PRODUCT SPECIFICATION

REV LTR	REVISION ISSUE DATE	APPROVED BY	REVISIONS
E	11/17/78		Changes for the Mark VIII.0 Release (cont) 10-28 Updated <SEARCH STATEMENT> : Added ON FILE PART Deleted all other references to SEARCH PART Deleted <FILE MISSING PART> Deleted <FILE LOCKED PART> Added <ON FILE PART> 10-31 Updated table: Added PROTECTION Added PROTECTION_IO 10-39 Updated COMPILE_CARD_INFO table Added USERCODE Added FILLER Added SESSION Changed CHARGE NUMBER CHARACTER from 6 to 7 10-45 Updated MESSAGE_COUNT Deleted [<FILE IDENTIFIER> [<EXPRESSION>] Added <SWITCH FILE IDENTIFIER>
F	6/25/80		Changes for the Mark 10.0 Release 5-6 Added "<LEVEL NUMBER> <STRUCTURE ELEMENT>" to <STRUCTURE ELEMENTS>. 5-20 Added "<HOST_NAME PART>" to <FILE ATTRIBUTE> list. 5-33 Added "<HOST_NAME PART>" ATTRIBUTE. 8-16 Added "<BINARY_SEARCH DESIGNATOR>", "<DATA_LENGTH DESIGNATOR>", "<DATA_TYPE DESIGNATOR>" "<LAST_LIO STATUS DESIGNATOR>", & "<TIMER DESIGNATOR>" to "VALUE GENERATING FUNCTIONS" list. 8-18 Added "BINARY SEARCH" description. 8-22 Added "DATA LENGTH" & "DATA TYPE" descriptions. 8-28 Added "LAST LIO STATUS" description. 8-39 Added "TIMER" description. 9-2 Added " <ON BEHALF OF MODE>" to "<OPEN ATTRIBUTE>." Added "<ON BEHALF OF MODE>" to OPEN STATEMENT. 9-6 Added "<READ PART> <RESULT MASK>; <ON SEQUENCE>" to "<READ STATEMENT>." Added "<RESULT MASK> ::= WITH RESULT_MASK <ADDRESS GENERATOR>" to the READ STATEMENT. 9-7 Added "if the <RESULT MASK>..." paragraph.

PRODUCT SPECIFICATION

REV LTR	REVISION ISSUE DATE	APPROVED BY	REVISIONS

Changes for the Mark 10.0 Release (cont.)

9-8 Added "<WRITE PART> <RESULT MASK>; <ON SEQUENCE>" to the WRITE STATEMENT.

9-9 Added " <RESULT MASK> ::= WITH RESULT_MASK <ADDRESS GENERATOR>" to the WRITE STATEMENT.

9-10 Added "If the <RESULT MASK>..." paragraph.

10-15 Added "<DYNAMIC HOST_NAME PART>" and "<DYNAMIC OPEN_ON_BEHALF_OF PART>" to <DYNAMIC FILE ATTRIBUTE> list.

10-25 Added "<DYNAMIC HOST_NAME PART>" and "<DYNAMIC OPEN_ON_BEHALF_OF>" descriptions.

10-36 Added "<REFER_ADDRESS DESIGNATOR>", "<REFER_LENGTH DESIGNATOR>" and "<REFER_TYPE DESIGNATOR>" as FUNCTION DESIGNATORS.

10-47 Added "REFER ADDRESS" description.

10-48 Added "REFER LENGTH" and "REFER TYPE" descriptions.

Burroughs Corporation

COMPUTER SYSTEMS GROUP
SANTA BARBARA PLANT

2212 5405
B1800/B1700 SDL (BNF Version)

PRODUCT SPECIFICATION

REV LTR	REVISION ISSUE DATE	APPROVED BY	REVISIONS
E	11/17/78		Changes to the Mark VIII.0 Release Changed title to B1800/B1700 SDL (BNF Version) 1-2 Changed BNF statement <IDENTIFIER> ::= <IDENTIFIER> to <IDENTIFIER> ::= <LETTER> 2-5 Replaced "/" with "|" 3-1 Updated STRUCTURE OF AN SDL PROGRAM Section: Added |<RECORD STATEMENT> to <DECLARATION STATEMENT> 5-5 Replaced 3 NVS BIT(1) with 3 NSR BIT(1) in PL/I-STYLE STRUCTURE 5-8 Updated NON-STRUCTURE DECLARATIONS BNF; Replaced <DECLARED PART> with ...| <DECLARED PART> |... in <DECLARED ELEMENT> declaration. 5-18 Updated REFERENCE DECLARATION: Replaced <DECLARED REF> REFERENCE with ...| <DECLARED REF> REFERENCE |... Updated REFERENCE RECORD DECLARATION: Replaced <DECLARED RECORD REF> REFERENCE with ...| <DECLARED RECORD REF> REFERENCE |... in <DECLARE ELEMENT> DECLARATION. 5-19 Updated FILE DECLARATIONS: Added | <PROTECTION PART> and | <PROTECTION_IO PART> to <FILE ATTRIBUTE> 5-21 Updated Syntax Deleted |READER_PUNCH <DEVICE OPTION> from <DEVICE SPECIFIER> Added DATA_RECORDER_80 to <DEVICE SPECIFIER> 5-22 Updated Format Deleted READER_PUNCH Added DATA_RECORDER_80 5-25 Updated Default section of UNBLOCKED RECORD LENGTHS 5-31 to 5-32 Added Default status of <PROTECTION PART> attribute and <PROTECTION_IO_PART> 6-2 Updated PROCEDURE HEAD: Added REFERENCE TO <TYPE PART> 7-1 Updated ASSIGNMENT STATEMENTS AND EXPRESSIONS: Deleted EXPRESSION from ASSIGNMENT STATEMENT Added EXPRESSION LIST to ASSIGNMENT STATEMENT 8-12 Description of NULL rewritten 9-12 ACCEPT STATEMENT section updated: Deleted <END-OF-TEXT SPECIFIER> Deleted paragraph pertaining to END-OF-TEXT 10-28 Updated SEARCH_DIRECTORY STATEMENT:

TABLE OF CONTENTS

BACKUS NAUR FORM 1-1
RELATED PUBLICATIONS 1-2
BASIC COMPONENTS OF THE SDL LANGUAGE 2-1
COMMENTS 2-2
NUMBERS 2-3
BIT STRINGS 2-3
CHARACTER STRINGS 2-4
CHAR_TABLE 2-5
OTHER CONSTANTS 2-5
STRUCTURE OF AN SDL PROGRAM 3-1
PROGRAM SEGMENTATION 4-1
DECLARATIONS 5-1
DATA TYPES 5-1
DECLARE STATEMENT 5-2
RECORD STATEMENT 5-3
NON-STRUCTURE DECLARATIONS 5-8
STRUCTURE DECLARATIONS 5-11
PAGED ARRAY DECLARATIONS 5-15
DYNAMIC DECLARATIONS 5-16
Restrictions: 5-17
REFERENCE DECLARATIONS 5-18
RECORD REFERENCE DECLARATIONS 5-19
FILE DECLARATIONS 5-20
SWITCH FILE DECLARATIONS 5-34
DEFINE STATEMENT 5-36
FORWARD DECLARATION 5-40
USE STATEMENT 5-43
PROCEDURES 6-1
PROCEDURE HEAD 6-2
INTRINSIC HEAD 6-5
PROCEDURE BODY 6-6
PROCEDURE ENDING 6-8
ASSIGNMENT STATEMENTS AND EXPRESSIONS 7-1
UNARY OPERATORS 7-5
ARITHMETIC OPERATORS 7-5
RELATIONAL OPERATORS 7-6
LOGICAL OPERATORS 7-7
REPLACE OPERATORS 7-8
CONCATENATION 7-11
PRIMARY ELEMENTS OF THE EXPRESSION 8-1
CONDITIONAL EXPRESSION 8-1
CASE EXPRESSION 8-2
BUMP 8-2
DECREMENT 8-4
ASSIGNOR 8-4
ADDRESS VARIABLES 8-5
INDEXING 8-5
ADDRESS GENERATING FUNCTIONS 8-8

SUBBIT AND SUBSTR 8-8
FETCH_COMMUNICATE_MSG_PTR 8-10
DESCRIPTORS 8-10
MAKE_DESCRIPTOR 8-11
NEXT_ITEM, PREVIOUS_ITEM 8-12
NULL 8-12
ADDRESS GENERATORS 8-13
VALUE VARIABLES 8-14
TYPED PROCEDURES 8-15
ADDRESS AND VALUE PARAMETERS 8-15
VALUE GENERATING FUNCTIONS 8-16
BASE_REGISTER 8-17
BINARY CONVERSION 8-18
BINARY SEARCH 8-18
COMMUNICATE_WITH_GISMO 8-19
CONSOLE_SWITCHES 8-19
CONTROL_STACK_BITS 8-19
CONTROL_STACK_TOP 8-19
CONVERT 8-20
DATA_ADDRESS 8-22
DATA_LENGTH 8-22
DATA_TYPE 8-22
DATE 8-22
DECIMAL CONVERSION 8-23
DELIMITED_TOKEN 8-23
DISPATCH 8-24
DISPLAY_BASE 8-25
DYNAMIC_MEMORY_BASE 8-25
EVALUATION_STACK_TOP 8-25
EXECUTE 8-25
EXTENDED ARITHMETIC FUNCTIONS 8-27
HASH_CODE 8-27
INTERROGATE_INTERRUPT_STATUS 8-28
LAST_LIO_STATUS 8-28
LENGTH 8-28
LIMIT_REGISTER 8-26
LOCATION 8-29
NAME_OF_DAY 8-29
NAME_STACK_TOP 8-30
NEXT_TOKEN 8-30
PARITY_ADDRESS 8-31
PROCESSOR_TIME 8-31
PROGRAM SWITCHES 8-31
SEARCH_LINKED_LIST 8-32
SEARCH_SOL_STACKS 8-33
SEARCH_SERIAL_LIST 8-34
S_MEM_SIZE, M_MEM_SIZE 8-35
SORT_SEARCH 8-35
SORT_STEP_DOWN 8-35
SORT_UNBLOCK 8-36
SPO_INPUT_PRESENT 8-36
SUBBIT AND SUBSTR 8-37
SWAP 8-38
TIME 8-39

TIMER 8-39
VALUE_DESCRIPTOR 8-40
WAIT 8-40
I/O CONTROL STATEMENTS 9-1
OPEN STATEMENT 9-2
CLOSE STATEMENT 9-4
READ STATEMENT 9-6
WRITE STATEMENT 9-8
SEEK STATEMENT 9-11
ACCEPT STATEMENT 9-12
DISPLAY STATEMENT 9-13
SPACE STATEMENT 9-14
SKIP STATEMENT 9-15
ON SEQUENCE 9-16
EXECUTABLE STATEMENTS 10-1
DO GROUPS 10-2
UNDO 10-4
IF STATEMENT 10-5
CASE STATEMENT 10-7
REFER STATEMENT 10-8
REDUCE STATEMENT 10-9
END OF STRING 10-11
MODIFY STATEMENTS (CLEAR, BUMP, DECREMENT) 10-12
NULL STATEMENT 10-13
FILE ATTRIBUTE STATEMENT (CHANGE STATEMENT) 10-14
STOP STATEMENT 10-26
ZIP STATEMENT 10-27
SEARCH_DIRECTORY STATEMENT 10-28
READ_FILE_HEADER, WRITE_FILE_HEADER 10-30
MAKE_READ_ONLY, MAKE_READ_WRITE 10-32
COROUTINE STATEMENT 10-33
EXECUTE-PROCEDURE STATEMENT 10-35
EXECUTE-FUNCTION STATEMENT 10-36
ACCESS_FILE_INFORMATION 10-37
CHANGE_STACK_SIZES 10-37
CHARACTER_FILL 10-38
COMMUNICATE 10-38
COMPILE_CARD_INFO 10-38
DC_INITIATE_IO 10-39
DEBLANK 10-40
DISABLE_INTERRUPTS 10-40
DUMP 10-40
DUMP_FOR_ANALYSIS 10-41
ENABLE_INTERRUPTS 10-41
ERROR_COMMUNICATE 10-41
EXECUTE 10-42
FETCH 10-42
FREEZE_PROGRAM 10-43
GROW 10-43
HALT 10-44
HARDWARE_MONITOR 10-44
INITIALIZE_VECTOR 10-44
MESSAGE_COUNT 10-45
MONITOR 10-45

OVERLAY 10-45
READ_CASSETTE 10-46
READ_FPB, WRITE_FPB 10-46
READ_OVERLAY, WRITE_OVERLAY 10-47
REFER ADDRESS 10-47
REFER LENGTH 10-48
REFER TYPE 10-48
REINSTATE 10-48
RESTORE 10-49
REVERSE_STORE 10-49
SAVE 10-50
SAVE_STATE 10-50
SORT 10-50
SORT_MERGE 10-51
SORT_SWAP 10-52
THAW_PROGRAM 10-52
THREAD_VECTOR 10-52
TRACE 10-53
TRANSLATE 10-54
APPENDIX I: RESERVED AND SPECIAL WORDS 11-1
APPENDIX II: SDL CONTROL CARD OPTIONS 12-1
APPENDIX III: PROGRAMMING OPTIMIZATION 13-1
APPENDIX IV: RUNNING THE COMPILER 14-1
APPENDIX V: CONDITIONAL COMPILATION 15-1
APPENDIX VI: SDL PROGRAMMING TECHNIQUES 16-1
APPENDIX VII: SDL PARTIAL RECOMPILATION FACILITY 17-1
APPENDIX VIII: SDL MONITORING FACILITY 18-1

BURROUGHS CORPORATION
COMPUTER SYSTEMS GROUP
SANTA BARBARA PLANT

BACKUS NAUR FORM

A language used to talk about a language is a metalanguage. The natural languages are, in fact, metalanguages; for example, the metalanguage English is used to talk about the structure of an English language sentence. Backus Naur Form (BNF), a metalanguage popularized by its use to describe the syntax of Algol 60 is used to describe the syntax of SDL. To avoid the confusion between the symbols of the metalanguage and those of the language being described, BNF uses only 4 metalinguistic symbols. Literal occurrences of symbols other than the the metasymbols, with no bracketing characters, represent themselves as terminal symbols of the language.

A grammar for SDL is written as a set of BNF statements, each of which has a left part, followed by the metasymbol "::=" followed by a list of right parts. The left part is a phrase name, and the right parts, separated by the metasymbol "|", are strings containing terminal symbols and/or phrase names.

METASYMBOL	ENGLISH EQUIVALENT	USE
::=	is defined as	separates a phrase name from its definition.
|	or	separates alternate definition of a phrase.
<IDENTIFIER>	"IDENTIFIER"	The bracketing characters indicate that the intervening characters are to be treated as a unit, i.e., as a phrase name.

Each BNF statement is a rewriting rule, such that we may substitute any right part for any occurrence of its associated left part; and we have a choice of right parts which we may substitute. The following example specifies the use of these rules to determine those strings which are grammatically correct identifiers in SDL.

```
<LETTER> ::=    A | B | C | D | E | F | G | H | I | J | K | L | M
              | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
              | a | b | c | d | e | f | g | h | i | j | k | l | m
              | n | o | p | q | r | s | t | u | v | w | x | y | z
```

```
<DIGIT> ::=        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<BREAK> ::=        _

<IDENTIFIER> ::=   <LETTER>
                 | <IDENTIFIER> <LETTER>
                 | <IDENTIFIER> <DIGIT>
                 | <IDENTIFIER> <BREAK>
```

XYZ12_B4 is a proper SDL <IDENTIFIER> since it can be generated as a terminating set of symbols by using the BNF rules.

Proof that XYZ12_B4 is an <IDENTIFIER> by starting with the fact that an <IDENTIFIER> can be a <LETTER>.

FORM	EXAMPLE
<IDENTIFIER> ::= <LETTER>	X
<IDENTIFIER> ::= <IDENTIFIER><LETTER>	XY
<IDENTIFIER> ::= <IDENTIFIER><LETTER>	XYZ
<IDENTIFIER> ::= <IDENTIFIER><DIGIT>	XYZ1
<IDENTIFIER> ::= <IDENTIFIER><DIGIT>	XYZ12
<IDENTIFIER> ::= <IDENTIFIER><BREAK>	XYZ12_
<IDENTIFIER> ::= <IDENTIFIER><LETTER>	XYZ12_B
<IDENTIFIER> ::= <IDENTIFIER><DIGIT>	XYZ12_B4

Notice that the BNF rules do not, in any way, limit the number of letters, digits, and dots which comprise the <IDENTIFIER>. In such cases, further semantic rules will be specified; e.g., an SDL <IDENTIFIER> is limited to a maximum of 63 characters.

