCK is used to define a control structure. In SDDL, a control structure has four parts:

(1) Initiator: Increases the indentation level for subsequent lines.

(2) Terminator: Closes all nested structures left open and returns the indentation level to that of the Initiator statement.

(3) Escape: A left arrow is added to the statement to indicate the program control flow. The arrow extends from the indentation level of the escape statement to the indentation level of the Initiator statement.

(4) Substructure: Closes all nested structures left open, returns the indentation level to that of the Initiator statement, prints the line, and increases the indentation level.

When defining a module or block, names for the four parts must be specified in the order shown above. Any punctuation may be used to separate the part names, but care must be taken to avoid using a MARK (i.e., punctuation which has been converted to a MARK by the #MARK directive). Names for any of the parts except the initiator may be omitted by using consecutive punctuation to show where a name has been left out. Any text following the name of the substructure will be ignored. Synonyms for part names, except for the initiator name, may be established by additional #DEFINE directives.

Indentation specific to the named structure may be indicated by including an unsigned integer between the word MODULE (BLOCK) and the initiator name. If a zero is specified or the integer is omitted, the current default indentation amount (see #INDENT) will be used.

Example (three equivalent directives):

```
#DEFINE MODULE 10 PROGRAM, END, STOP, ENTRYPOINT
#DEFINE MODULE 10 PROGRAM END, STOP ENTRYPOINT
#DEFINE MODULE 10 PROGRAM END STOP ENTRYPOINT WHATEVER
```

type	indentation	initiator	terminator	escape	substructure
module	10	PROGRAM	END	STOP	ENTRYPOINT

Example:

```
#DEFINE BLOCK BEGIN END
```

type	indentation	initiator	terminator	escape	substructure
block	default	BEGIN	END		

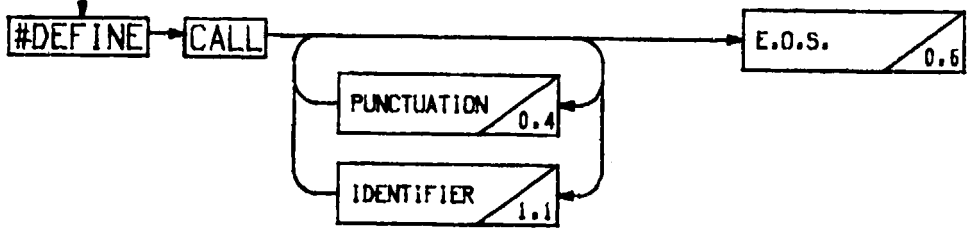
Example:

```
#DEFINE BLOCK   START, FINISH, LEAVE
#DEFINE BLOCK   START, , SCRAM
#DEFINE BLOCK 2 START, , VAMOOSE
```

type	indentation	initiator	terminator	escape	substructure
block	2	START	FINISH	LEAVE SCRAM VAMOOSE	---

Note that in this example, the last directive established the indentation amount to be two columns, overriding the default indentation amount indicated on the previous directives.

3.3 DEFINE DIRECTIVE (MODULE INVOCATION)



The word CALL is used with the #DEFINE directive to establish synonyms for the Module Invocation keyword (default keywords are CALL and DC), which indicates that a module is to be invoked at the point where the statement occurs. The identifiers to be defined as synonyms are listed after the word CALL. Punctuation for separating the words is optional.

Example:

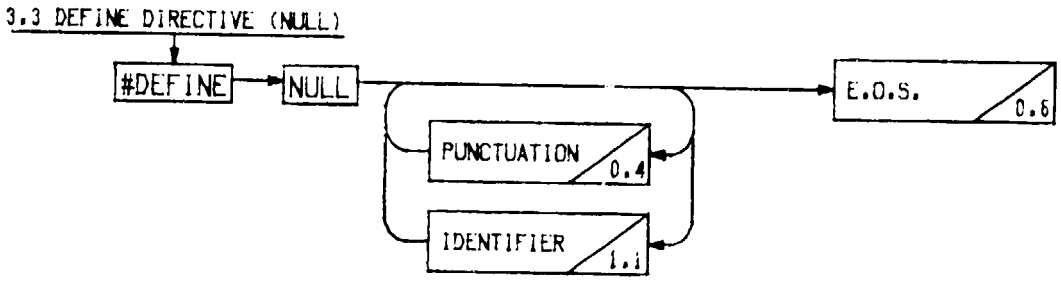
```
#DEFINE CALL PERFORM EXECUTE, GOGOGO
#DEFINE CALL DOITNOW
```

Example:

```
#MARK .
#DEFINE CALL DO.IT.NOW, PERFORM
```

The identifier DO.IT.NOW (also PERFORM) becomes a Module Invocation keyword because the period has been converted to a MARK by the prior #MARK directive. Where DO.IT.NOW appears in the context of a keyword (first word of the statement), it will not be included in the cross reference table.

When a Module Invocation statement is encountered, the processor places the statement in the output file with the appropriate indentation and adds a right arrow from the rightmost character in the line to the right margin. Matching parentheses are added to the right of the arrow to provide a place for adding the page number of the called module. If the module that is referenced in the Module Invocation statement has been defined on a prior page, the page number is supplied in the allocated space when the statement is encountered. Page reference numbers which cannot be supplied immediately must be filled in on a second pass over the output file. The user may exercise the P option at execution time to suppress the second pass, which supplies the remaining page reference numbers.



The NULL action of this directive provides a means for returning any previously defined keywords to the state of being undefined. Punctuation may be used as a keyword separator, if desired. MARKs which have been converted to letters by a previous #MARK or #STRING directive may also be listed for redefinition as punctuation. MARKs being redefined in this manner must have adjacent blanks or punctuation to disassociate them from other text.

Example:

```
#DEFINE NULL PROGRAM, ENDPROGRAM PROCEDURE
```

The words PROGRAM, ENDPROGRAM, and PROCEDURE are not recognized as keywords in the statements following this directive.

Example:

```
#MARK .$
#DEFINE NULL DO.IT.NOW $
```

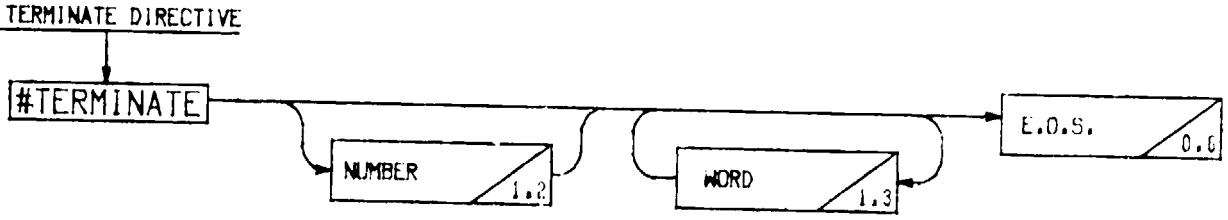
The word DO.IT.NOW is no longer a keyword and $ reverts to punctuation again. The periods in the keyword DO.IT.NOW are part of the identifier (unlike the $ in the example), and therefore the status of the period remains unchanged; i.e., it is still a MARK.

Example:

```
#MARK .
#DEFINE NULL . DO.IT.NOW
```

This example differs in that the status of the period is reconverted to punctuation first and is treated as such in the remainder of the statement. Therefore, DO, IT, and NOW are the words which become undefined. If DO, IT, and NOW are already undefined, they are not affected.

3.4 TERMINATE DIRECTIVE



This directive is a generalized terminator for block structures. It may be used in place of a number of specific terminators (specific terminators must match their respective initiators) to terminate the n innermost, nested, open block structures. If no integer is specified in the directive, only one structure will be terminated. If n is greater than the number of open block structures, they will all be terminated, but the module structure will not be affected.

Example:

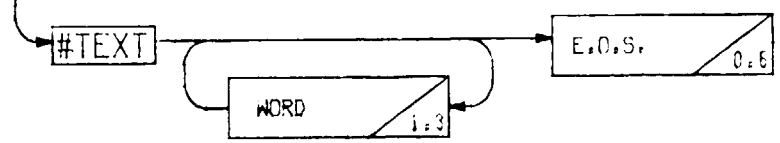
As input:

```
 1:PROGRAM "TERMINATE" EXAMPLE
 2:IF P INDENT 1 LEVEL
 3:LOOP Q INDENT 1 LEVEL
 4:INDENTATION IS 3 LEVELS DEEP
 5:ENDLOOP - SPECIFIC TERMINATOR
 6:ENDIF - SPECIFIC TERMINATOR
 7:IF P INDENT 1 LEVEL
 8:LOOP Q INDENT 1 LEVEL
 9:INDENTATION IS 3 LEVELS DEEP
10:#TERMINATE 100
11:ALL BLOCK STRUCTURES ARE TERMINATED - MODULE NOT AFFECTED
12:IF P INDENT 1 LEVEL
13:LOOP Q INDENT 1 LEVEL
14:INDENTATION IS 3 LEVELS DEEP
15:#TERMINATE ONLY ONE STRUCTURE TERMINATED
16:IF P INDENT 1 LEVEL
17:INDENTATION IS STILL 3 LEVELS DEEP
18:ENDPROGRAM - STRUCTURES LEFT OPEN ARE TERMINATED BY THE PROCESSOR
```