RELATED PUBLICATIONS

NAME	NUMBER
SDL/UPL COMPILER	P.S. 2212 5389%
B1700 SDL S-LANGUAGE	P.S. 2201 2389
B1700 SYSTEMS REFERENCE MANUAL	#1057155

BURROUGHS CORPORATION
COMPUTER SYSTEMS GROUP
SANTA BARBARA PLANT

BASIC COMPONENTS OF THE SDL LANGUAGE

In order to understand SDL grammar, the user should be familiar with the most basic elements of the Software Developmental Language below.

<DIGIT> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<LETTER> ::= A | B | C | D | E | F | G | H | I | J
| K | L | M | N | O | P | Q | R | S | T
| U | V | W | X | Y | Z | a | b | c | d
| e | f | g | h | i | j | k | l | m | n
| o | p | q | r | s | t | u | v | w | x
| y | z

<SPECIAL CHARACTER> ::= & | . | < | ; | , | / | /= |
| $ | : | > | >= | = | + | * |
| (|) | - | <= | [|] | <BLANK>

<BREAK> ::= _

<BLANK> ::=

NOTE: <BLANK> is the occurrence of one non-visible character " ".

IDENTIFIERS

<IDENTIFIER> ::= <LETTER> | <IDENTIFIER> <LETTER>
| <IDENTIFIER> <DIGIT>
| <IDENTIFIER> <BREAK>

RESTRICTIONS:

1. An identifier may not contain blanks.
2. An identifier may contain a maximum of 63 characters.
3. Reserved words may not be used as identifiers.
4. "Special" words may be used for segment and DO-group identifiers without losing their special significance in SDL.

5. In all other cases, "special" words may be used as identifiers, however, they lose their special significance throughout the entire program when declared at Lexic Level 0. When declared at any greater lexic level, they only lose their special meaning within the procedure in which they are declared.

   (Also see "Structure of an SDL Program" and "Appendix I")

6. All reserved and special words must be in all upper case.

7. Identifiers must contain exactly the same letters, where upper and lower case are concerned, to be identical. If an upper-case identifier, for example, is entered in lower case, it is a new identifier.

COMMENTS

```
<COMMENT STRING> ::=        /*  <COMMENT TEXT>  */
```

RESTRICTIONS:

1. The pair /* preceding the <COMMENT TEXT> must appear as adjacent symbols. Similarily, the pair */ following the <COMMENT TEXT> must also appear as adjacent symbols.

```
<COMMENT TEXT> ::=              <EMPTY>
                              | <COMMENT TEXT CHARACTER>
                              | <COMMENT TEXT CHARACTER>
                                <COMMENT TEXT>

<EMPTY> ::=
```

Note: <EMPTY> is the null set or the occurrence of nothing.

```
<COMMENT TEXT
CHARACTER> ::=                  <DIGIT>
                              | <LETTER>
                              | <SPECIAL CHARACTER>
                              |  " | @ | # | %

<CARD TERMINATOR> ::=           %
```

RESTRICTION: A % is treated as any other string character if it is contained within a <CHARACTER STRING> or in <COMMENT TEXT>. However, in all other cases, a % will cause the scanning of the current source image to terminate and to continue in the next source image.

NUMBERS

```
<NUMBER> ::=                    <DIGIT> | <NUMBER> <DIGIT>
```

NOTE: Range of signed numbers -(2 exp 23) to (2 exp 23)-1.
Range of unsigned numbers 0 to (2 exp 24)-1.

BIT STRINGS

```
<BINARY DIGIT> ::=              0 | 1 | <COMMENT STRING>

<BINARY DIGITS> ::=             <BINARY DIGIT>
                              | <BINARY DIGITS> <BINARY DIGIT>

<QUARTAL DIGIT> ::=             <BINARY DIGIT> | 2 | 3

<QUARTAL DIGITS> ::=            <QUARTAL DIGIT>
                              | <QUARTAL DIGITS> <QUARTAL DIGIT>

<OCTAL DIGIT> ::=               <QUARTAL DIGIT> | 4 | 5 | 6 | 7

<OCTAL DIGITS> ::=              <OCTAL DIGIT>
                              | <OCTAL DIGITS> <OCTAL DIGIT>

<HEX DIGIT> ::=                 <OCTAL DIGIT>
                              | 8 | 9 | A | B | C | D | E | F

<HEX DIGITS> ::=                <HEX DIGIT>
                              | <HEX DIGITS> <HEX DIGIT>

<BIT GROUP>::=                  (4) <HEX DIGITS>
                              | (3) <OCTAL DIGITS>
                              | (2) <QUARTAL DIGITS>
                              | (1) <BINARY DIGITS>

<BITS>::=                       <BIT GROUP> | <HEX DIGITS>
                              | <BITS> <BIT GROUP>
                              | <EMPTY>

<BIT STRING> ::=                @<BITS>@
```

RESTRICTIONS:

1. If no bit mode is specified (i.e., The indicator digit in parentheses is omitted), "Hex" is assumed. This can only be assumed if the bit string does not start with a mode indicator; when the mode is switched to "Hex", an explicit "(4)" is required.

2. As noted above, a <COMMENT STRING> may appear anywhere within a <BIT STRING>, but not within the parentheses bounding the indicator digit. The presence of a <COMMENT STRING> will, in no way, alter the value of the <BIT STRING> containing it. Blanks may not appear in a <BIT STRING>.

Example:
```
B(3)6330316260/* THIS */313230/* IS */63302560/* THE */
4321626360/* LAST */312523465124/* RECORD */B
```

```
<STRING> ::=          <CHARACTER STRING>
                    | <BIT STRING>
```

CHARACTER STRINGS

```
<CHARACTER STRING> ::=        "<STRING CHARACTER LIST>"

<STRING CHARACTER LIST> ::=   <EMPTY>
                            | <STRING CHARACTER LIST>
                              <STRING CHARACTER>

<STRING CHARACTER> ::=        <DIGIT> | <LETTER> | <SPECIAL CHARACTER>
                            | "" | @ | # | %
```

RESTRICTIONS: If a quote sign is desired in a character string, then two adjacent quote signs must appear in the text.

EXAMPLE:
```
DECLARE  STRING CHARACTER (6),
         QUOTE CHARACTER (1);

         STRING := "AB""CDE";
         QUOTE  := """";
```

After execution, STRING will contain: AB"CDE, and QUOTE will contain: ".

Note: A <CHARACTER STRING> may contain a maximum of 256 characters.

CHAR_TABLE

The translation bit table for the set-membership reduction is rather cumbersome to construct by hand, so the compiler provides a convenient notation for table constructs. These constants are written:

```
<TABLE CONSTANT> ::=      CHAR_TABLE ( <TABLE STRING> )
<TABLE STRING> ::=        <STRING> | <TABLE STRING> CAT <STRING>
```

The constant denoted is a 256-bit string with a(1)1a corresponding to every character in <TABLE STRING>. (When a <BIT STRING> occurs in the <TABLE STRING>, it is used to denote non-graphic characters in their hexidecimal (EBCDIC) form.)

OTHER CONSTANTS

```
<CONSTANT> ::=          <NUMBER> | <STRING> | TODAYS_DATE
                      | SEQUENCE_NUMBER
                      | HEX_SEQUENCE_NUMBER
                      | <TABLE CONSTANT>
```

TODAYS_DATE represents the date and time of compilation of the program. It is the same as the date and time appearing at the top of the program listing. It is a character string with the following format --

"MM/DD/YY HH:MM"

SEQUENCE_NUMBER represents a <CHARACTER STRING> of 8 characters which is the sequence number of the current source image being compiled.

HEX_SEQUENCE_NUMBER represents a bit string of 8 (hex) digits which is the sequence number of the current source image line being compiled. If this sequence field is blank, then HEX_SEQUENCE_NUMBER = @00000000@

If the current source image line sequence number is 12753000, then on this line:

```
SEQUENCE_NUMBER = "12753000"
HEX_SEQUENCE_NUMBER = @12753000@
```

STRUCTURE OF AN SDL PROGRAM

```
<PROGRAM> ::=                       <DECLARATION STATEMENT LIST>
                                    <PROCEDURE STATEMENT LIST>
                                    <EXECUTABLE STATEMENT LIST>
                                    FINI

<DECLARATION STATEMENT
LIST> ::=                           <EMPTY>
                                  | <DECLARATION STATEMENT>
                                    <DECLARATION STATEMENT LIST>

<DECLARATION STATEMENT> ::=         <DECLARE STATEMENT>;
                                  | <DEFINE STATEMENT>;
                                  | <FILE DECLARATION STATEMENT>;
                                  | <SWITCH FILE DECLARATION
                                    STATEMENT>;
                                  | <FORWARD DECLARATION>;
                                  | <USE STATEMENT>;
                                  | <SEGMENT STATEMENT>;
                                  | <DECLARATION STATEMENT>;
                                  | <RECORD STATEMENT>;

<PROCEDURE STATEMENT
LIST> ::=                           <EMPTY>
                                  | <PROCEDURE STATEMENT>;
                                    <PROCEDURE STATEMENT LIST>

<PROCEDURE STATEMENT> ::=           <PROCEDURE DEFINITION>
                                  | <SEGMENT STATEMENT>
                                    <PROCEDURE STATEMENT>

<EXECUTABLE STATEMENT
LIST> ::=                           <EMPTY>
                                  | <EXECUTABLE STATEMENT>
                                    <EXECUTABLE STATEMENT LIST>

<EXECUTABLE STATEMENT> ::=          See SECTION 10.
```

A program written in SDL must follow the sequential structure described in the syntax above. That is, the executable section of the program may not appear until all procedures have been defined, and procedures may not be defined before the formats of data items (variables, arrays, etc.) have been declared. "FINI" is not required, but if present must physically occur as the final statement in the program.

The procedure statement (including declaration, procedure, and executable statements) is the basic structure in SDL. An SDL program is a collection of procedures, each of which can be described for conceptual purposes as a microcosm of the program. Any given procedure may contain a collection of other procedures within itself. This process is known as "Nesting".

The "Lexicographic Level" of any statement in the program is equal to the number of procedures in which it is nested. The program itself will always be Lexic Level 0, and no procedure may have a lexic level greater than 15. The diagram in Figure 1 illustrates procedure nesting and lexic levels.

It is important to understand the relationships between these nested procedures. As Figure 1. indicates, the name of any given procedure is contained in the procedure in which it is nested at the next lower lexic level. For example, procedure D is a Lexic Level 2 procedure, however, its name, "D", is part of Lexic Level 1.

The "scope" of any given procedure is recursively defined as:

1) The procedure itself,

2) Any procedure(s) nested within the procedure,

3) Any procedure (and its nested procedures) whose name appears at the same lexic level and within the same procedure as its own name, and

4) The procedure in which its own name is defined.

In Figure 1, one can see that the scope of Procedure B includes:

1) Itself, i.e., Procedure B

2) The nested procedures within B (C and D),

3) The other procedures defined at LL0: E (and its nested procedures F and G) and procedure H (and its nested procedures J, K, L, M, N, and P.

4) The procedure which defines B, in this case, the program A.

Note: All the Lexic Level 0 procedures have scope to each other. This occurs because of rule 4 above, wherein the program itself is thought to be a "procedure".

In the same manner, the scope of procedure J includes J, K, L, M, N, P, and H.

By understanding the relationships between the various procedures, it is possible to determine which procedures may be invoked by any given procedure. SDL has been defined so that any procedure X may call or invoke any procedure Y, if the scope of Y encompasses X.

In Figure 1, Procedure J may call procedures J,K,L,M,H,E, and B because each of these contains J in its scope.

Note: J cannot call the program A since the name of the program, if there is one, exists outside the program and is, therefore, not compiled; however, J may access the data contained in A (i.e., A1, A2, A3, and A4).

Figure 2 below shows the relationship between scope and calling ability for program A.

```
PROGRAM A
DECLARE A1, A2, A3, A4;
PROCEDURE B;
          DECLARE B1, B2, B3;
          PROCEDURE C;
                    DECLARE C1, C2, C3;
                    EXECUTABLE STATEMENTS;
          END C;
          PROCEDURE D;
                    EXECUTABLE STATEMENTS;
          END D;
          EXECUTABLE STATEMENTS;
END B;
PROCEDURE E;
          DECLARE E1, E2;
          PROCEDURE F;
                    DECLARE F1, F2, F3;
                    EXECUTABLE STATEMENTS;
          END F;
          PROCEDURE G;
                    DECLARE G1, G2;
                    EXECUTABLE STATEMENTS;
          END G;
          EXECUTABLE STATEMENTS;
END E;
PROCEDURE H;
          DECLARE H1, H2, H3, H4;
          PROCEDURE J;
                    PROCEDURE K;
                    END K;
                    PROCEDURE L;
                    END L;
          END;
          PROCEDURE M;
                    PROCEDURE N;
                    END N;
                    PROCEDURE P;
                    END P;
          END M;
END A;
EXECUTABLE STATEMENTS;
FINI
```

Fig 1. Procedure Nesting

CALLING PROCEDURES

Procedure	Scope	A	B	C	D	E	F	G	H	J	K	L	M	N	P
		B	*	*	*	*	*	*	*	*	*	*	*	*	*	*
		C		*	*	*									
		D		*	*	*									
		E	*	*	*	*	*	*	*	*	*	*	*	*	*	*
		F					*	*	*							
		G					*	*	*							
		H	*	*	*	*	*	*	*	*	*	*	*	*	*	*
		J								*	*	*	*	*	*	*
		K									*	*	*			
		L									*	*	*			
		M								*	*	*	*	*	*	*
		N												*	*	*
		P												*	*	*

Note: To find the scope of a procedure, find the procedure in the column of procedure names. The horizontal rows to the right indicate the procedures in its scope. The procedures which may be called by a given procedure are marked in the vertical columns below that calling procedure.

Fig 2. Scope and Calling Ability

PROGRAM SEGMENTATION

```
<SEGMENT STATEMENT> ::= <SEGMENT STATEMENT WORD> (<SEGMENT PART>);
<SEGMENT STATEMENT WORD> ::= SEGMENT | SEGMENT_PAGE
<SEGMENT PART> ::= <SEGMENT IDENTIFIER> <PAGE PART> <IMPORTANT PART> |
                   <SEGMENT IDENTIFIER> <IMPORTANT PART> <PAGE PART>
<SEGMENT IDENTIFIER> ::= <IDENTIFIER>
<PAGE PART> ::= <EMPTY> | OF <PAGE IDENTIFIER>
<PAGE IDENTIFIER> ::= <IDENTIFIER>
<IMPORTANT PART> ::= <EMPTY> | , IMPORTANT
```

As the BNF indicates, the <SEGMENT STATEMENT> may occur anywhere within an SDL program. Its purpose is to reduce the memory requirement of the program by allowing segments to overlay each other.

There is a maximum of 16 pages with 64 segments per page. The segment names represent a page-number segment-number pair.

It is only necessary to specify SEGMENT_PAGE once for each page. Every subsequent segment will be compiled to that page until another SEGMENT_PAGE is encountered.

If there are no SEGMENT_PAGE specifications, all segments will be compiled to Page Zero, and there may be no more than 64 segments total. If a program is to be segmented, the first statement must be a <SEGMENT STATEMENT>. Otherwise a warning message will appear in the source listing.

There are two types of segmentation: "permanent" and "temporary". Every statement following a permanent <SEGMENT STATEMENT> will be compiled to that segment until another <SEGMENT STATEMENT> is read. Non-consecutive statements may be compiled to the same segment by using the same <SEGMENT IDENTIFIER>. Note, however, that <DO GROUP>s (See "DO GROUPS") and procedures must end in the same segment in which they begin. If this is not the case, the compiler issues a warning and inserts code to bring the program back to the proper segment so that the do-group or procedure may be exited correctly.