As output:

```
LINE                                                          PAGE     1
   1 PROGRAM "TERMINATE" EXAMPLE
   2    IF P INDENT 1 LEVEL
   3       LOOP Q INDENT 1 LEVEL
   4          INDENTATION IS 3 LEVELS DEEP
   5       ENDLOOP - SPECIFIC TERMINATOR
   6    ENDIF - SPECIFIC TERMINATOR
   7    IF P INDENT 1 LEVEL
   8       LOOP Q INDENT 1 LEVEL
   9          INDENTATION IS 3 LEVELS DEEP
  11    ALL BLOCK STRUCTURES ARE TERMINATED - MODULE NOT AFFECTED
  12    IF P INDENT 1 LEVEL
  13       LOOP Q INDENT 1 LEVEL
  14          INDENTATION IS 3 LEVELS DEEP
  16       IF P INDENT 1 LEVEL
  17          INDENTATION IS STILL 3 LEVELS DEEP
           ENDIF - STMT SUPPLIED BY PROCESSOR
       ENDIF - STMT SUPPLIED BY PROCESSOR
  18 ENDPROGRAM - STRUCTURES LEFT OPEN ARE TERMINATED BY THE PROCESS
     &OR
```

3.5 TEXT DIRECTIVE



Examples:

#TEXT

#TEXT COMMENTARY BEGINS ON NEXT LINE

The #TEXT directive is used to signal the beginning of a sequence of lines (not statements) of text intended as commentary to the SDD. When this directive is encountered, the processor performs the following actions:

(1) The words following the keyword are ignored.

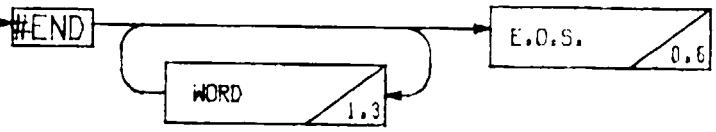
(2) The processor begins reading input lines into a holding buffer and continues until it encounters an input line whose first non-blank character is the pound sign.

(3) The lines buffered in step 2 (this does not include the line which terminated step 2) are not analyzed as statements but simply saved unaltered.

(4) The buffered lines, enclosed in a box formed by asterisks, are then written to the SDD output file at the current level of indentation.

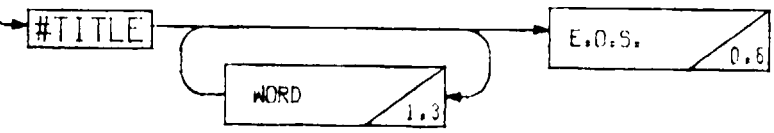
(5) The line which signaled the end of step 2 (the buffering step) is then processed in the usual way. Thus, any control directives or any statement which begins with a pound sign may be used as a terminator and still be recognized for regular processing. If no action other than termination of the text statement is desired, the #END directive may be used.

3.6 END DIRECTIVE



This directive has no effect other than that of terminating line buffering for #TEXT and #TITLE directives.

3.7 TITLE DIRECTIVE



Example:

#TITLE SDDL DESIGN DOCUMENT

This directive is used to produce a title page in the SDD. The #TITLE directive is similar to the #TEXT directive, but different in that the #TEXT directive resembles a Block Initiator statement while the #TITLE directive resembles a Module Initiator statement. The processor performs the following actions in response to input of a #TITLE directive.

(1) The keyword #TITLE is recognized.

(2) The initial pound sign is stripped off, and the remainder of the directive is entered into the SDD Table of Contents. Title line entries in the Table of Contents are preceded by a blank line and are written two columns to the left

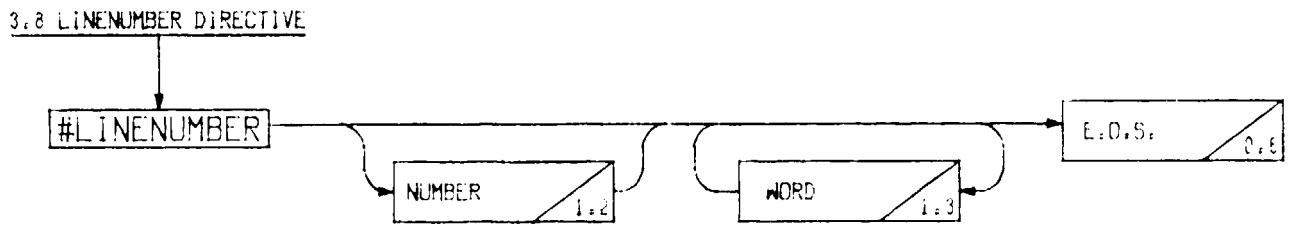
of module entries in order to distinguish them as the beginning of a document section.

(3) All structures left open are terminated with error messages.

(4) As in the case of a #TEXT directive, the processor reads and buffers input lines until it encounters a line whose first non-blank character is a pound sign. Termination of the title text is the same as for the #TEXT directive.

(5) A new page is started in the SDD output file.

(6) A title page is formed by (a) enclosing the lines in a box formed by asterisks, (b) centering each line within the box, and (c) centering the entire box on the page.



This directive provides control of the starting point of the input line numbering sequence which the processor produces in the left margin of the SDD.

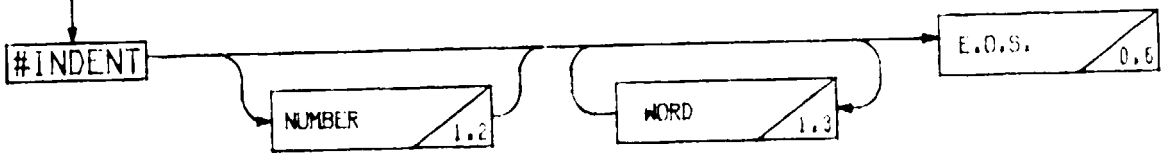
The input line numbers supplied by the SDDL processor correspond exactly to the positional line numbers of the data element (card deck) of the input to the SDDL processor. This feature obviates the listing of the raw input for revising and augmenting the SDD. Where more than one element (deck) is used as input to SDDL, it is desirable to reset the line counter so that numbering can be made to match the subsequent elements (card decks.)

If this instruction is issued without an accompanying integer, the processor will begin numbering subsequent lines from 1; otherwise it will begin numbering with the value specified by the integer. The syntax of this directive allows noise to be used for commentary if desired.

Examples:

```
#LINENUMBER 1001 STARTS THE NEXT ELEMENT

#LINENUMBER
```

3.9 INDENT DIRECTIVE



The SDDL #INDENT directive allows the user to override the default value for the number of spaces to be skipped for automatic statement indentation.

User-defined structures (see #DEFINE directive 3.3) which do not have a specific indentation amount declared and SDDL initial structure definitions always use the current default indentation value. The initial value of the system defined default indentation amount is three spaces.

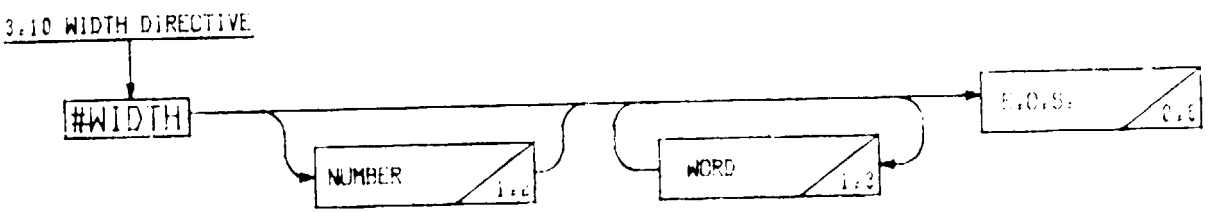
Text following the integer (i.e., noise) may be used for commentary if desired. If no integer is specified in the directive, the default value of three spaces is assumed.

Examples:

#INDENT 5 SPACES UNLESS OTHERWISE SPECIFIED

#INDENT SET TO DEFAULT OF THREE SPACES

3.10 WIDTH DIRECTIVE



The #WIDTH directive provides user control of the width of the output pages. The default page width is 80 characters = 20 cm (8 in.).

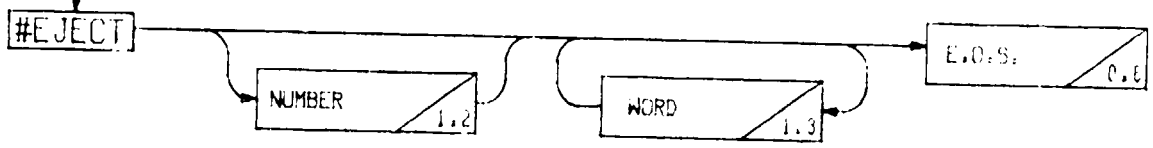
An integer specifying the width, in characters/output line, should be supplied. If the integer value is not in the range 60-130, an error message will be printed and the page width will not be altered. If no integer is specified in the directive, the default value of 80 columns is assumed.

This directive may be used as many times as desired throughout the program. Each use affects only the output which follows it.

Example:

```
#WIDTH 130 COLUMNS FOR A TABLE
      .
      .
      .
#WIDTH RESUME NORMAL PAGE WIDTH
```

3.11 EJECT DIRECTIVE



This directive provides immediate control of the start of a new page in the SDD. This page control is over and above the automatic new page start caused by (1) a title, (2) the beginning of a new module, or (3) page overflow. When a module becomes lengthy enough to cause an overflow to a new page, it is often desirable to control the start of the new page to prevent a group of lines from being split over a page boundary.

The #EJECT directive, without an accompanying integer, causes a new page to be started beginning with the next SDDL statement in the input stream.

Examples:

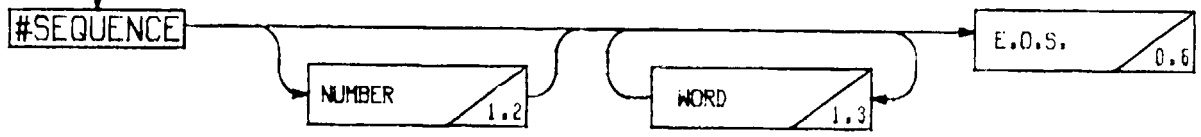
```
#EJECT

#EJECT A PAGE NO MATTER WHAT
```

When an integer is included in this command, it causes a new page to be started only when the remainder of the page cannot accommodate the number of lines specified by the value of the integer. An integer value greater than 50 gives rise to an error message and causes the directive to be ignored. Noise following the integer is ignored and may therefore be used for commentary.

Examples:

```
#EJECT 5

#EJECT 7 THE FOLLOWING 7 LINES MUST BE KEPT TOGETHER
```

3.12 SEQUENCE DIRECTIVE



The #SEQUENCE directive is provided for card input to the SDDL processor. When SDDL is used in a timesharing environment with file management and editing capabilities, card sequencing is unnecessary. In this case, the full 80 columns of the input line may be used entirely for SDDL statements and directives and the #SEQUENCE directive can be ignored, except to avoid its inadvertent use. The input line numbers supplied in the left margin of the output file correspond exactly to the line to edit the input file for corrections and updates and may be reliably used for this purpose. This feature makes it unnecessary to print out copies of the raw input file.

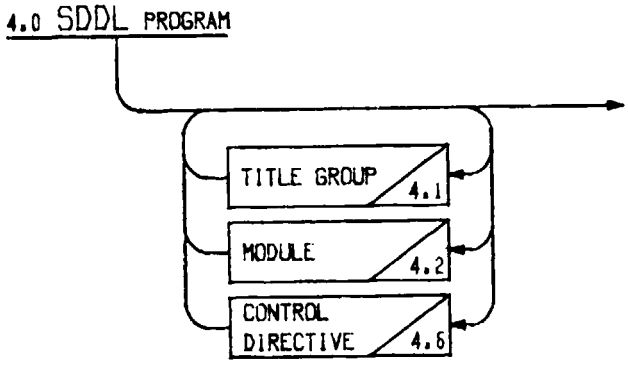
Where cards are used as the input medium, it may be desirable to have card sequence numbers at the right-hand edge of the card, in which case the #SEQUENCE directive must be used to differentiate between the input text and the sequence numbers. As shown in the syntax diagram above, the #SEQUENCE keyword may be followed by an optional integer. This integer may be used to specify the number of rightmost columns to be considered to contain sequence numbers. If no integer is supplied or a value greater than 8 is specified, the default value of eight characters, columns 73 through 80, is assumed. An integer value of zero has the effect of disabling the card sequence capability. When the #SEQUENCE capability is used, the input line (except for the sequence numbers) is handled in the usual way, and the sequence numbers are printed in the rightmost columns of the output page as determined by the #WIDTH directive (default = 80 columns). Where an input line is continued over more than one card, only the sequence number of the last card is printed.

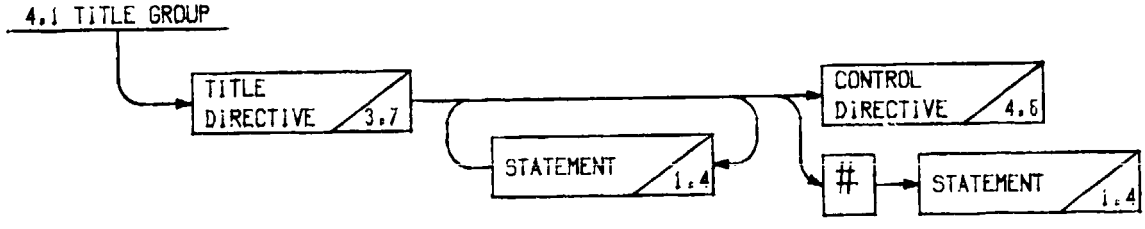
Example:

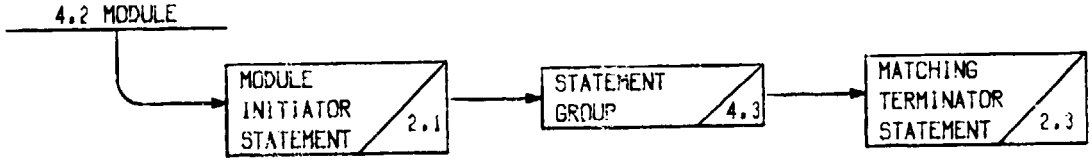
#SEQUENCE 4

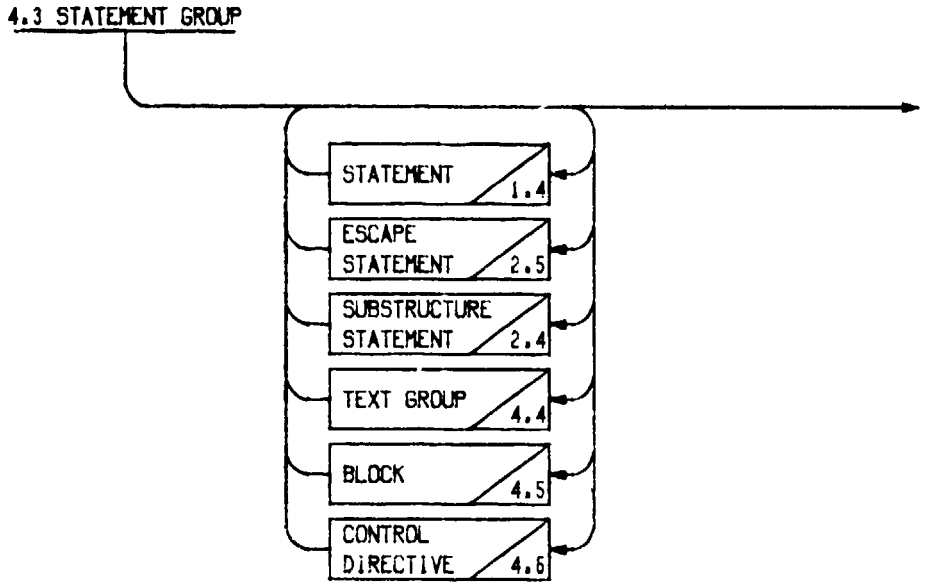
Columns 1 through 76 of the input deck are assumed to contain SDDL statements or directives, and columns 77 through 80 are assumed to contain sequence numbers.

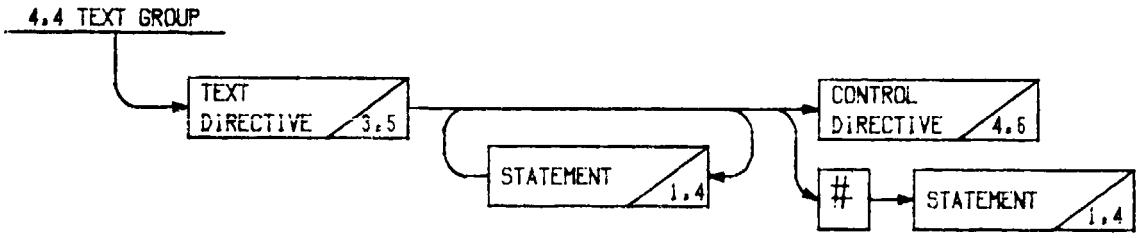
5. SDDL Syntax Overview Diagrams (Level 4)