The following example illustrates the use of the "permanent" <SEGMENT STATEMENT>.

```
SEGMENT (XX);
DECLARE A1, A2, A3, A4;
PROCEDURE B;
     DECLARE B1, B2, B3;
     SEGMENT (YY);
     PROCEDURE C;
          .
          .
          .
     END C;
     PROCEDURE D;
          .
          .
          .
     END D;
SEGMENT (XX);
     .
     .
     .
END B;
     .
     .
     .
FINI
```

Only procedures C and D have been compiled to the segment "YY". Segment "XX" is segment zero and includes everything else.

A <SEGMENT STATEMENT> is treated as "temporary" only when it precedes a "Subordinate Executable Statement" within any of the following statements:

<ACCESS FILE HEADER STATEMENT>	<SEARCH DIRECTORY STATEMENT>
<CASE STATEMENT>	<SEND STATEMENT>
<IF STATEMENT>	<SPACE STATEMENT>
<READ STATEMENT>	<WRITE STATEMENT>
<RECEIVE STATEMENT>	<OPEN STATEMENT>

In these specific cases, the segment change applies only to the subordinate statement following it. For example, the syntax for the <IF STATEMENT> could be written as follows:

```
<IF STATEMENT> ::=     IF <EXPRESSION>
                       THEN <SUBORDINATE EXECUTABLE STATEMENT>
                     | IF <EXPRESSION>
                       THEN <SUBORDINATE EXECUTABLE STATEMENT>
                       ELSE <SUBORDINATE EXECUTABLE STATEMENT>
```

The segmentation of a hypothetical <IF STATEMENT> is presented below to illustrate the use of a "temporary" <SEGMENT STATEMENT>.

```
SEGMENT (A);
PROCEDURE X;
      .
      .
      .
      IF Y>Z THEN Y:=Z; ELSE
      SEGMENT (B);
      DO SOME_FUNCTION;        *
            .                  *
            .                  *
            .                  *
      END SOME_FUNCTION;       *
            .
            .
            .
END X;
```

* Compiled to Segment (B)

Because the <DO GROUP>, "SOME_FUNCTION", is a subordinate <EXECUTABLE STATEMENT> in the <IF STATEMENT>, Segment (B) automatically ends when the <DO GROUP> is terminated. All statements following are compiled to Segment (A).

Notice the distinction between Segment (A), a "permanent" <SEGMENT STATEMENT>, and Segment (B), a "temporary" one.

If the construct ,IMPORTANT appears in the <IMPORTANT PART> of a segment statement, then the SDL/UPL compiler will set the decay factor for that segment to seven. If the control option word SIZE is used, a list of segment names, numbers and sizes will be printed at the end of the source listing. The segments that have been marked ,IMPORTANT will be noted.

EXAMPLES:

```
SEGMENT (SEGZERO , IMPORTANT) ;
SEGMENT_PAGE (SEGONE OF PAGEZERO , IMPORTANT);
SEGMENT (SEGTWO , IMPORTANT OF PAGEONE);
```

PRAGMATICS

The decay factor field in the segment dictionary is three bits long. It will always have a value of zero or seven. Whatever value the compiler puts in the code file, the MCP changes it. So when reading a memory dump, a value of zero means that the memory priority will decay more slowly. But when looking at code files, a value of seven means that the memory priority will decay more slowly.

DECLARATIONS

DATA TYPES

Three main types of data may be declared in SDL:

1) BIT
2) CHARACTER
3) FIXED

A bit field consists of a number of bits specified by a number in parentheses following the reserved word "BIT". The field may be a maximum of 65,535 bits.

A character field is a number of characters, 8 bits each, specified by a number in parentheses following the reserved word "CHARACTER". The field may be a maximum of 8,191 characters.

A fixed data field is a 24-bit, signed numeric field where the high order bit is interpreted as the sign. Negative numbers are represented in 2-s complement form.

The range of signed numbers (i.e., fixed data fields) is -(2 exp 23) to (2 exp 23)-1. The range of unsigned numbers (bit data fields) is 0 to (2 exp 24)-1. Bit fields, as noted above, are not restricted to 24 bits. However, for arithmetic purposes, only the low-order 24 bits will be considered except in the case of the extended arithmetic function.

DECLARE STATEMENT

```
<DECLARE STATEMENT> ::=     DECLARE <DECLARE ELEMENT>
                          | <DECLARE STATEMENT>, <DECLARE ELEMENT>

<DECLARE ELEMENT> ::=       <DECLARED PART>
                            <TYPE PART>
                          | <STRUCTURE LEVEL NUMBER>
                            <STRUCTURE DECLARED PART>
                            <STRUCTURE TYPE PART>
                          | PAGED <ELEMENTS-PER-PAGE PART>
                            <ARRAY IDENTIFIER> <ARRAY BOUND>
                            <TYPE PART>
                          | DYNAMIC <COMPLEX DYNAMIC>
                            <DYNAMIC TYPE PART>
                          | <DECLARED REF> REFERENCE
                          | <DECLARED RECORD REF> REFERENCE
```

The <DECLARE STATEMENT> specifies the addresses and characteristics of contents of memory storage areas.

Any number of <DECLARE ELEMENT>s may be declared in one <DECLARE STATEMENT>, and must be separated by commas. Best code is generated if all elements are declared within one <DECLARE STATEMENT>. (See Appendix VI).

The maximum number of data elements (including fillers, dummys, and implicit fillers) contained in one structure varies as to the compiler being used, (currently: 50 - small version, 75 - large version). Any attempt to declare more will cause a table overflow error to be detected at compile time.

An array may have a maximum of 65,535 elements, each being a maximum of 65,535 bits (8,191 characters).

The five types of <DECLARE ELEMENT>s are each discussed below.

RECORD STATEMENT

```
<RECORD STATEMENT> ::=          RECORD <RECORD IDENTIFIER>
                                <FIELD LIST>

<RECORD IDENTIFIER> ::=         <IDENTIFIER>

<FIELD LIST> ::=                <FIELD ELEMENT> |
                                <FIELD LIST>, <FIELD ELEMENT>
                              | [<COSPATIAL FIELD LIST>]
                              | <FIELD LIST>, [<COSPATIAL FIELD LIST>]

<COSPATIAL FIELD LIST> ::=      <FIELD ELEMENT>
                              | <COSPATIAL FIELD LIST>, <FIELD ELEMENT>

<FIELD ELEMENT> ::=             <SIMPLE FIELD ELEMENT>
                              | <COMPLEX FIELD ELEMENT>

<SIMPLE FIELD ELEMENT> ::=      <SIMPLE IDENTIFIER> <FIELD TYPE>
                              | FILLER <FIELD TYPE>

<COMPLEX FIELD ELEMENT> ::=     <ARRAY IDENTIFIER> <ARRAY BOUND>
                                <FIELD TYPE>

<SIMPLE IDENTIFIER> ::=         <IDENTIFIER>

<ARRAY IDENTIFIER> ::=          <IDENTIFIER>

<ARRAY BOUND> ::=               (<CONSTANT EXPRESSION>)

<FIELD TYPE> ::=                FIXED
                              | BIT <FIELD SIZE>
                              | CHARACTER <FIELD SIZE>
                              | <RECORD IDENTIFIER>

<FIELD SIZE> ::=                (<CONSTANT EXPRESSION>)
```

DATA STRUCTURING

A new mechanism called Record is intended to eventually replace the PL/I-style structures currently being used in SDL. For compatibility, of course, no current features will be removed until they have fallen into disuse. Although records are used for the same purpose as the current structures, they are different in declaration, reference, and run-time effect. They are designed to provide the following benefits:

1. Since fields of records are not represented by descriptors at run-time, they do not cause large name stacks. This removes the need for USE declarations and elaborate SUBBITting schemes which have been used in the

past.

2. Paged arrays may be structured using records.

3. Arrays may occur nested in structural levels.

4. Accessing of linked data structures is safer, simpler, and often faster.

5. The substructure is specified in one place, but may be invoked in many places to declare variable or specify substructure of other records, thus reducing the probability of error.

6. The syntax encourages the treatment of data structures as new types, hopefully imposing better structure on programs.

RECORDS

A record is an addressing template analogous to a structure declared REMAPS BASE in the current language. Declaration of a record causes no data space to be allocated; it only establishes an addressing schema in the scope of the declaration. An example of a record declaration is:

```
RECORD    TYPEFIELD
              NV          BIT(1),
              NSR         BIT(1),
              DATATYPE    BIT(6);

RECORD    DESCRIPTOR
              TYPE        TYPEFIELD,
              LEN         BIT(16),
            [ADDR         BIT(24),
              VAL         BIT(24)];
```

This two-layered definition provides roughly the same effect as the following PL/I-style structure:

```
DECLARE   1 DESCRIPTOR REMAPS BASE,
            2 TYPE,
                3 NV                    BIT(1),
                3 NSR                   BIT(1),
                3 DATATYPE              BIT(6),
            2 LEN                       BIT(16),
            2 ADDR                      BIT(24),
            2 VAL   REMAPS ADDR         BIT(24);
```

The concept of making several fields alternative formats for the same area, or "cospatial", is expressed by enclosing the list of alternatives in brackets. This has the advantage of not requiring a distinguished alternative (the largest) which is remapped, and it also groups all the alternatives in one spot textually.

Another distinction of record is in the nested use of definitions to achieve the effect of PL/I level numbers. The advantage here is that a single record may be used as part of several other records, at different levels, or even more than once in another record declaration. This can be done without repeating the definition of its substructure, thus simplifying modifications. The use of a record in more than one context, of course, requires that qualified names be introduced. This is discussed later in detail.

Each field of a record has a type associated with it in the declaration (the type may be another record identifier), and may also be arrayed by noting the array bound after the field identifier-- similar to an ordinary array declaration. The type of an array field may be a record which also contains array fields, i.e., arrays may be nested in a way not permitted by the current SDL structures.

STRUCTURES

A structure which would be the functional equivalent of the current SDL structure may be declared using the previously defined record:

```
DECLARE D DESCRIPTOR;
```

Declaring this structure allocates storage on the value stack for the data (48 bits in this case) and allocates one descriptor on the name stack. A structure array could also be declared (and paged, in this example):

```
DECLARE PAGED(16) DA(256) DESCRIPTOR;
```

This causes one array descriptor to be allocated. The space for the array is not allocated on the value stack in this case because the array is paged.

The field of a structure is accessed by use of a qualified name. For example, the length field of descriptor "D" is named "D.LEN" and the type field is named "D.TYPE". The name-value bit of the type field is named "D.TYPE.NV". When a component of the name is an array, a subscript must be mentioned after that component as in "DA(20).TYPE.NSR". Qualification must be complete and explicit, unlike that of PL/I or COBOL. The dot notation was chosen because it is almost a standard among languages using qualified names. The underscore character ("_") is used as a replacement for the current use of "." as an identifier break character.

INDEXED FIELD REFERENCES

To provide a link between current and new facilities, a field of a record may be named by itself (no qualification) with an index. The effect is the same as indexing a field of a structure declared REMAPS BASE. This eases reprogramming since in many applications the structure declaration could be rewritten as a record without changing the rest of the code.

STRUCTURED RECORD STATEMENT

```
<STRUCTURED RECORD STATEMENT> ::=
            RECORD 01 <RECORD IDENTIFIER> <TYPE>
                      <STRUCTURE ELEMENTS>

<RECORD IDENTIFIER> ::= <IDENTIFIER>

<STRUCTURE ELEMENTS> ::=
                , <LEVEL NUMBER> <STRUCTURE ELEMENT>
              |   <LEVEL NUMBER> <STRUCTURE ELEMENT>
                , <STRUCTURE ELEMENTS>

<STRUCTURE ELEMENT> ::=
                    <FIELD NAME> <TYPE>
                  | <FIELD NAME> <ARRAY BOUND> <TYPE>
                  | FILLER <TYPE>
                  | <FIELD NAME> REMAPS <REMAPS OBJECT> <TYPE>
```

Structured Records have been implemented to allow easier conversion of the current PL/I-style structures to records.

Structured Records have the same capabilities as RECORDS.

Fields declared as an array may not have nested structure.

NON-STRUCTURE DECLARATIONS

```
<DECLARE ELEMENT> ::=     ...|<DECLARED PART>|...

<DECLARED PART> ::=         <COMPLEX IDENTIFIER> <TYPE PART>
                          | (<COMPLEX IDENTIFIER LIST>)
                            <TYPE PART>
                          | <COMPLEX IDENTIFIER> REMAPS
                            <REMAP OBJECT> <REMAPS TYPE PART>

<COMPLEX IDENTIFIER
LIST> ::=                   <COMPLEX IDENTIFIER>
                          | <COMPLEX IDENTIFIER>,
                            <COMPLEX IDENTIFIER LIST>

<COMPLEX IDENTIFIER> ::=    <SIMPLE IDENTIFIER>
                          | <ARRAY IDENTIFIER> <ARRAY BOUND>

<SIMPLE IDENTIFIER> ::=     <IDENTIFIER>

<ARRAY IDENTIFIER> ::=      <IDENTIFIER>

<ARRAY BOUND>      ::=      (<CONSTANT EXPRESSION>)

<REMAP OBJECT> ::=          BASE
                          | <SIMPLE IDENTIFIER>
                          | <ARRAY IDENTIFIER>
                          | <ADDRESS GENERATOR>

<TYPE PART> ::=             FIXED
                          | CHARACTER <FIELD SIZE>
                          | BIT <FIELD SIZE>
                          | <RECORD IDENTIFIER>

<REMAPS TYPE PART ::=       FIXED
                          | CHARACTER <FIELD SIZE>
                          | BIT <FIELD SIZE>

<RECORD IDENTIFIER> ::=     <IDENTIFIER>

<FIELD SIZE> ::=            (<CONSTANT EXPRESSION>)

<CONSTANT EXPRESSION> ::=   <NUMBER> | <CONSTANT EXPRESSION>
                            <CONSTANT EXPRESSION OPERATOR>
                            <NUMBER> | (<CONSTANT EXPRESSION>)
<CONSTANT EXPRESSION
OPERATOR> ::=               % + | - | * | / | MOD
```

Data may be declared as simple, having one occurrence, or as subscripted, having as many occurrences as specified by the <ARRAY BOUND>.

The <TYPE PART> specifies the type of data in the field and the field size.

As the syntax indicates, different data fields having the same type may be declared collectively as a <COMPLEX IDENTIFIER LIST>.

The following examples illustrate the various options available in this type of <DECLARATION STATEMENT>.

```
DECLARE A FIXED,
        B CHARACTER (10),
        C BIT (40),
       (D, E, F (5) ) BIT (10),
        G (20) FIXED,
        H (5) CHARACTER (6);
```

1. A is a 24-bit signed numeric field.
2. B is a 10-byte character field.
3. C is a 40-bit field
4. D and E are 10-bit fields each.
5. F is a 5-element array of 10-bit fields.
6. G is a 20-element array of 24-bit signed numeric fields.
7. H is a 6-byte character array with five elements.

Data fields may be re-formatted by the use of the remapping device:

<COMPLEX IDENTIFIER> REMAPS <REMAP OBJECT> <TYPE PART>

Remapping is subject to the same general rules discussed above. The following example best illustrates its use.

```
DECLARE  A FIXED, B BIT (50),
         AA REMAPS A CHARACTER (3),
         BB(2) REMAPS SUBBIT(B,2) FIXED;
```

Note that BB specifies 48-bits (or 2 elements, 24-bits each). A field may not be remapped larger than its original size. If the <REMAPS OBJECT> is an <ADDRESS GENERATOR> this check cannot be made until run time. The check will be made only when the the compiler option FORMAL_CHECK is set.

There is no limit on the number of times a field may be remapped, A field which has remapped another may itself be remapped. The REMAP option specifies that the identifier on the left side of the reserved word REMAPS will have the same starting address as the identifier on the right side.

For rules concerning the remapping of dynamic or formal declarations, see those sections.

A data field may be remapped to base which will give the field a relative address of zero. For example:

```
DECLARE X REMAPS BASE BIT(7);
```

This device is used as a free-standing declaration since it does not remap a previously declared data item and is used primarily with data to be indexed (See ADDRESS VARIABLES).