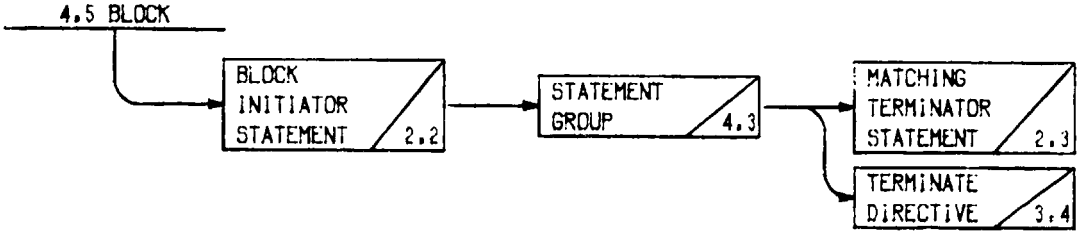


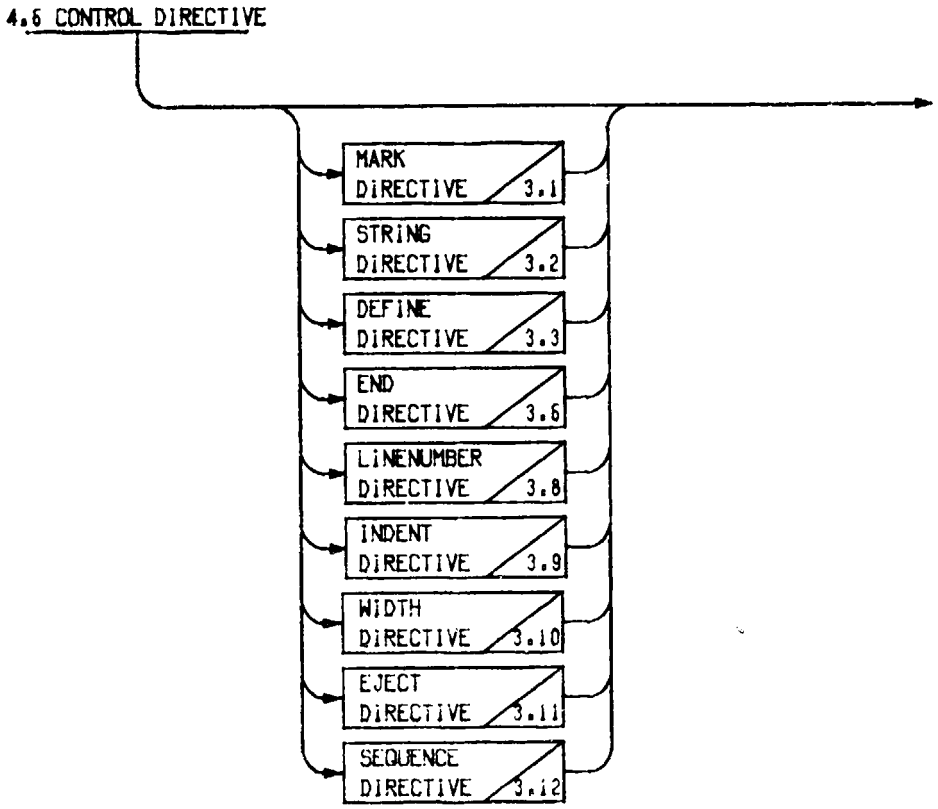





4.3 STATEMENT GROUP
STATEMENT 1.4
ESCAPE STATEMENT 2.5
SUBSTRUCTURE STATEMENT 2.4
TEXT GROUP 4.4
BLOCK 4.5
CONTROL DIRECTIVE 4.6


4.4 TEXT GROUP
TEXT DIRECTIVE 3.5
STATEMENT 1.4
CONTROL DIRECTIVE 4.6
#
STATEMENT 1.4


4.5 BLOCK
BLOCK INITIATOR STATEMENT 2.2
STATEMENT GROUP 4.3
MATCHING TERMINATOR STATEMENT 2.3
TERMINATE DIRECTIVE 3.4


4.6 CONTROL DIRECTIVE
MARK DIRECTIVE 3.1
STRING DIRECTIVE 3.2
DEFINE DIRECTIVE 3.3
END DIRECTIVE 3.6
LINENUMBER DIRECTIVE 3.8
INDENT DIRECTIVE 3.9
WIDTH DIRECTIVE 3.10
EJECT DIRECTIVE 3.11
SEQUENCE DIRECTIVE 3.12

SECTION III

SAMPLE DESIGN

Two examples are presented to illustrate the capability and potential of the SDDL processor. The design of the SDDL processor itself is the subject of the first example. Only a small subset of the actual SDDL design is shown in order to reduce the example size to expedient proportions. Even this small, top-level portion of the SDDL processor design, however, reveals information which has an important impact on the processor.

The second example demonstrates some of the actions taken by the processor in response to error situations. The subject material is not intended to convey any meaningful design information.

Example 1. Top-level SDD for the SDDL processor:

As input:

```
 1:*MARK REVISIONS * PROGRAM PORTABILITY CONSIDERATIONS ?
 2:*MARK ROUTINES AND FUNCTIONS _ DATA ITEMS
 3:*STRING DATA ITEMS "
 4:*DEFINE BLOCK 2 LIST
 5:*DEFINE BLOCK 2 MEMBER
 6:*DEFINE BLOCK LOOP, , , BEGIN
 7:*TITLE SDDL EXAMPLE
 8:
 9:SOFTWARE DESIGN AND DOCUMENTATION LANGUAGE
10:
11:
12:(AN ILLUSTRATION OF THE APPLICATION OF SDDL USING THE)
13:(SDDL PROCESSOR ITSELF AS THE OBJECT OF THE EXAMPLE. )
14:
15:*END
16:PROGRAM OBJECTIVES
17:*TEXT
18:    THE OBJECTIVE OF SDDL IS TO PROVIDE AN EFFECTIVE COMMUNICATIONS
19:MEDIUM TO SUPPORT THE DESIGN AND DOCUMENTATION OF COMPLEX SOFTWARE
20:APPLICATIONS.  THIS OBJECTIVE IS MET BY PROVIDING:
21:
22:    (1) A DESIGN AND DOCUMENTATION LANGUAGE WITH FORMS AND SYNTAX
23:        THAT ARE SIMPLE, UNRESTRICTIVE, AND COMMUNICATIVE
24:
25:    (2) A PROCESSOR WHICH CAN CONVERT DESIGN SPECIFICATIONS INTO AN
26:        INTELLIGIBLE, INFORMATIVE, MACHINE REPRODUCIBLE DOCUMENT
27:
28:    (3) METHODOLOGY FOR EFFECTIVE USE OF THE LANGUAGE AND PROCESSOR
29:
30:*END
31:PROGRAM DATA_STRUCTURE AND GLOSSARY
32:
```

```
33:INPUT.TEXT.BUFFER                    A GLOBAL CHARACTER ARRAY CONTAINING
34:                                     A SINGLE INPUT STATEMENT FORMED BY
35:                                     CONCATENATION OF CONTINUED INPUT LINES
36:
37:TEXT.LENGTH                          THE LENGTH OF THE CURRENT INPUT LINE
38:                                     (TRAILING BLANKS NOT INCLUDED)
39:
40:LIST: TOKEN.DICTIONARY               LINKED LIST OF DICTIONARY ENTRIES
41:MEMBER ENTITY: ENTRY                 POINTER TO A SINGLE DICTIONARY ENTRY
42:CHARACTER.COUNT                      NUMBER OF CHARACTERS IN THE ENTRY
43:TEXT.POINTER                         POINTER TO THE CHARACTER ARRAY
44:                                     CONTAINING THE TEXT OF THE ENTRY
45:PROGRAM.NAME                         IF ENTRY IS A KEYWORD THIS IS THE
46:                                     LOCATION OR IDENTIFICATION OF THE
47:                                     ROUTINE FOR PROCESSING THE STMT
48:                                     VALUE=0 IF ENTRY IS NOT A KEYWORD
49:LIST: REFERENCE.LIST                 FIRST-IN,FIRST-OUT LIST OF
50:                                     REFERENCES TO THE ENTRY
51:MEMBER ENTITY: "REFERENCE"
52:PAGE.NUMBER
53:LINE.NUMBER
54:*TERMINATE 4
55:
56:LIST: MODULE.STACK                   PUSH DOWN STACK OF NODES REPRESENTING
57:                                     THE NESTED STRUCTURES OF THE DESIGN
58:MEMBER ENTITY: NODE
59:NODE.NAME                                ( IF,LOOP,PROGRAM,ETC)
60:INDENTATION.COLUMN
61:*TERMINATE 2
62:ENDPROGRAM DATA_STRUCTURE
63:PROGRAM MAIN ROUTINE
64:CALL INITIALIZATION ROUTINE
65:LOOP UNTIL ALL INPUT DATA HAS BEEN PROCESSED
66:CALL GET_STATEMENT # %1
67:*YIELD TEXT.LENGTH
68:
69:CALL TOKEN_FINDER (FINDS THE FIRST TOKEN IN THE STATEMENT)
70:*YIELD TOKEN.TYPE
71:
72:IF TOKEN.TYPE IS "IDENTIFIER"
73:CALL ENTABLE TO FIND THE TOKEN IN THE TOKEN.DICTIONARY
74:ENDIF
75:
76:IF THE TOKEN WAS FOUND AND IT IS A KEYWORD
77:CALL KEYWORD_PROCESSOR
78:ELSE THE STATEMENT DOES NOT BEGIN WITH A KEYWORD
79:IF THE MODULE.STACK IS EMPTY
80:PUSH A PROGRAM MODULE ON THE MODULE.STACK
81:ENDIF
82:CALL SOURCE_LISTER TO SEND THE STATEMENT TO THE OUTPUT FILE
83:ENDIF
84:
85:FLUSH ANY "ERROR MESSAGES" TRIGGERED BY THE STATEMENT
86:REPEAT
```

```
 87:CALL WRAP_UP
 88:EXITPROGRAM
 89:ENDPROGRAM
 90:PROCEDURE:  GET_STATEMENT # %1
 91:•USING INPUT•TEXT•BUFFER
 92:•YIELD TEXT•LENGTH
 93:
 94:READ AN INPUT RECORD
 95:LOOP UNTIL A NON-BLANK RECORD IS FOUND
 96:IF THE MODULE•STACK IS NOT EMPTY (A MODULE EXISTS)
 97:PRINT THE INPUT RECORD NUMBER AND A BLANK LINE TO THE "SDD"
 98:ENDIF
 99:READ ANOTHER INPUT RECORD
100:REPEAT
101:COPY THE INPUT RECORD INTO THE INPUT•TEXT•BUFFER
102:SET TEXT•LENGTH = "USABLE COLUMNS"( 80 - CARD SEQUENCE COLS) # ???
103:LOOP
104:FIND THE LAST NON-BLANK CHARACTER IN INPUT•TEXT•BUFFER
105:SET TEXT•LENGTH = COLUMN NUMBER OF THE CHARACTER
106:IF THE CHARACTER IS NOT A CONTINUATION•MARK
107:EXITPROCEDURE
108:ENDIF
109:SUBTRACT 1 FROM THE TEXT•LENGTH (BACK UP OVER THE CONTINUATION•MARK)
110:IF THERE IS NO MORE DATA (END OF FILE ENCOUNTERED)
111:EXITPROCEDURE
112:ENDIF
113:IF THE SPACE LEFT IN INPUT•TEXT•BUFFER < 80 CHARACTERS # ???
114:EXPAND INPUT•TEXT•BUFFER BY AT LEAST 80 CHARACTERS # ???
115:ENDIF
116:READ IN ANOTHER INPUT RECORD
117:COPY THE INPUT RECORD INTO INPUT•TEXT•BUFFER BEGINNING AT TEXT•LENGTH
118:ADD "USABLE COLUMNS" TO TEXT•LENGTH
119:REPEAT
120:ENDPROCEDURE
121:PROCEDURE FOR INITIALIZATION
122:READ IN EXECUTION TIME OPTION FLAGS FROM EXECUTION STATEMENT
123:    OPTION•B = BREAKPOINT
124:    OPTION•C = CROSS REFERENCE
125:    OPTION•E = "ERROR MESSAGES"
126:    OPTION•K = KEYWORDS
127:    OPTION•M = MODULE CROSS REFERENCE
128:    OPTION•P = PAGE REFERENCE NUMBERS
129:    OPTION•R = REFERENCE TREE
130:    OPTION•T = TABLE OF CONTENTS
131:
132:IF OPTION•B IS NOT SET  BREAKPOINTING IS REQUIRED
133:READ IN REMAINDER OF EXECUTION STATEMENT
134:IF A NAME IS SPECIFIED FOR THE SDD OUTPUT FILE
135:SET UP A BUSE RELATIONSHIP WITH SDD
136:ENDIF
137:CATALOG AND ASSIGN SDD  AS THE OUTPUT FILE
138:IF THE CATALOG STEP FAILED
139:PRINT AN ERROR MESSAGE
```

```
140:TERMINATE THE PROCESSOR
141:EXITPROCEDURE
142:ENDIF
143:BREAKPOINT THE OUTPUT TO SDD
144:ENDIF
145:ESTABLISH THE FOLLOWING MACHINE DEPENDENT CONSTANTS
146:   CHARACTERS.PER.WORD          = 6                              * ???
147:   BUFFER.COUNT                = 14 (14*6=84 CHARS/LINE)        * ???
148:   READ.UNIT                   = 5                              * ???
149:   WRITE.UNIT                  = 6                              * ???
150:   DEFAULT.INDENT              = 3
151:   RIGHT.MARGIN                = 80
152:
153:INITIALIZE INPUT.TEXT.BUFFER TO AT LEAST 80 CHARACTERS    * ???
154:ESTABLISH TOKEN.DICTIONARY DATA STRUCTURE
155:CALL KEYWORD_SET_UP TO ESTABLISH DEFAULT KEYWORDS
156:EXITPROCEDURE
157:ENDPROCEDURE
158:PROCEDURE FOR KEYWORD_SET_UP
159:LOOP USING THE FOLLOWING DATA PAIRS
160:       ($ = POUND SIGN IN KEYWORDS BELOW)
161:   KEYWORD                     PROCEDURE NAME
162:   -------                     --------------
163:   $MARK                       SET_DATA_CHAR            * %1
164:   $STRING                     SET_STRING_CHAR          * %1
165:   $INDENT                     SET.INDENTATION          * %1
166:   $LINENUMBER                 SET.LINENUMBER           * %1
167:   $TEXT                       BOX_TEXT                 * %1
168:   $TITLE                      BOX_TEXT                 * %1
169:   $END                        END_CONTROL              * %1
170:   $DEFINE                     DEFINE_WORDS             * %1
171:   $EJECT                      EJECT_PAGE               * %1
172:   $WIDTH                      SET.PAGE_WIDTH           * %1
173:   $SEQUENCE                   CARD_SEQUENCING          * %1
174:   $TERMINATE                  BLIND_TERMINATOR         * %1
175:
176:BEGIN ITERATION
177:FORCE THE KEYWORD INTO THE TOKEN.DICTIONARY
178:STORE THE PROCEDURE NAME INTO PROGRAM.NAME OF THE ENTRY
179:ENDLOOP
180:ENDPROCEDURE
```