STRUCTURE DECLARATIONS

```
<DECLARE ELEMENT> ::=        ...|<STRUCTURE LEVEL NUMBER>
                             <STRUCTURE DECLARED PART>
                             <STRUCTURE TYPE PART> | ...

<STRUCTURE LEVEL
NUMBER> ::=                  <NUMBER>

<STRUCTURE DECLARED
PART> ::=                    <DECLARED PART>
                           | FILLER
                           | <DUMMY PART> REMAPS <REMAPS OBJECT>

<DECLARED PART> ::=          See NON-STRUCTURE DECLARATIONS

<DUMMY PART> ::=             DUMMY <ARRAY BOUND PART>

<ARRAY BOUND PART> ::=       <EMPTY>
                           | <ARRAY BOUND>

<ARRAY BOUND> ::=            (<CONSTANT EXPRESSION>)

<STRUCTURE TYPE PART> ::=    <EMPTY>
                           | <TYPE PART>
                           | CHARACTER | BIT

<TYPE PART> ::=  See NON-STRUCTURE DECLARATIONS
```

SDL allows the structuring of data where a field may be subdivided into a number of sub-fields, each of which has its own identifier. The whole structure is organized in a hierarchical form, where the most general declaration is at Level 01 (or 1) and the highest at Level 99. A subdivided field is called a group item, and a field not subdivided is known as an elementary item.

When the REMAPS option appears on a declare with level number greater than one, it is known as an intra-structure remap. In this case, the <REMAPS OBJECT> must be the last identifier declared in the same structure with the same level number unless that identifier was also declared with REMAPS. In that case both must remap the same identifier.

```
DECLARE 1 A,
          2 B  BIT(5),
          2 C  BIT(40),
               3 D  BIT (1),
          2 E REMAPS C  CHARACTER(1),
          2 F REMAPS C  FIXED,
          2 G FIXED;
```

is legal, but E and F may not remap B or D.

The type and length of data need not be specified on the group level. All elementary items must indicate type and length, and the compiler will assume type bit and add the lengths of the components to determine the length of the group item. For example:

```
DECLARE 01 A,
          02 C,
             03 D BIT(20),
             03 E BIT(30),
          02 D CHARACTER(5);
```

In this example, both A and C are considered group items, with A having a total length of 90 bits and C being 50 bits long.

FILLER

FILLERs may be used to designate certain elementary items which the program does not reference. If the group item has a length specified and the FILLER is the last item in a structure, it may be omitted, and the compiler will consider the item to be an implied FILLER. A FILLER may never be used as a group item.

A group item may have a type specified with length omitted. The compiler will calculate the length from the length of the sub-items. For example:

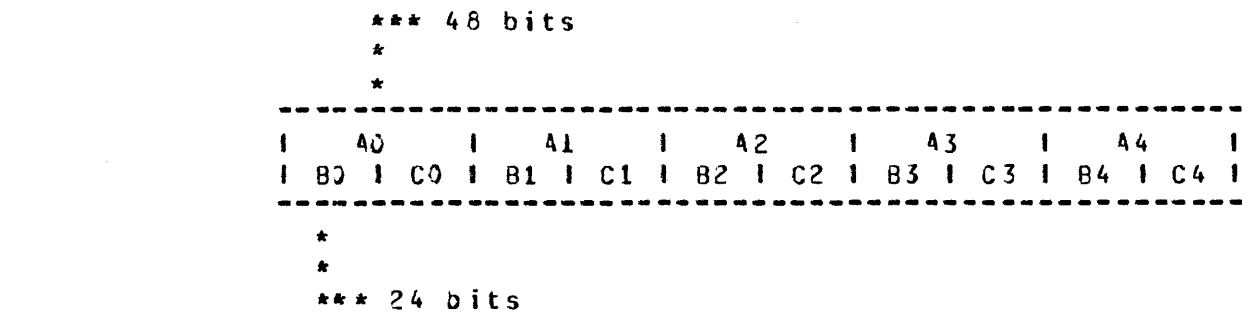
```
DECLARE  01  A  CHARACTER,
           02  B  FIXED,
           02  C  BIT(5);
```

A will become type CHARACTER(4) leaving an implied 3-bit filler after C.

If the 01 level group item is an array, it is mapped as a contiguous area in memory. However, subdivisions of this array are not contiguous. In the example structure below:

```
01 A(5) BIT(48),          01 A(5),
```

```
02 B FIXED,      or      02 B FIXED,
02 C FIXED;              02 C FIXED;
```



If a group item is an array, an array specification may not appear in any subordinate item; that is, only one-dimensional arrays are allowed. Down-level carry of array specifications is implied.

Structured data may be remapped in the same manner as non-structured data. In addition, structured data may be remapped with a dummy group identifier. The purpose of this construct is to allow the user to remap data items without having to declare another group item which describes the same memory area. Thus, in the following example:

```
01 A BIT(100),
   02 B BIT(20),
   02 C BIT(80);
```

"A" might be REMAPped as

```
01 AA REMAPS A BIT(100),      01 DUMMY REMAPS A BIT(100),
   02 BB BIT(30),      or        02 BB BIT(30),
   02 CC BIT(70);                02 CC BIT(70);
```

Both A and AA in the above example refer to the same area in memory. Hence AA is redundant. During runtime, the descriptor for AA will also be on the stack.

If DUMMY is substituted for the identifier AA, no descriptor will be generated, however BB and CC will both point to A in the correct fashion.

The user should note the distinction between DUMMY and FILLER. DUMMY is used in conjunction with REMAPS to eliminate the necessity of declaring a redundant group item. FILLER is used if one desires to skip over an area of core.

The following restrictions apply to the use of DUMMY REMAPS:

1. DUMMY may only be used with remap declarations.

2. All the restrictions applying to REMAPS apply to DUMMY REMAPS.

3. DUMMY must not remap another DUMMY.

4. DUMMY group items must have at least one non-filler component.

PAGED ARRAY DECLARATIONS

<DECLARE ELEMENT > ::= ...| PAGED <ELEMENTS-PER-PAGE PART> <ARRAY IDENTIFIER> <ARRAY BOUND> <TYPE PART>

<ELEMENTS-PER-PAGE PART> ::= (<CONSTANT EXPRESSION>)

<ARRAY IDENTIFIER> ::= <IDENTIFIER>

<ARRAY BOUND> ::= (<CONSTANT EXPRESSION>)

The paged array declaration allows the user to segment arrays. The <ELEMENTS-PER-PAGE PART> specifies the number of array elements contained in each segment. For example:

```
PAGED(64) A(4096) BIT(1);
```

is an array of 4096, 1-bit elements, segmented into 64, 64-element segments.

Restrictions:

1. Paged arrays may not be indexed.
2. Paged arrays may not be part of a structure.
3. Paged arrays may not be remapped.
4. The number of elements per page must be a power of 2, and may not exceed 32,768.
5. The <ARRAY BOUND> may not exceed 65,535 but the bounds may be subsequently increased to a maximum of 16,777,215 by use of the GROW statement.

DYNAMIC DECLARATIONS

```
<DECLARE ELEMENT> ::=          ...| DYNAMIC <DYNAMIC COMPLEX
                               IDENTIFIER> <DYNAMIC
                               TYPE PART>|...

<DYNAMIC COMPLEX
IDENTIFIER> ::=                <IDENTIFIER> | <ARRAY IDENTIFIER>
                               <DYNAMIC SUBSCRIPT BOUNDS>
                             | PAGED <DYNAMIC ELEMENTS PER PAGE>
                               <ARRAY IDENTIFIER>
                               <DYNAMIC SUBSCRIPT BOUNDS>

<DYNAMIC ELEMENTS
  PER PAGE>            ::=   (<EXPRESSION>)

<DYNAMIC SUBSCRIPT
  BOUNDS> ::=                (<EXPRESSION>)

<DYNAMIC TYPE PART> ::=        BIT <DYNAMIC FIELD SIZE>
                             | CHARACTER <DYNAMIC FIELD SIZE>
                             | FIXED
                             | <RECORD IDENTIFIER>

<DYNAMIC FIELD SIZE> ::=       (<EXPRESSION>)
```

The dynamic declare statement allows the user to declare simple data with a non-static field length and/or array bound. For example:

```
PROCEDURE ABX;
      DECLARE DYNAMIC X BIT(A);
```

where A will determine the length of X. The value of the <EXPRESSION> appearing in the <DYNAMIC FIELD SIZE> is used to determine the number of bits or characters in the declared data item. If X were an array, its bounds would be evaluated at run time as well.

Restrictions:

1. The variables used in the <DYNAMIC FIELD SIZE> must have been previously initialized.

2. Dynamics may not appear on Lexic Level 0.

Dynamic variables may be remapped, however a warning message will appear in the source listing. It is the programmer's responsibility to ensure that a dynamic is not remapped larger than allowed. If $FORMAL_CHECK is set, this remapping length will be run time checked.

REFERENCE DECLARATIONS

```
<DECLARE ELEMENT> ::=          ...|<DECLARED REF> REFERENCE|...

<DECLARED REF> ::=               <SIMPLE IDENTIFIER>
                               | (<SIMPLE IDENTIFIER LIST>)

<SIMPLE IDENTIFIER LIST> ::= <SIMPLE IDENTIFIER>
                               | <SIMPLE IDENTIFIER>,
                                 <SIMPLE IDENTIFIER LIST>
```

Reference variables are used as pointers to data and their declaration does not allocate data space. A reference variable has a close analog in a formal parameter declared VARYING. Such a parameter has only one type, length, and address associated with it for each invocation of the procedure in which it is declared, but it may be different for each invocation. The formal parameter is bound (to the actual parameter) by the procedure call mechanism. A reference variable is an extension of this idea because it may be declared anywhere other variables may be declared and may be rebound at any time using a statement known as the reference assignment statement or REFER statement. This statement binds the reference variable to a new referent. A few other SDL statements may change the referent of a reference variable also, but not to any arbitrary address generator as does the REFER statement.

RECORD REFERENCE DECLARATIONS

<DECLARE ELEMENT> ::= ...|<DECLARED RECORD REF> REFERENCE|...

<DECLARED RECORD REF> ::= <SIMPLE IDENTIFIER>
<RECORD IDENTIFIER>

RECORD REFERENCE VARIABLES

In some cases, storage is not to be directly allocated for a record, but a certain area of an array or large string is known to have the format specified by a record. This is the case in which indexing is applied currently. Record reference variables are designed to replace this use of indexing.

A record reference variable is declared, say for record DESCRIPTOR, as

```
DECLARE DR DESCRIPTOR REFERENCE;
```

Record reference variables are assigned with a REFER statement like ordinary reference variables, but they may be written in other statements as though they were structure names, i.e., they may have field qualifiers attached with the dot notation. Such an access subfields the current memory area described by the reference variable according to the record specification. For example,

```
REFER DR TO SUBBIT(MYAREA, 100, 48);
X := DR.LEN;
```

assigns X to bits 108 through 124 of the string MYAREA.

All restrictions which apply to normal reference variables are applicable to record reference variables as well. Record reference variables may not be used in the REDUCE statement.

FILE DECLARATIONS

```
<FILE DECLARATION
STATEMENT> ::=                  FILE <FILE DECLARE ELEMENT LIST>

<FILE DECLARE
ELEMENT LIST> ::=               <FILE DECLARE ELEMENT>
                              | <FILE DECLARE ELEMENT>,
                                <FILE DECLARE ELEMENT LIST>

<FILE DECLARE ELEMENT> ::=      <FILE IDENTIFIER><FILE ATTRIBUTE PART>

<FILE IDENTIFIER> ::=           <IDENTIFIER>

<FILE ATTRIBUTE PART> ::=       <EMPTY>
                              | (<FILE ATTRIBUTE LIST>)

<FILE ATTRIBUTE LIST> ::=       <FILE ATTRIBUTE>
                              | <FILE ATTRIBUTE>, <FILE ATTRIBUTE LIST>

<FILE ATTRIBUTE> ::=            <LABEL PART>
                              | <DEVICE PART>
                              | <MODE PART>
                              | <BUFFERS PART>
                              | <VARIABLE RECORD PART>
                              | <LOCK PART>
                              | <SAVE FACTOR PART>
                              | <RECORD SPECIFICATION PART>
                              | <REEL NUMBER PART>
                              | <DISK FILE DESCRIPTION PART>
                              | <PACK-ID PART>
                              | <OPEN OPTION PART>
                              | <ALL_AREAS_AT_OPEN PART>
                              | <AREA_BY_CYLINDER PART>
                              | <EU_ASSIGNMENT PART>
                              | <MULTI_PACK PART>
                              | <USE_INPUT_BLOCKING PART>
                              | <END_OF_PAGE PART>
                              | <REMOTE_KEY PART>
                              | <NUMBER_OF_STATIONS PART>
                              | <FILE TYPE PART>
                              | <WORK FILE PART>
                              | <LABEL TYPE PART>
                              | <INVALID CHARACTER REPORTING PART>
                              | <MONITOR SPECIFICATION PART>
                              | <SERIAL NUMBER PART>
                              | <OPTIONAL FILE PART>
                              | <TAPE LABEL PART>
                              | <EXCEPTION MASK PART>
                              | <TRANSLATE PART>
                              | <USER NAMED BACKUP PART>
                              | <PROTECTION PART>
                              | <PROTECTION_IO PART>
                              | <HOST_NAME PART>
```

All attributes are optional, as the above syntax indicates. Default status will automatically be set for omitted attributes as follows:

SYNTAX: <LABEL PART> ::= LABEL =
<FILE IDENTIFICATION PART>

<FILE IDENTIFICATION PART> ::= <MULTI-FILE IDENTIFICATION>
| <MULTI-FILE IDENTIFICATION>
<FILE IDENTIFICATION>

<MULTI-FILE IDENTIFICATION> ::= <CHARACTER STRING>

<FILE IDENTIFICATION> ::= <CHARACTER STRING>

where:

<FILE IDENTIFIER> is a file or program identifier by which the program identifies the file.

and:

<MULTI-FILE IDENTIFICATION> and <FILE IDENTIFICATION> are name or contents of identification field on file label or Disk Directory by which the system identifies the file.

FORMAT: LABEL = "NAME_1" / "NAME_2"
or
LABEL = "NAME_1"

Example:

```
FILE INV_DATA_1 (LABEL = "RCD_TAPE" / "FILE_1");
```

Note: The system will use only the first ten characters of the "NAME".

DEFAULT If LABEL(s) is (are) not specified, the INTERNAL FILE NAME, i.e., <FILE IDENTIFIER>, is moved to <MULTI-FILE IDENTIFICATION>, and blanks are moved to <FILE IDENTIFICATION> in the FPB (FILE PARAMETER BLOCK).

SYNTAX: <DEVICE PART> ::= DEVICE = <DEVICE SPECIFIER>

```
<DEVICE SPECIFIER> ::=          TAPE
                              | TAPE_7
                              | TAPE_9
                              | TAPE_PE
                              | TAPE_NRZ
                              | DISK <ACCESS MODE>
                              | DISK_PACK <ACCESS MODE>
                              | DISK_FILE <ACCESS MODE>
                              | DISK_PACK_CENTURY <ACCESS MODE>
                              | DISK_PACK_CAELUS <ACCESS MODE>
                              | CARD
                              | CARD_READER
                              | CARD_PUNCH <DEVICE OPTION>
                              | PRINTER <DEVICE OPTION>
                              | PUNCH <DEVICE OPTION>
                              | PAPER_TAPE_PUNCH
                                <DEVICE OPTION>
                              | DATA_RECORDER_80
                              | READER_PUNCH_PRINTER
                                <DEVICE OPTION>
                              | PUNCH_PRINTER <DEVICE OPTION>
                              | READER_96
                              | PAPER_TAPE_READER
                              | SORTER_READER
                              | READER_SORTER
                              | CASSETTE
                              | REMOTE (<QUEUE SIZE>) <REMOTE
                                OPTION>
                              | QUEUE (<QUEUE SIZE>)
                                <QUEUE OPTION>

<ACCESS MODE> ::=               <EMPTY> | SERIAL | RANDOM
<DEVICE OPTION> ::=             <EMPTY>
                              | <BACKUP OPTION>
                              | <SPECIAL FORMS OPTION>
                              | <SPECIAL FORMS OPTION>
                                <BACKUP OPTION>

<BACKUP OPTION> ::=             <BACKUP SPECIFIER>
                              | OR <BACKUP SPECIFIER>
                              | NO BACKUP
<BACKUP SPECIFIER> ::=          BACKUP | BACKUP TAPE
                              | BACKUP DISK

<SPECIAL FORMS OPTION> ::=      FORMS

<REMOTE OPTION> ::=             <EMPTY> | FAMILY | WITH HEADERS
                              | FAMILY WITH HEADERS

<QUEUE SIZE> ::=                <NUMBER>

<QUEUE OPTION> ::=              <EMPTY>
```

| FAMILY (<FAMILY SIZE>)

<FAMILY SIZE> ::= <NUMBER>

FORMAT: DEVICE = CARD
CARD_READER
TAPE
TAPE_7
TAPE_9
TAPE_PE
TAPE_NRZ
** DISK
** DISK_PACK
** DISK_FILE
** DISK_PACK_CENTURY
** DISK_PACK_CAELUS
* CARD_PUNCH
* PRINTER
* PRINTER FORMS
* PUNCH
* PUNCH FORMS
* PAPER_TAPE_PUNCH
* PAPER_TAPE_PUNCH FORMS
* DATA_RECORDER_80
* READER_PUNCH_PRINTER
* READER_PUNCH_PRINTER FORMS
* PUNCH_PRINTER
* PUNCH_PRINTER FORMS
READER_96
PAPER_TAPE_READER
SORTER_READER
READER_SORTER
CASSETTE
*** REMOTE (<QUEUE SIZE>)
*** QUEUE (<QUEUE SIZE>)

* may or may not be followed by any single option below:

BACKUP
BACKUP TAPE
BACKUP DISK
OR BACKUP
OR BACKUP TAPE
OR BACKUP DISK
NO BACKUP

Note: See <USER NAMED BACKUP PART> for more on backup.

** may or may not be followed by any single option below:

SERIAL
RANDOM

*** may or may not be followed by options applicable to this "device". See syntax above.

Examples: DEVICE = TAPE
DEVICE = PRINTER BACKUP
DEVICE = PRINTER FORMS BACKUP TAPE
DEVICE = REMOTE(5) WITH HEADERS

DEFAULT: In the absence of any specification, disk will be assumed by the compiler.

SYNTAX: <MODE PART> ::= MODE = <MODE SPECIFIER>

<MODE SPECIFIER> ::= <FILE PARITY PART>
| <TRANSLATION PART>

<FILE PARITY PART> ::= ODD | EVEN

<TRANSLATION PART> ::= EBCDIC | ASCII | BCL | BINARY

FORMAT: MODE = BCL
MODE = ASCII
MODE = EVEN

DEFAULT: Default is odd or EBCDIC, whichever is applicable.

SYNTAX: <BUFFERS PART> ::= BUFFERS =
<NUMBER OF BUFFERS>

<NUMBER OF BUFFERS> ::= <NUMBER>

FORMAT: BUFFERS = NUMBER

DEFAULT: If not specified, buffers will be set to 1 in the FPB.

SYNTAX: <VARIABLE RECORD PART> ::= VARIABLE

FORMAT: VARIABLE

DEFAULT:= Not variable, i.e., fixed-size records.

SYNTAX: <LOCK PART> ::= LOCK

FORMAT: LOCK

DEFAULT:= LOCK is not set.

SYNTAX: <SAVE FACTOR PART> ::= SAVE = <SAVE FACTOR>

<SAVE FACTOR> ::= <NUMBER>

FORMAT: SAVE = NUMBER (of days to save file)

DEFAULT: If not specified, the SAVE specifier will be set to 30 in the FPB.

SYNTAX: <RECORD SPECIFICATION PART> ::= RECORDS = <RECORD SIZE SPECIFIER>

<RECORD SIZE SPECIFIER> ::= <PHYSICAL RECORD SIZE>
| <LOGICAL RECORD SIZE>
<SLASH>
<LOGICAL RECORDS PER PHYSICAL RECORD>

<PHYSICAL RECORD SIZE> ::= <NUMBER>

<LOGICAL RECORD SIZE> ::= <NUMBER>

<LOGICAL RECORDS PER PHYSICAL RECORD> ::= <NUMBER>

FORMAT: RECORDS = NUMBER
or
RECORDS = NUMBER / NUMBER

Note: <PHYSICAL RECORD SIZE> indicates the number of characters per block; <LOGICAL RECORD SIZE>, the number of characters per record.

Example:

RECORDS = 1200
or
RECORDS = 120 / 10

DEFAULT: In the absence of record specifications, unblocked records

of the following lengths will be assumed.

Disk	180 bytes
Tape	80 bytes
Any paper tape configuration	80 bytes
Any 96 column card configuration	96 bytes
All remaining card configurations	80 bytes
Any printer configuration	132 bytes
All others	72 bytes

SYNTAX: <REEL NUMBER PART> ::= REEL = <REEL NUMBER>

<REEL NUMBER> ::= <NUMBER>

FORMAT: REEL = 2

DEFAULT: The FPB assumes #1 in the absence of any specification.

SYNTAX: <DISK FILE DESCRIPTION PART> ::= AREAS = <NUMBER OF AREAS> <SLASH> <PHYSICAL RECORDS PER AREA>

<NUMBER OF AREAS> ::= <NUMBER>

<PHYSICAL RECORDS PER AREA> ::= <NUMBER>

Format: Areas = # of Areas / #of Blocks Per Area

Example: Areas = 20 / 80

Note: <PHYSICAL RECORDS PER AREA> indicates the number of blocks per area. This attribute is applicable for disk files only.

DEFAULT: If areas are not specified, the FPB will assume 25 Areas with 100 Blocks Per Area. If the record specifications have been given the compiler will compute the number of Records Per Area. However, if record specifications are omitted, the FPB will assume 100 records per area. In either case then, whether areas are specified or not, the compiler will have computed the number of records for insertion in the FPB.

SYNTAX: <PACK_ID PART> ::= PACK_ID = <PACK IDENTIFICATION>

<PACK IDENTIFICATION> ::= <CHARACTER STRING>

FORMAT: PACK_ID = "NAME"

Note: The system will use only the first ten characters of the "NAME".

DEFAULT: If absent, <PACK IDENTIFICATION> will be set to blanks in the FPB.

SYNTAX: <OPEN OPTION>::= OPEN_OPTION= <OPEN OPTION ATTRIBUTE LIST>

<OPEN OPTION ATTRIBUTE LIST>::= <OPEN ATTRIBUTE>
| <OPEN ATTRIBUTE> <SLASH> <OPEN OPTION ATTRIBUTE LIST>

<OPEN ATTRIBUTE> ::= SEE "OPEN STATEMENT" 9-2

FORMAT: OPEN_OPTION = ATTRIBUTE / ATTRIBUTE. . .

Example: OPEN_OPTION = OUTPUT / NEW

CAVEAT - These apply only to default OPEN

Note:

<OPEN STATEMENT> may be separated by commas, and the <OPEN ATTRIBUTE>s in the <OPEN OPTION> above are separated by slashes.

i.e. Names are overridden by explicit OPEN!!

DEFAULT: If absent, the <OPEN ATTRIBUTE>s will be set as follows:

If <DEVICE> is:	<OPEN OPTION> is:
CARD	INPUT
PRINTER	OUTPUT
PUNCH	OUTPUT
DISK	INPUT
REMOTE	INPUT/OUTPUT
TAPE	INPUT
QUEUE	INPUT/OUTPUT

SYNTAX: <ALL_AREAS_AT_OPEN PART> ::= ALL_AREAS_AT_OPEN

FUNCTION: If this option is set, disk space for each area will be allocated when the file is opened. If insufficient space is available, a SPO message will indicate that there is no user disk.

DEFAULT: Areas are created as needed.

SYNTAX: <AREA_BY_CYLINDER PART> ::= AREA_BY_CYLINDER

FUNCTION: If this option is specified, each area will be placed at the beginning of a cylinder. If there is no (more) space at the beginning of any cylinder, a SPO message will indicate that there is no user disk.

DEFAULT: Areas are placed anywhere on disk.

SYNTAX: <EU ASSIGNMENT PART> ::= EU_SPECIAL = <NUMBER>
| EU_INCREMENTED = <NUMBER>

FUNCTION: The <NUMBER> specifies any integer 0 through 15. "EU_SPECIAL" is applicable only with head-per-track disks and systems disk packs, and specifies the drive on which the file is to go. "EU_INCREMENTED" specifies the disk drive on which the first area of a file is to go. Each subsequent area is placed on the next drive. If, with either option, the necessary E.U. is not available, E.U. 0 will be taken.

DEFAULT: Space for files and areas is allocated anywhere on disk.

SYNTAX: <MULTI PACK PART>::= MULTI_PACK

FUNCTION: If this option is specified, the entire file may be put onto several disk packs.

DEFAULT: The file will be placed on one disk pack.

SYNTAX: <USE_INPUT_BLOCKING PART> ::= USE_INPUT_BLOCKING

FUNCTION: This option applies to input disk, tape, or card files. If specified for disk, the record and block size specifications will be taken from the Disk File Header and the user's specifications will be ignored. If specified for tape, the tape must be labeled; otherwise, a run-time error occurs. If specified for card files, the following record lengths will be assumed:

80-col = 80 bytes
96-col = 96 bytes
BIN = 960 bits

DEFAULT: The record and block size are as stated in the file declaration. Those options omitted are set to default status.

SYNTAX: <END_OF_PAGE PART> ::= END_OF_PAGE_ACTION

FUNCTION: This attribute will cause the <EOF PART> of a <WRITE STATEMENT> to be executed at the end of a page on a printer file. Refer to "WRITE STATEMENT" and "ON SEQUENCE" for details.

DEFAULT: No automatic paging action

SYNTAX: <REMOTE_KEY PART>::= REMOTE_KEY

FUNCTION: This atrribute is used only with files of type "REMOTE". When present, it indicates that a key may be present on a read or write to that file. If missing, then no key can be used. The format of the key is given below. Each field of the key is in decimal characters. The key is a total of 10 characters formatted as follows:

Station Number	3 characters
Message Length (byte count)	4 characters
Message Type (must be "000")	3 characters

DEFAULT: No remote key

SYNTAX: <NUMBER_OF_STATIONS PART>:= NUMBER_OF_STATIONS = <NUMBER>

FUNCTION: This attribute is used only with files of type "REMOTE". When present, it specifies the maximum number of stations that can be attached to this file.

DEFAULT: NUMBER_OF_STATIONS=1

SYNTAX: <FILE TYPE PART>::= FILE_TYPE=<FILE TYPE SPECIFIER>

<FILE TYPE SPECIFIER>::= DATA | INTERPRETER | CODE
| INTRINSIC | PSR_DECK

FUNCTION: This attribute allows SDL programs to specify the type of the files they are creating. In particular, the compilers will use the type "CODE" for their codefiles.

DEFAULT: FILE_TYPE = DATA

SYNTAX: <WORK FILE PART>::= WORK_FILE

FUNCTION: This attribute causes the job number to be included as part of the file identifier.

DEFAULT: Not a workfile

SYNTAX: <LABEL TYPE PART>::= LABEL_TYPE=<LABEL TYPE SPECIFIER>

<LABEL TYPE SPECIFIER>::= UNLABELED | BURROUGHS

FUNCTION: This attribute allows the label type to be specified.

DEFAULT: ANSII STANDARD LABEL

SYNTAX: <INV_CHAR_REPORTING PART>::= INVALID_CHARACTERS=
<INV_CHAR_REPORT TYPE PART>

<INV_CHAR_
REPORT TYPE PART> ::= 0 | 1 | 2 | 3

FUNCTION: Invalid characters occurring in a print file will be reported on the SPO to the computer operator, as

specified:

VALUE	TYPE
0	Report all lines containing invalid characters.
1	Report all lines containing invalid characters and then stop program.
2	Report once that the file contains invalid characters.
3	Do not report that the file contains invalid characters.

DEFAULT: 0

SYNTAX: <MONITOR SPEC PART> ::= MONITOR_INPUT_FILE
| MONITOR_OUTPUT_FILE

FUNCTION: See Appendix VIII: SDL MONITORING FACILITY

DEFAULT Not present

SYNTAX: <SERIAL NUMBER PART> ::= SERIAL = <NUMBER>
| SERIAL = <CHARACTER STRING>

FUNCTION: The file will be opened on the output media with the specified serial number.

DEFAULT Not present

SYNTAX: <OPTIONAL_FILE_PART> ::= OPTIONAL

FUNCTION: If this option is used on an input file, then the file may be missing and the operator may respond with the OF message to the FILE MISSING message. This will result in the execution of the ON EOF branch on the execution of the first read of the file.

DEFAULT: Reset

SYNTAX: <EXCEPTION MASK PART> ::= EXCEPTION_MASK = <BIT STRING>

FUNCTION: The exception mask specifies the types of exceptions that the program is willing to handle for this particular file. See the B1700 MCP Manual for a description of the bit assignment within the bit string. Note that this string should generate a 24-bit value.

DEFAULT: @0000000@

SYNTAX: <TRANSLATE PART> ::= TRANSLATE = <CHARACTER STRING>

FUNCTION: The MCP will do a soft translation on the file using <CHARACTER STRING> as the file-id for the translate table file. The multi-file-id for the translate table file will be "TRANSLATE".

DEFAULT: DEFAULT: No translation.

SYNTAX: <USER NAMED BACKUP PART>::= USER_NAMED_BACKUP

FUNCTION: If the file goes to backup, its name will be its given external name rather than a system selected name.

DEFAULT: System selects backup file names.

SYNTAX: <PROTECTION PART>::= PROTECTION = <PROTECTION TYPE PART>
<PROTECTION TYPE PART>::= 0 | 1 | 2 | 3

FUNCTION: (See MCP Control Syntax product specification in File Attribute description.)

SYNTAX: <PROTECTION_IO_PART>::= PROTECTION_IO = <PROTECTION_IO TYPE PART>
<PROTECTION_IO TYPE PART::= 0 | 1 | 2 | 3

FUNCTION: (See MCP Control Syntax product specification in File Attribute description.)

SYNTAX: <HOST_NAME PART> := HOST_NAME = <CHARACTER STRING>

FUNCTION: Specifies the name of the host system for this file.

DEFAULT: No host specified.

SWITCH FILE DECLARATIONS

```
<SWITCH FILE
DECLARATION STATEMENT>::=     SWITCH_FILE <SWITCH FILE
                              DECLARE ELEMENT LIST>

<SWITCH FILE
DECLARE ELEMENT LIST>::=        <SWITCH FILE DECLARE ELEMENT>
                              | <SWITCH FILE DECLARE ELEMENT>,
                                <SWITCH FILE DECLARE ELEMENT LIST>

<SWITCH FILE
DECLARE ELEMENT> ::=          <SWITCH FILE IDENTIFIER> (<FILE
                              IDENTIFIER LIST>)

<SWITCH FILE IDENTIFIER>::=   <IDENTIFIER>

<FILE IDENTIFIER LIST>::=     <FILE IDENTIFIER>
                              | <FILE IDENTIFIER>, <FILE IDENTIFIER LIST>
```

A switch file declaration specifies the elements of a "CASE", these elements being files. A subscripted <SWITCH FILE IDENTIFIER> may be used anywhere that a <FILE IDENTIFIER> may be used. If there are N files in the <FILE IDENTIFIER LIST>, then the subscript must range from 0 to N-1. The value of the subscript selects one of the N files in the list, depending upon ordinal position (the files in the <FILE IDENTIFIER LIST> are numbered from left to right, begining with 0). If all files in the <FILE IDENTIFIER LIST> are of type "REMOTE", then the switch file identifier is of type "REMOTE".