As output:

```
SOFTWARE DESIGN AND DOCUMENTATION LANGUAGE

(AN ILLUSTRATION OF THE APPLICATION OF SDDL USING THE)
(SDDL PROCESSOR ITSELF AS THE OBJECT OF THE EXAMPLE. )
```

TABLE OF CONTENTS PAGE I

```
                                                                      PAGE    1
LINE
 PROGRAM OBJECTIVES
     ******************************************************************
     *                                                                *
     *    THE OBJECTIVE OF SDDL IS TO PROVIDE AN EFFECTIVE COMMUNICATIONS *
     * MEDIUM TO SUPPORT THE DESIGN AND DOCUMENTATION OF COMPLEX SOFTWARE *
     * APPLICATIONS. THIS OBJECTIVE IS MET BY PROVIDING:               *
     *                                                                *
     *    (1) A DESIGN AND DOCUMENTATION LANGUAGE WITH FORMS AND SYNTAX *
     *        THAT ARE SIMPLE, UNRESTRICTIVE, AND COMMUNICATIVE        *
     *                                                                *
     *    (2) A PROCESSOR WHICH CAN CONVERT DESIGN SPECIFICATIONS INTO AN *
     *        INTELLIGIBLE, INFORMATIVE, MACHINE REPRODUCIBLE DOCUMENT *
     *                                                                *
     *    (3) METHODOLOGY FOR EFFECTIVE USE OF THE LANGUAGE AND PROCESSOR *
     *                                                                *
     *                                                                *
     ******************************************************************
   ENDPROGRAM - STMT SUPPLIED BY PROCESSOR

                                                                      PAGE    2
LINE
 PROGRAM DATA_STRUCTURE AND GLOSSARY

     INPUT.TEXT.BUFFER                   A GLOBAL CHARACTER ARRAY CONTAINING
                                         A SINGLE INPUT STATEMENT FORMED BY
                                         CONCATENATION OF CONTINUED INPUT LINES

     TEXT.LENGTH                         THE LENGTH OF THE CURRENT INPUT LINE
                                         (TRAILING BLANKS NOT INCLUDED)

     LIST: TOKEN.DICTIONARY              LINKED LIST OF DICTIONARY ENTRIES
       MEMBER ENTITY: ENTRY                POINTER TO A SINGLE DICTIONARY ENTRY
         CHARACTER.COUNT                     NUMBER OF CHARACTERS IN THE ENTRY
         TEXT.POINTER                        POINTER TO THE CHARACTER ARRAY
                                             CONTAINING THE TEXT OF THE ENTRY
         PROGRAM.NAME                        IF ENTRY IS A KEYWORD THIS IS THE
                                             LOCATION OR IDENTIFICATION OF THE
                                             ROUTINE FOR PROCESSING THE STMT
                                             VALUE=0 IF ENTRY IS NOT A KEYWORD
         LIST: REFERENCE.LIST                FIRST-IN,FIRST-OUT LIST OF
                                               REFERENCES TO THE ENTRY

           MEMBER ENTITY: "REFERENCE"
             PAGE.NUMBER
             LINE.NUMBER

      LIST: MODULE.STACK                 PUSH DOWN STACK OF NODES REPRESENTING
                                           THE NESTED STRUCTURES OF THE DESIGN

        MEMBER ENTITY: NODE                  ( IF,LOOP,PROGRAM,ETC)
          NODE.NAME
           INDENTATION.COLUMN
 ENDPROGRAM DATA_STRUCTURE
```

```
LINE                                                                    PAGE    3
  63 PROGRAM MAIN ROUTINE
  64    CALL INITIALIZATION ROUTINE---------------------------------------->(  5)
  65    LOOP UNTIL ALL INPUT DATA HAS BEEN PROCESSED
  66       CALL GET_STATEMENT ------------------------------------------ %1>(  4)
  67       •YIELD TEXT.LENGTH
  68
  69       CALL TOKEN_FINDER (FINDS THE FIRST TOKEN IN THE STATEMENT)----->(   )
  70       •YIELD TOKEN.TYPE
  71
  72       IF TOKEN.TYPE IS "IDENTIFIER"
  73          CALL ENTABLE TO FIND THE TOKEN IN THE TOKEN.DICTIONARY------>(   )
  74       ENDIF
  75
  76       IF THE TOKEN WAS FOUND AND IT IS A KEYWORD
  77          CALL KEYWORD_PROCESSOR--------------------------------------->(   )
  78       ELSE THE STATEMENT DOES NOT BEGIN WITH A KEYWORD
  79          IF THE MODULE.STACK IS EMPTY
  80             PUSH A PROGRAM MODULE ON THE MODULE.STACK
  81          ENDIF
  82          CALL SOURCE_LISTER TO SEND THE STATEMENT TO THE OUTPUT FILE->(   )
  83       ENDIF
  84
  85       FLUSH ANY "ERROR MESSAGES" TRIGGERED BY THE STATEMENT
  86    REPEAT
  87    CALL WRAP_UP------------------------------------------------------->(   )
  88 <--EXITPROGRAM
  89 ENDPROGRAM


LINE                                                                    PAGE    4
  90 PROCEDURE:  GET_STATEMENT                                             %1
  91    •USING INPUT.TEXT.BUFFER
  92    •YIELD TEXT.LENGTH
  93
  94    READ AN INPUT RECORD
  95    LOOP UNTIL A NON-BLANK RECORD IS FOUND
  96       IF THE MODULE.STACK IS NOT EMPTY (A MODULE EXISTS)
  97          PRINT THE INPUT RECORD NUMBER AND A BLANK LINE TO THE "SDD"
  98       ENDIF
  99       READ ANOTHER INPUT RECORD
 100    REPEAT
 101    COPY THE INPUT RECORD INTO THE INPUT.TEXT.BUFFER
 102    SET TEXT.LENGTH = "USABLE COLUMNS"( 80 - CARD SEQUENCE COLS)          ???
 103    LOOP
 104       FIND THE LAST NON-BLANK CHARACTER IN INPUT.TEXT.BUFFER
 105       SET TEXT.LENGTH = COLUMN NUMBER OF THE CHARACTER
 106       IF THE CHARACTER IS NOT A CONTINUATION.MARK
 107 <--------EXITPROCEDURE
 108       ENDIF
 109       SUBTRACT 1 FROM THE TEXT.LENGTH (BACK UP OVER THE CONTINUATION.MARK)
 110       IF THERE IS NO MORE DATA (END OF FILE ENCOUNTERED)
 111 <--------EXITPROCEDURE
 112       ENDIF
 113       IF THE SPACE LEFT IN INPUT.TEXT.BUFFER < 80 CHARACTERS              ???
 114          EXPAND INPUT.TEXT.BUFFER BY AT LEAST 80 CHARACTERS               ???
 115       ENDIF
 116       READ IN ANOTHER INPUT RECORD
 117       COPY THE INPUT RECORD INTO INPUT.TEXT.BUFFER BEGINNING AT TEXT.LENGTH
 118       ADD "USABLE COLUMNS" TO TEXT.LENGTH
 119    REPEAT
 120 ENDPROCEDURE
```

```
LINE
 121 PROCEDURE FOR INITIALIZATION                                          PAGE     5
 122    READ IN EXECUTION TIME OPTION FLAGS FROM EXECUTION STATEMENT
 123       OPTION.B = BREAKPOINT
 124       OPTION.C = CROSS REFERENCE
 125       OPTION.E = "ERROR MESSAGES"
 126       OPTION.K = KEYWORDS
 127       OPTION.M = MODULE CROSS REFERENCE
 128       OPTION.P = PAGE REFERENCE NUMBERS
 129       OPTION.R = REFERENCE TREE
 130       OPTION.T = TABLE OF CONTENTS
 131
 132    IF OPTION.B IS NOT SET  BREAKPOINTING IS REQUIRED
 133       READ IN REMAINDER OF EXECUTION STATEMENT
 134       IF A NAME IS SPECIFIED FOR THE SDD OUTPUT FILE
 135          SET UP A BUSE RELATIONSHIP WITH SDD
 136       ENDIF
 137       CATALOG AND ASSIGN SDD  AS THE OUTPUT FILE
 138       IF THE CATALOG STEP FAILED
 139          PRINT AN ERROR MESSAGE
 140          TERMINATE THE PROCESSOR
 141 <--------EXITPROCEDURE
 142       ENDIF
 143       BREAKPOINT THE OUTPUT TO SDD
 144    ENDIF
 145    ESTABLISH THE FOLLOWING MACHINE DEPENDENT CONSTANTS
 146      CHARACTERS.PER.WORD          = 6                                  ???
 147      BUFFER.COUNT                 = 14 (14*6=84 CHARS/LINE)            ???
 148      READ.UNIT                    = 5                                  ???
 149      WRITE.UNIT                   = 6                                  ???
 150      DEFAULT.INDENT               = 3                                  ???
 151      RIGHT.MARGIN                 = 80
 152
 153    INITIALIZE INPUT.TEXT.BUFFER TO AT LEAST 80 CHARACTERS               ???
 154    ESTABLISH TOKEN.DICTIONARY DATA STRUCTURE
 155    CALL KEYWORD_SET_UP TO ESTABLISH DEFAULT KEYWORDS---------------->(  6)
 156 <--EXITPROCEDURE
 157 ENDPROCEDURE


LINE
 158 PROCEDURE FOR KEYWORD_SET_UP                                          PAGE     6
 159    LOOP USING THE FOLLOWING DATA PAIRS
 160             (# = POUND SIGN IN KEYWORDS BELOW)
 161          KEYWORD                    PROCEDURE NAME
 162          -------                    --------------
 163          #MARK                      SET_DATA_CHAR
 164          #STRING                    SET_STRING_CHAR                   #1
 165          #INDENT                    SET_INDENTATION                   #1
 166          #LINENUMBER                SET_LINENUMBER                    #1
 167          #TEXT                      BOX_TEXT                          #1
 168          #TITLE                     BOX_TEXT                          #1
 169          #END                       END_CONTROL                       #1
 170          #DEFINE                    DEFINE_WORDS                      #1
 171          #EJECT                     EJECT_PAGE                        #1
 172          #WIDTH                     SET_PAGE_WIDTH                    #1
 173          #SEQUENCE                  CARD_SEQUENCING                   #1
 174          #TERMINATE                 BLIND_TERMINATOR                  #1
 175
 176    BEGIN ITERATION
 177       FORCE THE KEYWORD INTO THE TOKEN.DICTIONARY
 178       STORE THE PROCEDURE NAME INTO PROGRAM.NAME OF THE ENTRY
 179    ENDLOOP
 180 ENDPROCEDURE
```

```
   ••••••••••••••  MODULE REFERENCE TREE  ••••••                    PAGE     7
LN  PAGE
 1      1  OBJECTIVES

 2      2  DATA_STRUCTURE

 3      3  MAIN
 4      5  .  INITIALIZATION
 5      6  .  .  KEYWORD_SET_UP
 6      4  .  GET_STATEMENT
 7      •  .  TOKEN_FINDER
 8      •  .  ENTABLE
 9      •  .  KEYWORD_PROCESSOR
10      •  .  SOURCE_LISTER
11      •  .  WRAP_UP
```

```
                                  MODULE
                          CROSS REFERENCE LISTING                    PAGE     8
IDENTIFIER••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

DATA_STRUCTURE
     PAGE   2  PROGRAM DATA_STRUCTURE
      LINES  31, 62
ENTABLE
     PAGE   3  PROGRAM MAIN
      LINES  73
GET_STATEMENT
     PAGE   3  PROGRAM MAIN
      LINES  66
     PAGE   4  PROCEDURE:  GET_STATEMENT
      LINES  90
INITIALIZATION
     PAGE   3  PROGRAM MAIN
      LINES  64
     PAGE   5  PROCEDURE FOR INITIALIZATION
      LINES  121
KEYWORD_PROCESSOR
     PAGE   3  PROGRAM MAIN
      LINES  77
KEYWORD_SET_UP
     PAGE   5  PROCEDURE FOR INITIALIZATION
      LINES  155
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  158
MAIN
     PAGE   3  PROGRAM MAIN
      LINES  63
OBJECTIVES
     PAGE   1  PROGRAM OBJECTIVES
      LINES  16
SOURCE_LISTER
     PAGE   3  PROGRAM MAIN
      LINES  82
TOKEN_FINDER
     PAGE   3  PROGRAM MAIN
      LINES  69
WRAP_UP
     PAGE   3  PROGRAM MAIN
      LINES  87
```

```
                                DATA ITEMS
                          CROSS REFERENCE LISTING                    PAGE    9
IDENTIFIER****************************************************************

BUFFER.COUNT
     PAGE   5  PROCEDURE FOR INITIALIZATION
      LINES  147
CHARACTERS.PER.WORD
     PAGE   5  PROCEDURE FOR INITIALIZATION
      LINES  146
CHARACTER.COUNT
     PAGE   2  PROGRAM DATA_STRUCTURE
      LINES  42
CONTINUATION.MARK
     PAGE   4  PROCEDURE:  GET_STATEMENT
      LINES  106, 109
DEFAULT.INDENT
     PAGE   5  PROCEDURE FOR INITIALIZATION
      LINES  150
ERROR MESSAGES
     PAGE   3  PROGRAM MAIN
      LINES  85
     PAGE   5  PROCEDURE FOR INITIALIZATION
      LINES  125
IDENTIFIER
     PAGE   3  PROGRAM MAIN
      LINES  72
INDENTATION.COLUMN
     PAGE   2  PROGRAM DATA_STRUCTURE
      LINES  60
INPUT.TEXT.BUFFER
     PAGE   2  PROGRAM DATA_STRUCTURE
      LINES  33
     PAGE   4  PROCEDURE:  GET_STATEMENT
      LINES  91, 101, 104, 113, 114, 117
     PAGE   5  PROCEDURE FOR INITIALIZATION
      LINES  153
LINE.NUMBER
     PAGE   2  PROGRAM DATA_STRUCTURE
      LINES  53
MODULE.STACK
     PAGE   2  PROGRAM DATA_STRUCTURE
      LINES  56
     PAGE   3  PROGRAM MAIN
      LINES  79, 80
     PAGE   4  PROCEDURE:  GET_STATEMENT
      LINES  96
NODE.NAME
     PAGE   2  PROGRAM DATA_STRUCTURE
      LINES  59
OPTION.B
     PAGE   5  PROCEDURE FOR INITIALIZATION
      LINES  123, 132
OPTION.C
     PAGE   5  PROCEDURE FOR INITIALIZATION
      LINES  124
OPTION.E
```