The following example copies card images from cards, tape, or disk to cards, printer, tape, or disk:

```
FILE
     CARDS(DEVICE=CARD)
    ,TAPEI(DEVICE=TAPE,USE_INPUT_BLOCKING)
    ,DISKI(DEVICE=DISK,USE_INPUT_BLOCKING)
    ;
FILE
     PUNCH(DEVICE=PUNCH)
    ,LINE(DEVICE=PRINTER)
    ,TAPEO(DEVICE=TAPE,RECORDS=80/4)
    ,DISKO(DEVICE=DISK,RECORDS=80/9)
    ;
SWITCH_FILE
     INPUT(CARDS,TAPEI,DISKI)
    ,OUTPUT(PUNCH,LINE,TAPEO,DISKO)
    ;
DECLARE
     INPUT_TYPE BIT(24)
    ,OUTPUT_TYPE BIT(24)
    ,BUFFER CHARACTER(80)
    ;
DISPLAY "***** INPUT TYPE";
ACCEPT INPUT_TYPE;
INPUT_TYPE:BINARY(SUBSTR(INPUT_TYPE,0,1)) MOD 3;
DISPLAY "***** OUTPUT TYPE";
ACCEPT OUTPUT_TYPE;
OUTPUT_TYPE:BINARY(SUBSTR(OUTPUT_TYPE,0,1)) MOD 4;
OPEN INPUT(INPUT_TYPE) INPUT;
OPEN OUTPUT(OUTPUT_TYPE) OUTPUT, NEW;
DO FOREVER;
    READ INPUT(INPUT_TYPE) (BUFFER);
         ON EOF UNDO;
    WRITE OUTPUT(OUTPUT_TYPE) (BUFFER);
END;
CLOSE OUTPUT(OUTPUT_TYPE) WITH LOCK;
STOP;
FINI
```

DEFINE STATEMENT

```
<DECLARATION STATEMENT> ::=  ...|<DEFINE STATEMENT>;|...

<DEFINE STATEMENT> ::=       DEFINE <DEFINE ELEMENT>
                           | <DEFINE STATEMENT>,
                             <DEFINE ELEMENT>

<DEFINE ELEMENT> ::=         <DEFINE IDENTIFIER>
                             <FORMAL PARAMETER PART>
                             AS <DEFINE STRING>

<DEFINE IDENTIFIER> ::=      <IDENTIFIER>

<FORMAL PARAMETER PART> ::=  (<FORMAL PARAMETER LIST>)
                           | [<FORMAL PARAMETER LIST>]
                           | <EMPTY>

<FORMAL PARAMETER LIST> ::=  <FORMAL PARAMETER>
                           | <FORMAL PARAMETER>,
                             <FORMAL PARAMETER LIST>

<FORMAL PARAMETER> ::=       <IDENTIFIER>

<DEFINE STRING> ::=          #<WELL-FORMED CONSTRUCT>#

<WELL-FORMED CONSTRUCT> ::=  <EMPTY>

                           | <BASIC COMPONENT>
                             <WELL-FORMED CONSTRUCT>

<BASIC COMPONENT> ::=        <RESERVED WORD>      %SEE APPENDIX
                           | <IDENTIFIER>
                           | <SPECIAL CHARACTER>
                           | <COMMENT STRING>
                           | <CONSTANT>
```

The <DEFINE STATEMENT> assigns the text enclosed between the "#" signs following the reserved word AS to the <DEFINE IDENTIFIER>. Invocation of the <DEFINE IDENTIFIER> causes the text to replace the identifier, thereby providing a form of shorthand code.

At declaration time, the compiler is unconcerned with the contents of the <DEFINE STRING>. However, when the <DEFINE IDENTIFIER> is invoked, the <WELL-FORMED CONSTRUCT> must conform to the syntactical requirements of the statement containing the identifier.

There are two types of <DEFINE STATEMENT>s: Simple and Parametric, where the parameters are enclosed in parentheses or brackets following the <DEFINE IDENTIFIER>. Below are examples of both types:

```
DEFINE A AS #IF X>10 THEN PROCX#,
       CH AS #CHARACTER#,
       B(Y,Z) AS #IF Y<Z THEN Y:=Z #,
       C(M) AS # X:=M;  A #;
```

Notice that <DEFINE STATEMENT>s may be factored, with commas separating each element.

The <DEFINE IDENTIFIER> has scope in the same manner as any other identifier (except for SEGMENT and DO-GROUP identifiers).

Restrictions on the use of DEFINEs:

1. Reserved words may not be used as <DEFINE IDENTIFIER>s, however, an identifier may define a reserved word.

2. "Special" words may be used as <DEFINE IDENTIFIER>s, however, their special significance is lost within the the scope of that <DEFINE STATEMENT>.

3. <DEFINE INVOCATION>s may appear within a <WELL-FORMED CONSTRUCT>, i.e., a <DEFINE IDENTIFIER> may appear within another <DEFINE ELEMENT>. <DEFINE IDENTIFIER>s may be nested no more than 12 levels deep.

4. The identifiers listed below are never looked up in the list of define names.

   DECLARE, DEFINE, PROCEDURE, and FORMAL IDENTIFIERS,

   SEGMENT and DO-GROUP IDENTIFIERS,

   FILE, OPEN, and CLOSE ATTRIBUTES,

   <FILE ATTRIBUTE STATEMENT> attribute names

   "ON" condition names (EOF, EXCEPTION, FILE_MISSING, Q_FULL, Q_EMPTY,NO_INPUT, FILE_LOCKED, INCOMPLETE_IO)_

   "ACCEPT"/"DISPLAY" specifiers: END_OF_TEXT and CRUNCHED.

If one of these identifiers happens to be the same as a <DEFINE IDENTIFIER>, no substitution occurs. The <WELL-FORMED CONSTRUCT> of the define will not replace the identifier. Note, however, that duplicate identifiers may not appear within the same lexic level; an error message results.

5. There may be no more than eight <FORMAL PARAMETER>s in a <FORMAL PARAMETER LIST>.

6. Refer to Appendix V for rules concerning conditional inclusion cards within defines.

The following syntax illustrates the format used in the invocation of a <DEFINE IDENTIFIER>:

```
<DEFINE INVOCATION> ::=      <SIMPLE DEFINE IDENTIFIER>
                           | <PARAMETRIC DEFINE IDENTIFIER>
                             (<DEFINE ACTUAL PARAMETER LIST>)
                           | <PARAMETRIC DEFINE IDENTIFIER>
                             [<DEFINE ACTUAL PARAMETER LIST>]

<SIMPLE DEFINE
IDENTIFIER> ::=              <DEFINE IDENTIFIER>

<PARAMETRIC
DEFINE IDENTIFIER> ::=       <DEFINE IDENTIFIER>

<DEFINE ACTUAL
PARAMETER LIST> ::=          <DEFINE ACTUAL PARAMETER>
                           | <DEFINE ACTUAL PARAMETER>,
                             <DEFINE ACTUAL PARAMETER LIST>

<DEFINE ACTUAL
PARAMETER> ::=               <WELL-FORMED CONSTRUCT>
```

A <DEFINE INVOCATION> may occur anywhere within an SDL program except in the cases listed above in Restriction 4. As indicated by the above BNF, the actual parameters of a define are not confined to constants and variables but may have a wide range of constructs. For example, the <DEFINE STATEMENT> mentioned above:

```
DEFINE A AS #IF X>10 THEN PROCX#,
       CH AS #CHARACTER#,
       B(Y,Z) AS #IF Y<Z THEN Y:=Z #,
       C(M) AS # X:=M; A #;
```

might be invoked as follows:

```
C(Z;BUMP I[R,S]);
```

which expands to:

```
X:=Z; BUMP I[R,S]; IF X>10 THEN PROCX;
```

The following restrictions apply to the use of the <DEFINE INVOCATION>:

1. No unpaired bracketing symbols, i.e., () or [], may appear.

2. Within a <DEFINE ACTUAL PARAMETER LIST>, commas not enclosed within paired bracketing symbols act to delimit the <DEFINE ACTUAL PARAMETER>s. Therefore a <WELL-FORMED CONSTRUCT> not enclosed in bracketing symbols may not contain commas. For example:

```
DEFINE X(A,B) AS # A(B) #;
```

and invoked as:

```
Z:=X(M,Q,R,S);
```

would result in the error message:

```
DEFINE INVOCATION HAS TOO MANY PARAMETERS
```

Proper invocation is possible by removing the parens from the define and placing them in the invocation:

```
DEFINE X(A,B) AS # A B #;
    Z:=X(M,(Q,R,S));
```

3. Comments are allowed but will be deleted from the actual parameter text.

FORWARD DECLARATION

```
<DECLARATION STATEMENT> ::=    ...|<FORWARD DECLARATION>|...

<FORWARD DECLARATION> ::=      FORWARD <COMPOUND PROCEDURE HEAD>

<COMPOUND PROCEDURE
HEAD> ::=                      <PROCEDURE HEAD>
                               <FORMAL PARAMETER DECLARATION
                               STATEMENT LIST>

<PROCEDURE HEAD> ::=           <BASIC PROCEDURE HEAD>
                               <PROCEDURE TYPE PART>;

<BASIC PROCEDURE HEAD> ::=     <PROCEDURE NAME>
                               <FORMAL PARAMETER PART>

<PROCEDURE NAME> ::=           PROCEDURE <PROCEDURE IDENTIFIER>

<PROCEDURE IDENTIFIER> ::=     <TYPED PROCEDURE IDENTIFIER>
                             | <NON-TYPED PROCEDURE IDENTIFIER>

<TYPED PROCEDURE
IDENTIFIER> ::=                <IDENTIFIER>

<NON-TYPED PROCEDURE
IDENTIFIER> ::=                <IDENTIFIER>

<FORMAL PARAMETER PART> ::=    <EMPTY>
                             | (<FORMAL PARAMETER LIST>)

<FORMAL PARAMETER LIST> ::=    <FORMAL PARAMETER>
                             | <FORMAL PARAMETER>,
                               <FORMAL PARAMETER LIST>

<FORMAL PARAMETER> ::=         <IDENTIFIER>

<PROCEDURE TYPE PART> ::=      <EMPTY>
                             | <FORMAL TYPE PART>

<FORMAL TYPE PART> ::=         <TYPE PART>
                             | <TYPE VARYING PART>

<TYPE PART> ::=                FIXED
                             | CHARACTER <FIELD SIZE>
                             | BIT <FIELD SIZE>

<TYPE VARYING PART> ::=        VARYING
                             | BIT VARYING
                             | CHARACTER VARYING

<FORMAL PARAMETER DECLA-
RATION STATEMENT LIST> ::=     <EMPTY>
```

```
                              | <FORMAL PARAMETER DECLARATION STATEMENT>;
                                <FORMAL PARAMETER DECLARATION
                                STATEMENT LIST>

<FORMAL PARAMETER
DECLARATION STATEMENT> ::=      FORMAL <FORMAL ELEMENT>
                              | FORMAL_VALUE <FORMAL ELEMENT>
                              | <FORMAL PARAMETER DECLARATION STATEMENT>,
                                <FORMAL ELEMENT>

<FORMAL ELEMENT> ::= (<FORMAL IDENTIFIER LIST>)
                                <FORMAL TYPE PART>
                              | <FORMAL IDENTIFIER>
                                <FORMAL TYPE PART>

<FORMAL INDENTIFIER LIST> ::= <FORMAL IDENTIFIER>
                              | <FORMAL IDENTIFIER LIST>,
                                <FORMAL IDENTIFIER>

<FORMAL IDENTIFIER> ::= <COMPLEX IDENTIFIER>
                              | <VARYING ARRAY SPECIFIER>

<COMPLEX IDENTIFIER> ::= <SIMPLE IDENTIFIER>
                              | <ARRAY IDENTIFIER>
                                <ARRAY BOUND>

<VARYING ARRAY SPECIFIER> ::= <ARRAY IDENTIFIER>
                                <VARYING ARRAY BOUND>

<VARYING ARRAY BOUND> ::= (*)
```

Before a procedure may be called, SDL specifies that it must have been previously declared. A contradiction arises when one procedure calls another procedure which in turn references the first. In this case, whichever procedure appears first must necessarily contain at least one reference to the second which has not yet been declared.

The <FORWARD DECLARATION> allows the programmer to use recursive references by providing a temporary procedure declaration. The <FORWARD DECLARATION>, however, does not eliminate the need for the normal procedure declaration which must follow in the program and must have the same scope.

The parameters mentioned in the <FORWARD DECLARATION> must be the same formal parameters (in type and size, but not in name) that the procedure itself will declare.

Procedures may be either typed or non-typed depending on their use. Formal data types may either be static or varying, again depending on the program. These specifications will be discussed in the section entitled "THE PROCEDURE STATEMENT".

The following examples illustrate the use of the <FORWARD DECLARATION>:

```
FORWARD   PROCEDURE X CHARACTER VARYING;
FORWARD   PROCEDURE J(K,L,M);
          FORMAL K(*) BIT VARYING,
          L(15) CHARACTER (8),
          M FIXED;
```

USE STATEMENT

```
<USE STATEMENT> ::=          USE (<SIMPLE IDENTIFIER LIST>)
                             OF <DEFINE IDENTIFIER>

<SIMPLE IDENTIFIER
LIST> ::=                    <SIMPLE IDENTIFIER>
                           | <SIMPLE IDENTIFIER LIST>, <SIMPLE IDENTIFIER>

<SIMPLE IDENTIFIER> ::=      <IDENTIFIER>

<DEFINE IDENTIFIER> ::=      <IDENTIFIER>
```

The purpose of the <USE STATEMENT> is to allow the programmer to declare specific elements in a defined structure within a procedure. By specifying only the desired elements, the Name Stack size is kept to a minimum, and program maintenance is simplified. The compiler will generate the structure using fillers and the specified elements.

The following restrictions apply to the <USE STATEMENT>:

1. It must appear within a procedure (i.e., on a lexic level greater than 0).
2. The referenced <DEFINE IDENTIFIER> must define one structured declare statement.
3. The structure may not contain arrays.
4. The outermost level of the structure (01) must be a "DUMMY REMAPS".

EXAMPLE:

```
DEFINE X AS #
DECLARE 01 DUMMY REMAPS A, % MIGHT ALSO REMAP BASE
           02 B    BIT(5),
             03 B1 BIT(2),
             03 B2 BIT(3),
           02 C    CHARACTER(10),
           02 D    BIT(1),
           02 E    FIXED,
           02 F    BIT(24)#;
PROCEDURE FIRST;
     USE (C,D) OF X;
```

From the above <USE STATEMENT> the compiler will generate the following structure:

```
01  DUMMY REMAPS A,
    02  FILLER    BIT(5),
       03 FILLER BIT(2),
       03 FILLER BIT(3),
    02  C         CHARACTER(10),
    02  D         BIT(1),
    02  FILLER    FIXED,
    02  FILLER    BIT(24);
```

Note that filler was substituted for the group item B. This would normally generate a syntax error, and is allowable only in the <USE STATEMENT>.

PROCEDURES

<SEGMENT STATEMENT>::= SEE "THE SEGMENT STATEMENT"

<PROCEDURE STATEMENT> ::= <PROCEDURE DEFINITION>
| <SEGMENT STATEMENT>
<PROCEDURE STATEMENT>

<PROCEDURE DEFINITION> ::= <COMPOUND PROCEDURE HEAD>
<PROCEDURE BODY>

<PROCEDURE BODY> ::= <DECLARATION STATEMENT LIST>
<PROCEDURE STATEMENT LIST>
<PROCEDURE EXECUTABLE STATEMENT LIST>
<PROCEDURE ENDING>

Procedures are self-contained functional units within an SDL program which may be accessed according to specific rules discussed under "BASIC STRUCTURE OF THE SDL PROGRAM". Procedures may be created by preceding self-contained statements with a <COMPOUND PROCEDURE HEAD>, and terminating it with a <PROCEDURE ENDING>.

The <PROCEDURE DEFINITION> is composed of three basic parts: heading, body, and ending. Identifiers declared in a procedure may be accessed only in the procedure in which they are declared, and in procedures nested within the declaring procedure.

Procedures may be either "TYPED" or "NON-TYPED". A "TYPED" procedure returns some value of the type specified in the procedure declaration to the expression where the procedure was invoked. See "VALUE VARIABLES" for details. A "NON-TYPED" procedure performs a function, does not return a value, and is invoked in an <EXECUTE PROCEDURE STATEMENT>. See "EXECUTE PROCEDURE STATEMENT".

PROCEDURE HEAD

The syntax for the procedure heading is:

```
<COMPOUND PROCEDURE
HEAD> ::=                       <PROCEDURE HEAD>
                                <FORMAL PARAMETER DECLARATION
                                STATEMENT LIST>

<PROCEDURE HEAD> ::=            <BASIC PROCEDURE HEAD>
                                <PROCEDURE TYPE PART>

<BASIC PROCEDURE HEAD> ::=      <PROCEDURE NAME>
                                <FORMAL PARAMETER PART>

<PROCEDURE NAME> ::=            PROCEDURE <PROCEDURE IDENTIFIER>
                              | INTRINSIC <INTRINSIC IDENTIFIER>

<PROCEDURE IDENTIFIER> ::=      <TYPED PROCEDURE IDENTIFIER>
                              | <NON-TYPED PROCEDURE IDENTIFIER>

<TYPED PROCEDURE
IDENTIFIER> ::=                 <IDENTIFIER>

<NON-TYPED PROCEDURE
IDENTIFIER> ::=                 <IDENTIFIER>

<INTRINSIC IDENTIFIER> ::=      <TYPED INTRINSIC IDENTIFER>
                              | <NON-TYPED INTRINSIC IDENTIFER>

<TYPED INTRINSIC
IDENTIFIER> ::=                 <IDENTIFIER>

<NON-TYPED INTRINSIC
IDENTIFIER> ::=                 <IDENTIFIER>

<FORMAL PARAMETER PART> ::=     <EMPTY>
                              | (<FORMAL PARAMETER LIST>)

<FORMAL PARAMETER LIST ::=      <FORMAL PARAMETER>
                              | <FORMAL PARAMETER>,
                                <FORMAL PARAMETER LIST

<FORMAL PARAMETER> ::=          <IDENTIFIER>

<PROCEDURE TYPE PART> ::=       <EMPTY>
                              | <FORMAL TYPE PART>

<FORMAL TYPE PART> ::=          <TYPE PART>
                              | <TYPE VARYING PART>

<TYPE PART> ::=                 FIXED
```

```
                          | CHARACTER <FIELD SIZE>
                          | BIT <FIELD SIZE>
                          | REFERENCE

<FIELD SIZE> ::=            (<CONSTANT EXPRESSION>)

<TYPE VARYING PART> ::=     VARYING
                          | BIT VARYING
                          | CHARACTER VARYING

<FORMAL PARAMETER DECLA-
RATION STATEMENT LIST> ::=  <EMPTY>
                          | <FORMAL PARAMETER DECLARATION STATEMENT
                            LIST>;
                            <FORMAL PARAMETER DECLARATION>

<FORMAL PARAMETER
DECLARATION STATEMENT> ::=  FORMAL <FORMAL ELEMENT>
                          | FORMAL_VALUE <FORMAL ELEMENT>
                          | <FORMAL PARAMETER DECLARATION STATEMENT>,
                            <FORMAL ELEMENT>

<FORMAL ELEMENT> ::=        (<FORMAL IDENTIFIER LIST>)
                            <FORMAL TYPE PART>
                          | <FORMAL IDENTIFIER>
                            <FORMAL TYPE PART>

<FORMAL IDENTIFIER
LIST> ::=                   <FORMAL IDENTIFIER>
                          | <FORMAL IDENTIFIER LIST>,
                            <FORMAL IDENTIFIER>

<FORMAL IDENTIFIER> ::=     <COMPLEX IDENTIFIER>
                          | <VARYING ARRAY SPECIFIER>

<COMPLEX IDENTIFIER> ::=    <SIMPLE IDENTIFIER>
                          | <ARRAY IDENTIFIER>
                            <ARRAY BOUND>

<VARYING ARRAY
SPECIFIER> ::=              <ARRAY IDENTIFIER>
                            <VARYING ARRAY BOUND>

<VARYING ARRAY BOUND> ::=   (*)
```

The procedure heading, i.e., <COMPOUND PROCEDURE HEAD>, contains the <PROCEDURE NAME>, formal parameters (if any), and the <PROCEDURE TYPE PART>, i.e., the field type of the value to be returned if the procedure is "TYPED". For example:

```
PROCEDURE X (M,N) FIXED;
   FORMAL (M,N) VARYING;
```

which corresponds to the following syntax:

```
PROCEDURE <TYPED PROCEDURE IDENTIFIER>
          (<FORMAL PARAMETER>,<FORMAL PARAMETER>)
          <PROCEDURE TYPE PART>;
   FORMAL (<FORMAL IDENTIFIER>,<FORMAL IDENTIFIER>)
          <FORMAL TYPE PART>;
```

In this case, the value returned to the point of invocation should be fixed. There is, however, no check for this at compile time. If the control card option FORMAL_CHECK is present, the returned values will be checked against the procedure type at run time.

The "NON-TYPED" procedure follows the same format except that the <PROCEDURE TYPE PART> is omitted since no value is returned. For instance:

```
PROCEDURE A (J,K,L);
   FORMAL J FIXED, (K,L) BIT VARYING;
```

which syntactically is the same as:

```
PROCEDURE  <NON-TYPED PROCEDURE IDENTIFIER>
           (<FORMAL PARAMETER>,<FORMAL PARAMETER>,
           <FORMAL PARAMETER>);
   FORMAL <FORMAL IDENTIFIER> <FORMAL TYPE PART>,
          (<FORMAL IDENTIFIER>,<FORMAL IDENTIFIER>)
          <FORMAL TYPE PART>;
```

When a formal parameter is declared as FORMAL_VALUE, the actual parameter will always be passed by value. See the section on ADDRESS and VALUE PARAMETERS.

The field type of formal parameters (i.e., components of the <FORMAL TYPE PART>) may be static (BIT, CHARACTER, or FIXED) or variable (BIT VARYING, CHARACTER VARYING, or VARYING).

The <FIELD SIZE> must be a <CONSTANT EXPRESSION> (i.e., an expression whose value can be determined during compilation).

Often however, it is impossible to determine the data type at compile time especially if the actual parameters are passed to the procedure from different points in the program and under differing circumstances. SDL allows the user to specify variable data fields in the formal declaration. The actual parameters passed to that procedure will provide the specifics. Thus formals may be declared as "BIT VARYING", "CHARACTER VARYING", or "VARYING".

In a variable bit or character field, the type of data passed must be that which is specified (i.e., BIT or CHARACTER). The length, however, remains variable. Formals specified as "VARYING" may accept any type of data of any length.

The data types of corresponding formal and actual parameters will not be checked at compile time and only at run time when FORMAL.CHECK has been specified as a control card option.

Varying formals may be remapped, but it is the programmer's responsibility to ensure that the remapped formal parameter and its corresponding actual parameter match. A warning message will appear in the source listing where the remapping has occurred.

SDL also allows formally declared arrays to have a variable number of elements by substituting "*" for the number following the <ARRAY IDENTIFIER>. For instance:

```
PROCEDURE X (A,B);
   FORMAL A (*) FIXED, B (*) VARYING;
```

INTRINSIC HEAD

The word "INTRINSIC" may be used interchangeably with the word "PROCEDURE". It is, however, intended only for use by the SDL group in order to provide SDL intrinsics.

The use of "INTRINSIC" forces the intrinsic to have as entry point the displacement 0 within a new segment.

***

PROCEDURE BODY

The body of the procedure follows the heading. Included are declaration of local data (discussed under "THE DECLARATION STATEMENT"), nested procedures (also see "BASIC STRUCTURE OF THE SDL PROGRAM"), executable statements, and an ending. The syntax for the <PROCEDURE EXECUTABLE STATEMENT LIST> follows:

```
<PROCEDURE BODY> ::=          <DECLARATION STATEMENT LIST>
                              <PROCEDURE STATEMENT LIST>
                              <PROCEDURE EXECUTABLE STATEMENT LIST>
                              <PROCEDURE ENDING>

<PROCEDURE EXECUTABLE
STATEMENT LIST> ::=           <PROCEDURE EXECUTABLE STATEMENT>
                            | <PROCEDURE EXECUTABLE STATEMENT>
                              <PROCEDURE EXECUTABLE STATEMENT LIST>

<PROCEDURE EXECUTABLE
STATEMENT> ::=                <EXECUTABLE STATEMENT>
                            | <RETURN STATEMENT>
                            | <SEGMENT STATEMENT>
                              <PROCEDURE EXECUTABLE STATEMENT>
```

The <EXECUTABLE STATEMENT>s will be discussed in the section entitled "EXECUTABLE STATEMENTS". As indicated by the above syntax, executable statements within a procedure may be segmented. However, a procedure must end in the same segment in which it begins. For other segmentation restrictions see "THE SEGMENT STATEMENT".

The syntax for the <RETURN STATEMENT> is:

```
<RETURN STATEMENT> ::=        <TYPED PROCEDURE RETURN STATMENT>
                            | <NON-TYPED PROCEDURE RETURN STATEMENT>

<TYPED PROCEDURE
RETURN STATMENT> ::=          RETURN <EXPRESSION>

<NON-TYPED PROCEDURE
RETURN STATEMENT> ::=         RETURN
                            | RETURN_AND_ENABLE_INTERRUPTS
```

The <RETURN STATEMENT> takes one of two forms depending on the type of the procedure encompassing it. If the procedure is "TYPED", an <EXPRESSION> must be returned to the point of invocation. In a "NON-TYPED" procedure, only a simple return is needed. For expression specifications refer to the sections entitled "EXPRESSIONS" and "PRIMARIES".

Type checking on a <RETURN STATEMENT> is done only at run time when FORMAL.CHECK appears as a control card option.

Within any given procedure (at any lexic level), certain statements are nested within other statements and are accessed, much like a procedure, by an address generated by the larger statement. The most general nesting level is zero, and the nesting level of any statement appears on an SDL listing under the column "NL". The most common instance of statements occurring at Nesting Level 1 or greater are:

1. The conditionally executed statements following "THEN" and "ELSE" in the <IF STATEMENT>.

2. Statements contained within a <CASE STATEMENT>.

3. <DO-GROUP>s.

If the compiler cannot find a <RETURN STATEMENT> on NL 0, it will generate one directly preceding the <PROCEDURE ENDING>. This is merely a safety measure to insure that a procedure can always be properly exited.

A compiler-generated return works essentially in the same manner as an explicit return. In a non-typed procedure, control is returned to the point of the procedure's invocation. In a typed procedure, the following values are returned.

If the procedure is typed:	the compiler will return:
BIT	BITS CONTAINING 0 OF LENGTH SPECIFIED
CHARACTER	BLANKS OF LENGTH SPECIFIED
FIXED	FIXED ZERO
BIT VARYING	8-BITS OF ZERO
CHARACTER VARYING	ONE BLANK
VARYING	FIXED ZERO

RETURN_AND_ENABLE_INTERRUPTS is for MCP use only. It will cause a normal procedure exit to take place, and will enable interrupts as well.

* * *

PROCEDURE ENDING

The <PROCEDURE ENDING> is the final statement of a procedure, and the syntax is:

```
<PROCEDURE ENDING> ::=      END
                          | END <PROCEDURE IDENTIFIER>
```

The identifier following the reserved word "END" is optional. Its sole purpose is to simplify the documentation of the program. If an identifier is supplied by the user, the compiler will perform a syntax check to guarantee that the <PROCEDURE ENDING> is appropriately placed.

BURROUGHS CORPORATION
COMPUTER SYSTEMS GROUP
SANTA BARBARA PLANT

ASSIGNMENT STATEMENTS AND EXPRESSIONS

```
<ASSIGNMENT STATEMENT> ::=    <ADDRESS VARIABLE>
                              <REPLACE>
                              <EXPRESSION>

<ADDRESS VARIABLE> ::=        SEE "ADDRESS VARIABLES"

<REPLACE> ::=                 :=

<EXPRESSION> ::=              <STRING EXPRESSION>
                            | <STRING EXPRESSION>
                              CAT <EXPRESSION>

<STRING EXPRESSION> ::=       <LOGICAL FACTOR>
                            | <LOGICAL FACTOR>
                              <OR-ING OPERATOR>
                              <STRING EXPRESSION>

<OR-ING OPERATOR> ::=         OR  |  EXOR

<LOGICAL FACTOR> ::=          <LOGICAL SECONDARY>
                            | <LOGICAL SECONDARY>
                              AND <LOGICAL FACTOR>

<LOGICAL SECONDARY> ::=       <LOGICAL PRIMARY>
                            | NOT <LOGICAL PRIMARY>

<LOGICAL PRIMARY> ::=         <ARITHMETIC EXPRESSION>
                            | <ARITHMETIC EXPRESSION>
                              <RELATION>
                              <ARITHMETIC EXPRESSION>

<RELATION> ::=                < | <= | = | /= | >= | > |
                              LSS  |  LEQ  |  EQL  |  NEQ  |
                              GEQ  |  GTR

<ARITHMETIC
EXPRESSION> ::=               <TERM>
                            | <TERM>
                              <ADDITIVE OPERATOR>
                              <ARITHMETIC EXPRESSION>

<ADDITIVE OPERATOR> ::=       + | -

<TERM>::=                     <SIGNED PRIMARY>
                            | <SIGNED PRIMARY>
                              <MULTIPLICATIVE OPERATOR>
                              <TERM>

<MULTIPLICATIVE
OPERATOR> ::=                 * | MOD | /
```

```
<SIGNED PRIMARY>::=     <PRIMARY>
                      | <UNARY OPERATOR>
                        <PRIMARY>

<UNARY OPERATOR> ::=    + | -
```

The following is a list of the SDL operators from highest precedence to lowest. This list or the table in Figure 3 may be used when evaluating an expression.

```
+ , - (<UNARY OPERATOR>)
*, /, MOD
+, - (<ADDITIVE OPERATOR>)
<, /=, =, <=, >=, >
NOT
AND
OR, EXOR
CAT
```

1. The assignment operator has higher precedence than any operator to its left and lower precedence than any to its right.

2. The order of evaluation of operators having equal precedence is always from left to right.

PREVIOUS OP. \ PRESENT OP.	NEG	*	+ -	=	NOT	AND	OR	CAT	:=	(	)	ET
NEG	>	>	>	>	<	>	>	>	<	<	>	>
*	<	>	>	>	<	>	>	>	<	<	>	>
+ -	<	<	>	>	<	>	>	>	<	<	>	>
=	<	<	<	>	<	>	>	>	<	<	>	>
NOT	<	<	<	<	>	>	>	>	<	<	>	>
AND	<	<	<	<	<	>	>	>	<	<	>	>
OR	<	<	<	<	<	<	>	>	<	<	>	>
CAT	<	<	<	<	<	<	<	>	<	<	>	>
:=	<	<	<	<	<	<	<	<	<	<	>	>
(	<	<	<	<	<	<	<	<	<	<	=	
)		>	>	>		>	>	>	>		>	>
BT	<	<	<	<	<	<	<	<	<	<		=

FORMULA: PRECEDENCE <PREVIOUS OP> <RELATION> PRECEDENCE <PRESENT OP>

NOTE:
NEG	UNARY OPERATORS
*	MULTIPLICATIVE OPERATORS
=	RELATIONAL OPERATORS
:=	REPLACE OPERATORS
BT	INFERRED BEGINNING TERMINATOR
ET	INFERRED ENDING TERMINATOR

Fig 3. Operator Precedence Table

UNARY OPERATORS

+
-

The unary operator acts upon one operand and may never appear as an infix operator between two operands. It may appear to the right of any other operator, including itself.

The UNARY MINUS (-) generates the two's complement of the operand associated with it (i.e., -X = (NOT X)+1). The operand may be any data type. If it is fixed, the UNARY MINUS has the effect of reversing the sign, and the result is labeled on the Evaluation Stack as fixed.

If the operand is either a character or bit string, only the low-order 24 bits will be evaluated. Strings less than 24 bits will be padded with leading zeroes to 24 bits. The two's complement of the string is generated and returned to the stack as type fixed.

The SDL compiler generates no code for the unary plus (+) which exists solely for the convenience of the programmer.

ARITHMETIC OPERATORS

+	Addition
-	Subtraction
*	Multiplication
MOD	Division yielding integer value of remainder
/	Division yielding integer value of quotient

The arithmetic operators perform 24-bit arithmetic on two operands of any of the three data types. Sign analysis will be done only if both operands are fixed. With any other combination of data types, the magnitudes of the operands are evaluated.

For both bit and character data, if the field is greater than 24 bits, only the low-order 24 bits will be evaluated. If the field is less than 24 bits, leading zeroes will be supplied from the left.

A 24-bit result will be returned to the Evaluation Stack. If both operands are fixed, the result will be fixed. Otherwise, the result will be type bit.

SDL division results in an integer value. Any remainder is truncated thus:

```
7 / 3 = 2
3 / 7 = 0
```

Note this means that "*" and "/" do not associate. In general, (A * B) / C does not equal A * (B / C).

The MOD operation is division resulting in the integer value of the remainder. It is evaluated by the following formula:

Y MOD Z = Y-(Z*(Y/Z)) using integer division explained above.

For example:

```
 7 MOD  3 = 7-(3 * 2) = 1
-7 MOD  3 = -7-(3*(-2)) = -1
 3 MOD -7 = 3-((-7)*(-0)) = 3
-3 MOD -7 = (-3)-((-7) * 0) = -3
```

Note: For negative arguments, this definition is not the same as the traditional definitions from mathematics.