DATA ITEMS
CROSS REFERENCE LISTING PAGE 10

IDENTIFIER

```
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  125
OPTION.K
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  126
OPTION.M
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  127
OPTION.P
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  128
OPTION.R
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  129
OPTION.T
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  130
PAGE.NUMBER
    PAGE   2  PROGRAM DATA_STRUCTURE
     LINES  52
PROGRAM.NAME
    PAGE   2  PROGRAM DATA_STRUCTURE
     LINES  45
    PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
     LINES  178
READ.UNIT
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  148
REFERENCE
    PAGE   2  PROGRAM DATA_STRUCTURE
     LINES  51
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  124, 127, 128, 129
REFERENCE.LIST
    PAGE   2  PROGRAM DATA_STRUCTURE
     LINES  49
RIGHT.MARGIN
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  151
SDD
    PAGE   4  PROCEDURE:  GET_STATEMENT
     LINES  97
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  134, 135, 137, 143
TEXT.LENGTH
    PAGE   2  PROGRAM DATA_STRUCTURE
     LINES  37
    PAGE   3  PROGRAM MAIN
     LINES  67
    PAGE   4  PROCEDURE:  GET_STATEMENT
     LINES  92, 102, 105, 109, 117, 118
TEXT.POINTER
    PAGE   2  PROGRAM DATA_STRUCTURE
```

```
                                DATA ITEMS
                            CROSS REFERENCE LISTING                    PAGE   11
IDENTIFIER****************************************************************

     LINES  43
TOKEN.DICTIONARY
    PAGE   2  PROGRAM DATA_STRUCTURE
     LINES  40
    PAGE   3  PROGRAM MAIN
     LINES  73
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  154
    PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
     LINES  177
TOKEN.TYPE
    PAGE   3  PROGRAM MAIN
     LINES  70, 72
USABLE COLUMNS
    PAGE   4  PROCEDURE:  GET_STATEMENT
     LINES  102, 118
WRITE.UNIT
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  149
```

```
                                REVISIONS
                            CROSS REFERENCE LISTING                    PAGE   12
IDENTIFIER****************************************************************

$1
    PAGE   3  PROGRAM MAIN
     LINES  66
    PAGE   4  PROCEDURE:  GET_STATEMENT
     LINES  90
    PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
     LINES  163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174
```

```
                    PROGRAM PORTABILITY CONSIDERATIONS
                            CROSS REFERENCE LISTING                    PAGE   13
IDENTIFIER****************************************************************

???
    PAGE   4  PROCEDURE:  GET_STATEMENT
     LINES  102, 113, 114
    PAGE   5  PROCEDURE FOR INITIALIZATION
     LINES  146, 147, 148, 149, 153
```

```
                                ROUTINES AND FUNCTIONS
                                CROSS REFERENCE LISTING                          PAGE   14
IDENTIFIER++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

BLIND_TERMINATOR
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  174
BOX_TEXT
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  167, 168
CARD_SEQUENCING
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  173
DEFINE_WORDS
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  170
EJECT_PAGE
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  171
END_CONTROL
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  169
SET_DATA_CHAR
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  163
SET_INDENTATION
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  165
SET_LINENUMBER
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  166
SET_PAGE_WIDTH
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  172
SET_STRING_CHAR
     PAGE   6  PROCEDURE FOR KEYWORD_SET_UP
      LINES  164
```

Example 2. Illustration of SDDL responses to sample input errors:

As input (part 1):

```
 1:*DEFINE NULL PROCEDURE, ENDPROCEDURE, EXITPROCEDURE &
 2:  SELECT, CASE  ENDSELECT ENDIF
 3:*DEFINE MODULE  FUNCTION END RETURN
 4:*DEFINE BLOCK IF ALWAYS
 5:*DEFINE BLOCK GIVEN, ENDDATA, .YIELDING
 6:*DEFINE BLOCK GIVEN,,,USING
 7:PROCEDURE TO ILLUSTRATE THE CONTINUATION CAPABILITIES FOR INPUT &
 8:OF LONG LINES & FOR OUTPUT OF LONG LINES
 9:LOOP
10:LOOP AGAIN
11:IF NOW IS THE TIME
12:DO IT AS BEST YOU CAN
13:GIVEN INPUT ARGUMENTS
14:INPUT 1
15:INPUT 2
16:USING COMMON VARIABLES
17:ITEM 2
18:ITEM 3
19:YIELDING RETURN ARGUMENTS
20:ANSWER
21:ENDDATA FOR PROCEDURE INTERFACE
22:SELECT IS NOT A KEYWORD ANYMORE
23:*INDENT 20 COLUMNS FROM NOW ON
24:IF ANSWER = AGAIN
25:CYCLE
26:ELSEIF ANSWER = STOP
27:EXITPROCEDURE
28:EXITPROGRAM
29:RETURN
30:ELSE
31:EXITLOOP
32:ALWAYS
33:IF A
34:LOOP B
35:IF C
36:LOOP D WRAPS AROUND THE LEFT MARGIN BECAUSE OF THE DEEP INDENTATION
37:*INDENT = 4
38:INDENTATION AMOUNT IS SET TO 4 BUT THE PROCESSOR WILL UNINDENT CORRECTLY
39:ENDLOOP
40:AN IF STATEMENT WILL BE CLOSED BY THE PROCESSOR
41:ENDLOOP
42:NEXT, 3 STRUCTURES ARE TERMINATED BY THE TERMINATION DIRECTIVE
43:*TERMINATE 3
44:FINALLY ENDPROGRAM CLOSES THE REMAINING OPEN BLOCKS
45:ENDPROGRAM
46:*LINENUMBER
```

As input (part 2):

```
 1:FUNCTION FOR IT
 2:GIVEN
 3:FIRST INPUT
 4:SECOND INPUT
 5:USING GLOBAL VARIABLES
 6:A
 7:B
 8:YIELDING OR RETURNING CALCULATIONS
 9:ANSWER 1
10:ANSWER 2
11:END DATA
12:LOOP UNTIL DONE
13:IF TODAY IS TUESDAY
14:THIS MUST BE BELGIUM
15:SERIOUSLY, FOLKS *NOTICE HOW THIS LINE IS SPLIT
16:IF A LINE HAS A POUND SIGN * THE PROCESSOR
17:LINES UP THE PART AFTER THE * AGAINST THE RIGHT MARGIN
18:THE REMAINDER OF THE DOCUMENT WILL BE
19:WRAPPED UP BY THE END OF FILE MARK.
```

Execution step:

```
@SDDL*SDDL.SDDL
@ADD SDDL*SDDL.INPUT1
#LINENUMBER
@ADD SDDL*SDDL.INPUT2
@FREE SDD.
@SYM SDD.,HOLD/HOLD,G9300A
```

As output:

```
LINE                                                                    PAGE     1
     PROGRAM - STATEMENT SUPPLIED BY PROCESSOR
   7    PROCEDURE TO ILLUSTRATE THE CONTINUATION CAPABILITIES FOR INPUT OF LONG
         &LINES & FOR OUTPUT OF LONG LINES
   9    LOOP
  10       LOOP AGAIN
  11          IF NOW IS THE TIME
  12             DO IT AS BEST YOU CAN---------------------------------->(  2)
  13             GIVEN INPUT ARGUMENTS
  14                INPUT 1
  15                INPUT 2
  16             USING COMMON VARIABLES
  17                ITEM 2
  18                ITEM 3
  19             YIELDING RETURN ARGUMENTS
  20                ANSWER
  21             ENDDATA FOR PROCEDURE INTERFACE
  22             SELECT IS NOT A KEYWORD ANYMORE
  24             IF ANSWER = AGAIN
  25       <--------------------------CYCLE
  26             ELSEIF ANSWER = STOP
  27                                   EXITPROCEDURE
  28 <-------------------------------EXITPROGRAM
  29                                   RETURN
***** ^                               *** ERROR *** INCORRECT MODULE ESCAPE WORD
  30             ELSE
  31       <--------------------------EXITLOOP
  32             ALWAYS
  33             IF A
  34                                   LOOP B
  35                                                    IF C
  36                                                                    LOO
     &P D WRAPS AROUND THE LEFT MARGIN BECAUSE OF THE DEEP INDENTATION
  38
     &INDENTATION AMOUNT IS SET TO 4 BUT THE PROCESSOR WILL UNINDENT CORRECTLY
  39                                                                    END
     &LOOP
  40                                                                    AN
     &IF STATEMENT WILL BE CLOSED BY THE PROCESSOR
                                                       ENDIF - STMT SUPPLIED BY
  41                                   ENDLOOP
  42                                   NEXT, 3 STRUCTURES ARE TERMINATED BY THE TE
                                        &RMINATION DIRECTIVE
  44       FINALLY ENDPROGRAM CLOSES THE REMAINING OPEN BLOCKS
        ENDLOOP - STMT SUPPLIED BY PROCESSOR
  45 ENDPROGRAM
```

```
LINE                                                              PAGE     2
   1 FUNCTION FOR IT
   2    GIVEN
   3       FIRST INPUT
   4       SECOND INPUT
   5    USING GLOBAL VARIABLES
   6       A
   7       B
   8    YIELDING OR RETURNING CALCULATIONS
   9       ANSWER 1
  10       ANSWER 2
       ENDGIVEN - STMT SUPPLIED BY PROCESSOR
  11 END DATA
```

```
LINE                                                              PAGE     3
     PROGRAM - STATEMENT SUPPLIED BY PROCESSOR
  12    LOOP UNTIL DONE
  13       IF TODAY IS TUESDAY
  14          THIS MUST BE BELGIUM
  15          SERIOUSLY, FOLKS                    NOTICE HOW THIS LINE IS SPLIT
  16          IF A LINE HAS A POUND SIGN                        THE PROCESSOR
  17             LINES UP THE PART AFTER THE          AGAINST THE RIGHT MARGIN
  18             THE REMAINDER OF THE DOCUMENT WILL BE
  19             WRAPPED UP BY THE END OF FILE MARK.
              ENDIF - STMT SUPPLIED BY PROCESSOR
          ENDIF - STMT SUPPLIED BY PROCESSOR
       ENDLOOP - STMT SUPPLIED BY PROCESSOR
    ENDPROGRAM - STMT SUPPLIED BY PROCESSOR
```

SECTION IV

USING THE SDDL PROCESSOR ON THE JPL U1108

Since SDDL usage (except for system tests and experimental runs) always involves large volumes of input and output, it is most practical to prepare the input in advance of the processing and to send the output to a printer for later, off-line perusal. In conformance with this primary operating mode, the processor has been designed to automatically handle the necessary U1108 EXEC 8 file cataloging and output stream breakpointing. The procedure for using the processor is as follows.

After the SDDL input has been loaded into one or more elements (say, QUAL*FILE.IN1 and QUAL*FILE.IN2), it is processed and printed by entering the following EXEC 8 commands:

```
@SDDL*SDDL.SDDL[,options]     [SDD-output-filename.]
@ADD QUAL*FILE.IN1
@ADD QUAL*FILE.IN2
@EOF
@FREE SDD.
@SYM SDD.,HOLD/HOLD,G9300A
```

Option	Meaning
B	Supresses catalog and Breakpoint operations. All output goes directly to the terminal.
C	Suppresses data element Cross reference tables.
E	Suppresses Error messages.
K	Suppresses generation of default Keywords (e.g., PROGRAM, IF, etc.).
M	Suppresses Module cross reference table.
P	Suppresses second pass editing operation to supply missing Page references on Module Invocation statements.
R	Suppresses module Reference tree.
T	Suppresses Table of contents.

The first command invokes the SDDL processor. The usages of all processor options shown above are consistent in that, when exercised, they all cause a feature or action of the processor to be suppressed. Thus if no options are exercised, the processor performs all its functions.

The user also has the option to supply an output file name. If none is given, the name SDD. is selected as the default name of the output file. If the user does supply a name, an EXEC 8 @USE relationship is set up equivalencing SDD. to the name supplied by the user. The

user-supplied output file name must end with a period. If it is incorrectly specified, the processor will write an error message and terminate without processing the input.

With the output file name established, the processor then performs the equivalent of the following EXEC 8 operations:

```
@ASG,A  SDD. or @ASG,UP SDD.
@BRKPT PRINT$/SDD
```

and begins processing the input stream.

When all of the input data has been processed, a second-pass, editing operation is prepared to supply the page reference numbers which were unavailable during the first pass (see P option above). The second-pass editing operation will be set up by the SDDL processor and performed independently by a Text Editor program. This step is automatic and, except for brief messages to report the state of system, is transparent to the user.

To set up the second pass, the SDDL processor writes appropriate editing commands to a scratch file (SIMU1.) and then queues the file (@ADD SIMU1.) for execution when the processor is finished.

When the second-pass editing is finished, the message PAGE REFERENCE EDITING COMPLETED will appear on the terminal, and the output may be sent to the printer with the appropriate @SYM command.

Sample execution setup:

```
@SDDL*SDDL.SDDL
@ADD QUAL*FILE.IN
@FREE SDD.
@SYM SDD.,HOLD/HOLD,G9300A
```

The SDDL processor and several user's information elements are contained in a read-only file named SDDL*SDDL. The following elements are contained in this file:

SDDL*SDDL.SDDL	Processor executable element
SDDL*SDDL.INFO	User's information memo
SDDL*SDDL.XQT	Sample execution element
SDDL*SDDL.INPUT1	Sample SDDL input element (part 1)
SDDL*SDDL.INPUT2	Sample SDDL input element (part 2)
SDDL*SDDL.USERS	Mailing list for SDDL information bulletins

BIBLIOGRAPHY

Baker, F. T., "Structured Programming in a Production Programming Environment," IEEE Trans. on Software Engr., Vol. SE-1, No. 2, pp. 241-252, June 1975.

Baker, F. T., and Mills, H. D., "Chief Programmer Teams," Datamation, Vol. 19, No. 12, pp. 58-61, Dec. 1973.

Basili, V. R., SIMPL-X, A Language for Writing Structured Programs, Nat. Tech. Info. Service Report AD755-703, U.S. Dept. of Commerce, Springfield, VA, Jan. 1973.

Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," Datamation, Vol. 19, No. 5, May 1973.

Brinch Hansen, P., "The Purpose of Concurrent Pascal," Proceedings of the 1975 International Conference on Reliable Software, IEEE Catalog No. 75 CH0940-7CSR, pp. 305-309. (Also published in SIGPLAN Notices, June 1975, pp. 305-309.)

Brinch Hansen, P., Concurrent Pascal: A Programming Language for Operating System Design, California Institute of Technology Information Science Technical Report No. 10, Pasadena, CA, April 1974.

Brooks, F. P., "The Mythical Man-Month," Datamation, Vol. 20, No. 12, pp. 45-52, Dec. 1974.

Caine, S. H., and Gordon, E. K., "PDL--A Tool for Software Design," Program Design Language Reference Guide, Caine, Farber, and Gordon, Inc., Pasadena, CA, Sept. 18, 1974.

Constantine, L. L., Fundamentals of Program Design, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

Dahl, O. J., and Hoare, C. A. R., "Hierarchical Program Structures," in Structured Programming, Academic Press, New York, 1972.

Dijkstra, E. W., "Notes on Structured Programming," in Structured Programming, Academic Press, New York, 1972 (particularly pp. 16-23).

Flynn, J., SFTRAN User's Guide, Comput. Memo. 914-337, Jet Propulsion Laboratory, Pasadena, CA, July 1973 (JPL internal document).

Hoare, C. A. R., "Notes on Data Structuring," in Structured Programming, Academic Press, New York, 1972.

Katzan, H., Jr., Advanced Programming, D. Van Nostrand Reinhold Co., NJ, 1970, pp. 153-163.

Kernighan, B. W., and Plauger, P. J., The Elements of Programming Style, McGraw-Hill Book Co. New York, 1974, pp. 36-39.

Kleine, H., and Morris, R. V., "Modern Programming: A Definition," SIGPLAN Notices, Vol. 9, No. 9, Sept. 1974, pp. 14-17.

Liskov, B., and Zilles, S., "Programming with Abstract Data Types," SIGPLAN Notices, March 1974, pp. 50-59.

Luppino, F. B., and Smith, R. L., "Programming Support Library Functional Requirements," Vol. V of Structured Programming Series, RADC-TR-74-300, U. S. Air Force, July 25, 1974.

Miller, E. F., Jr., A Compendium of Language Extensions to Support Structured Programming, RN-42, General Research Corp., Santa Barbara, CA, Jan. 1973.

Mills, H. D., "Top-Down Programming in Large Systems," in Debugging Techniques in Large Systems, Edited by R. Rustin, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1971, pp. 43-45.

Mills, H. D., Mathematical Foundations of Structured Programming, IBM Document FSC 72-6012, IBM Federal Systems Division, Gaithersburg, MD, Feb. 1972.

Myers, G. J., Composite Design: The Design of Modular Programs, Technical Report TR00.2406, IBM, Poughkeepsie, N. Y., Jan. 29, 1973.

Robert, D. C., "File Organization Techniques," Advances in Computers, Vol. 12, Academic Press, New York, 1972.

Shneiderman, B., "A Review of Design Techniques for Programs and Data," Software-Practice and Experience, Vol. 6, 1976, pp. 555-567.

Shneiderman, B., et al., "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," Communications of the ACM, Vol. 20, No. 6, June 1977, pp. 373-381.

Tausworthe, R. C., Standardized Development of Computer Software. Part 1, Methods, SP 43-29, Jet Propulsion Laboratory, Pasadena, CA, July 1976.