RELATIONAL OPERATORS

=	EQL	EQUAL TO
/=	NEQ	NOT EQUAL TO
>	GTR	GREATER THAN
<	LSS	LESS THAN
>=	GEQ	GREATER THAN OR EQUAL TO
<=	LEQ	LESS THAN OR EQUAL TO

The relational operators do a comparison between two operands of any data type. A 1-bit result is returned -- @(1)1@ if the condition is true, @(1)0@ if the condition is false.

If both operands are fixed, the operator does a true signed compare. If both operands are character strings, the shorter one is padded on the right with blanks, and a character by character magnitude compare by collating sequence is done.

For all other operand combinations, leading zeroes are supplied to the shorter of the two. No sign analysis is done, and operands are treated as positive magnitudes.

LOGICAL OPERATORS

NOT
AND
OR
EXOR

The logical operators perform a bit by bit analysis on all three data types. NOT is considered to be a unary operator, and may appear to the right of any other operator (including itself).

The other operators require two operands. The shorter of the two is padded on the left with zeroes to duplicate the length of the larger. The following chart illustrates the use of each operator.

IF X =	0	0	1	1
IF Y =	0	1	0	1
NOT X =	1	1	0	0
NOT Y =	1	0	1	0
X AND Y =	0	0	0	1
X OR Y =	0	1	1	1
X EXOR Y =	0	1	1	0

REPLACE OPERATORS

```
<ASSIGNMENT STATEMENT> ::=   <ADDRESS VARIBLE>
                             <REPLACE>
                             <EXPRESSION>

<REPLACE> ::=                :=

<ASSIGNOR> ::=               <ADDRESS VARIABLE>
                             <NON-DESTRUCTIVE REPLACE>
                             <EXPRESSION>

<NON-DESTRUCTIVE
REPLACE> ::=                 <REPLACE, DELETE LEFT PART>
                           | <REPLACE, DELETE RIGHT PART>

<REPLACE, DELETE
LEFT PART> ::=               :=

<REPLACE, DELETE
RIGHT PART> ::=              ::=
```

NOTE: <REPLACE, DELETE RIGHT PART> symbol ":::=" is the same as the BNF definition symbol.

There are two basic types of replace operators: The destructive <REPLACE> associated with the <ASSIGNMENT STATEMENT>, and the <NON-DESTRUCTIVE REPLACE> which occurs only within an expression.

The destructive <REPLACE> operator causes the expression on its right to "REPLACE" the variable on its left. The Evaluation Stack is flushed since this replace is necessarily the last operation in the statement.

The <NON-DESTRUCTIVE REPLACE> takes two forms: "DELETE LEFT" and "DELETE RIGHT". The "DELETE LEFT" causes the expression to the right of the operator to replace the variable on its left. The variable is then deleted from the top of the Evaluation Stack, and the expression is left on the top of the stack.

The "DELETE RIGHT" causes the same replacement. However, the expression to the right of the operator is deleted from the Evaluation Stack, and the variable to the left remains on the top of the stack.

The following example illustrates the use of the <NON-DESTRUCTIVE REPLACE>:

```
PROCEDURE GOOD  BIT VARYING;
     DECLARE X  BIT(48);
     RETURN X ::= "RESULT";
END GOOD;
PROCEDURE BAD  BIT VARYING;
     DECLARE Y  BIT(48);
     RETURN Y := "RESULT";
END BAD;
```

PROCEDURE GOOD will execute properly since X, declared as bit, is associated with the procedure type--bit varying. Notice, however, that in PROCEDURE BAD, Y is deleted from the stack and the character string "RESULT" remains. Unless the control card option FORMAL.CHECK is set at compile time, there will be no indication that the data types (as in PROCEDURE BAD) do not match the procedure type. If FORMAL.CHECK is specified, the following execute time error message will be printed:

"TYPE ERROR IN RETURNED VALUE"

If both operands associated with any replace operator are character fields, and the receiving field is longer than the sending field, trailing blanks will be added. If the receiving field is shorter, characters will be truncated from the right.

With every other combination of data types, when the receiving field is not equal in length to the sending field, leading binary zeroes will be appended to the larger receiving field, or high-order bits are truncated from the larger sending field.

Inconsistant results may be obtained in cases such as

```
A:=SUBSTR (A,2,5)
```

(i.e., where the sending field and the receiving field are simple primaries less than 24 bits apart). This problem can be avoided by enclosing the SUBSTR in parentheses.

```
A:= (SUBSTR(A,2,5));
```

Also see the reverse store operation in the section entitled "EXECUTE-FUNCTION STATEMENT".

CONCATENATION

Data items may be linked together (concatenated) by using the "CAT" operator. Although this operator is intended to concatenate bit strings or character strings, it may be used with any combination of data types. The result of any concatenation may not be greater than 8,191 characters or 65,535 bits.

If all the operands are character strings, the result is a character string. For any other combination of data types, the result is a bit string. For example:

```
 LET A = "B"                          1 CHARACTER
     B = @(1)101@                     3 BITS
     C = 10                           FIXED
THEN
     B CAT B = @(1)101101@            BIT STRING, LENGTH 6
     A CAT A = "BB"                   CHARACTER STRING, LENGTH 2
     A CAT B = @(1)110000101011@      BIT STRING, LENGTH 11
     B CAT C = @(3)500000012@         BIT STRING, LENGTH 27
                                      (EXPRESSED IN OCTAL)
```

PRIMARY ELEMENTS OF THE EXPRESSION

```
<PRIMARY> ::=          <CONSTANT>
                     | <VARIABLE>
                     | (<EXPRESSION>)
                     | <CONDITIONAL EXPRESSION>
                     | <CASE EXPRESSION>
                     | <BUMPOR>
                     | <DECREMENTOR>
                     | <ASSIGNOR>

<VARIABLE> ::=         <ADDRESS VARIABLE>
                     | <VALUE VARIABLE>
```

A primary is the most basic component of the SDL expression. To avoid unnecessary repetition, see "BASIC COMPONENTS OF THE SDL LANGUAGE" for discussion of constants, and see "ADDRESS VARIABLES" and "VALUE VARIABLES" for discussion of variables.

CONDITIONAL EXPRESSION

```
<CONDITIONAL EXPRESSION> ::= IF <EXPRESSION>
                             THEN <EXPRESSION>
                             ELSE <EXPRESSION>
```

The expression following the reserved word "IF" is evaluated. If the low-order bit of the result is 1, the expression following "THEN" is evaluated. Otherwise, the expression following "ELSE" is evaluated. Unlike the <IF STATEMENT>, the "ELSE" part of the expression must be present.

CASE EXPRESSION

```
<CASE EXPRESSION> ::=        CASE <EXPRESSION>
                             OF (<EXPRESSIONLIST>)

<EXPRESSION LIST> ::=          <EXPRESSION>
                             | <EXPRESSION>,
                               <EXPRESSION LIST>
```

In the <CASE EXPRESSION>, the value of the <EXPRESSION> following the reserved word "CASE" is used as an index into the list of expressions. The expression thus selected is evaluated, and the other expressions in the list ignored. The range of the index is from zero to N-1, where N is the number of <EXPRESSION>s in the list. An example of an <ASSIGNMENT STATEMENT> containing a <CASE EXPRESSION> follows:

```
A:=CASE I OF (A+B, A-B, A*B, A/B, A MOD B) +
   CASE J OF (Q*F-6, 9, 34+B, (A+3) MOD 9, C)
```

if I=2 and J=3, the statement will be evaluated as follows:

```
A:=(A*B) + (A+B) MOD B;
```

BUMP

```
<BUMPOR> ::=                 BUMP <ADDRESS VARIABLE>
                             <MODIFIER>

<MODIFIER> ::=                 <EMPTY>
                             | BY <EXPRESSION>
```

BUMPOR leaves on the Evaluation Stack, a descriptor of the variable which has been incremented by the value of the modifying <EXPRESSION>. If <MODIFIER> is <EMPTY>, then the variable is incremented by 1. The results of the following expressions (where A is an <ARRAY IDENTIFIER>) are equivalent:

```
BUMP A(X+Y) BY N
A(X+Y) ::= A(X+Y) + N
```

The advantage of using <BUMPOR> is that the code for putting the descriptor on the stack is executed only once. Thus it is more efficient.

Like any variable, (<BUMPOR>) will cause a value to be loaded to the top of the stack. Hence:

```
P(BUMP X BY C-D);
```

passes X by address but,

```
P((BUMP X BY C-D));
```

passes X by value.

<BUMPOR> operates on all three data types. Character strings are treated as if they were bit strings. For fields greater than 24 bits, only the low-order 24 bits are evaluated. If the field is less than 24 bits, it is padded with leading zeroes to 24 bits.

DECREMENT

<DECREMENTOR> ::= DECREMENT <ADDRESS VARIABLE> <MODIFIER>

<MODIFIER> ::= <EMPTY>
| BY <EXPRESSION>

The <DECREMENTOR> works exactly like <BUMPOR> except that the variable is decreased by the value of the <EXPRESSION>. See above.

ASSIGNOR

<ASSIGNOR> ::= See REPLACE OPERATORS in Chapter 7.

ADDRESS VARIABLES

```
<ADDRESS VARIABLE> ::=      <SIMPLE VARIABLE>
                          | <SUBSCRIPTED VARIABLE>
                          | <INDEXED VARIABLE>
                          | <ADDRESS-GENERATING FUNCTION DESIGNATOR>

<SIMPLE VARIABLE> ::=       <SIMPLE IDENTIFIER>

<SIMPLE IDENTIFIER> ::=     <IDENTIFIER>

<SUBSCRIPTED VARIABLE> ::=  <ARRAY IDENTIFIER>(<EXPRESSION>)

<ARRAY IDENTIFIER> ::=      <IDENTIFIER>
```

As noted above, <ADDRESS VARIABLE>s may take the form of a <SIMPLE IDENTIFIER>, or an <ARRAY IDENTIFIER> followed by an (<EXPRESSION>) designating the array element in question. In addition, simple and array identifiers may be indexed.

INDEXING

```
<INDEXED VARIABLE> ::=      <SIMPLE IDENTIFIER> <INDEX PART>
                          | <ARRAY IDENTIFIER> <INDEX PART>

<INDEX PART> ::=            [<EXPRESSION LIST>]
```

Each of the expressions in the <INDEX PART> is evaluated, and the sum of these is formed. This will be called the index.

The indexing operation occurs functionally as follows:

1. The simple or array descriptor is loaded to the top of the Evaluation Stack.
2. If the descriptor is an array descriptor, then it is converted to a simple descriptor which describes the first (zero) element of the array.
3. The address field of the descriptor is modified by adding to it the index.

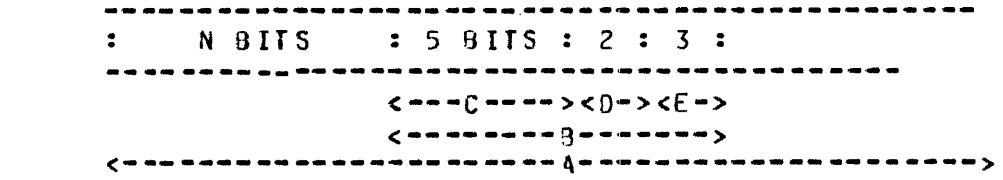
Note that self-relative data items (i.e., data items whose length is not greater than 24, which are not in a structure, and which do not remap some other data item) may not be indexed.

There are two methods of indexing:

1. The descriptor provides the address, and the index provides the offset from this address.

2. The descriptor provides the offset, and the index provides the address.

Example:



Field D may be accessed using either method (1) or method (2). Assume N contains the offset to B.

Method (1):

```
DECLARE
     01  A BIT(5000),
         02  B,
             03  C BIT(5),
             03  D BIT(2),
             03  E BIT(3),
     N BIT(24),
     X BIT(2);
/* THE NEXT STATEMENT WILL MOVE D (WITH THE OFFSET
   GIVEN BY N) INTO X */
X = D[N];
```

Method (2):

```
DECLARE
     A BIT(5000),
     01  BB REMAPS BASE,
          02  CC BIT(5),
          02  DD BIT(2),
          02  EE BIT(3),
     N BIT(24),
     X BIT(2);
/* THE NEXT STATEMENT WILL MOVE DD
   (WITH THE OFFSET GIVEN BY N) INTO X */
X = DD[N, DATA_ADDRESS(A)];
```

Note the following:

1. The structure above, comprised of BB, CC, DD, and EE, which remaps base is called a "template".

2. This template may be applied to any data area merely by providing the address as part of the index. This is not the case when method(1) indexing is used.

3. The example above is contrived --in method (2), if N contained the address of B rather than the offset to B from the beginning of A, then the statements which store D into X would be identical: X = DD[N];

ADDRESS GENERATING FUNCTIONS

```
<ADDRESS-GENERATING
FUNCTION DESIGNATOR> ::=    <SUB-STRING ADDRESS DESIGNATOR>
                          | <FETCH COMMUNICATE MESSAGE
                            POINTER DESIGNATOR>
                          | <DESCRIPTOR DESIGNATOR>
                          | <DESCRIPTOR-GENERATOR DESIGNATOR>
                          | <ADDRESS-MODIFIER DESIGNATOR>
                          | NULL
```

SUBBIT AND SUBSTR

```
<SUB-STRING ADDRESS
DESIGNATOR> ::=             <SUB-STRING FUNCTION IDENTIFIER>
                            (<STRING ADDRESS>,<OFFSET PART>)
                          | <SUB-STRING FUNCTION IDENTIFIER>
                            (<STRING ADDRESS>,<OFFSET PART>,
                            <LENGTH PART>)

<SUB-STRING FUNCTION
IDENTIFIER> ::=             SUBBIT | SUBSTR

<STRING ADDRESS> ::=        <ADDRESS GENERATOR>

<ADDRESS GENERATOR> ::=     SEE "ADDRESS GENERATOR"

<OFFSET PART> ::=           <EXPRESSION>

<LENGTH PART> ::=           <EXPRESSION>
```

SUBSTR yields a sub-string of a character string identified by the <STRING ADDRESS>. The beginning character of the sub-string is specified by the <OFFSET PART> (where the first character of the string is zero). The <LENGTH PART> specifies the length of the sub-string. If omitted, the rest of the string from the "OFFSET" character is assumed. For example:

```
If X = "CHARACTER"
   C = "COALITION"

then

   SUBSTR(X,4) := SUBSTR(C,0,4)
```

yields the character string:

"CHARCOAL "

Like all character-to-character store operations, if the receiving field is larger than the sending field, the sending field is padded with blanks on the right. If the sending field is longer, characters are truncated from the right. Note that this is a function of the store operator and not substr.

SUBBIT yields a sub-string of a bit string identified by the <STRING ADDRESS>. The beginning bit of the sub-string is specified by the <OFFSET PART> (Note: The first bit of the string is 0). The length of the sub-string is specified by the <LENGTH PART> which, if omitted, will be assumed to be the rest of the string.

EXAMPLE:

If A = @(1)00101011101@
B = @(1)00001111101@

then

SUBBIT(A,2,3) CAT SUBBIT(B,5)

results in:

@(1)101111101@

and

SUBBIT(A,3) CAT SUBBIT(B,0,6)

results in:

@(1)010110100001 1@

FETCH_COMMUNICATE_MSG_PTR

<FETCH COMMUNICATE MESSAGE POINTER DESIGNATOR> ::= FETCH_COMMUNICATE_MSG_PTR

See the B1700 MCP Reference Manual for a description of the run structure.

If the RS_MCP_BIT is set, then RS_COMMUNICATE_MSG_PTR is accessed. Otherwise, RS_REINSTATE_MSG_PTR is accessed. The accessed field is assumed to be a descriptor and is placed on the top of the Evaluation Stack.

EXAMPLE:

```
DESCRIPTOR(COMM_MSG) :=
    VALUE.DESCRIPTOR(FETCH_COMMUNICATE_MSG_PTR);
```

COMM_MSG now describes the communicate message, assuming that the message was described by a non-self-relative descriptor.

DESCRIPTORS

<DESCRIPTOR DESIGNATOR>::= DESCRIPTOR (<SIMPLE IDENTIFIER>)
| DESCRIPTOR (<ARRAY IDENTIFIER>)

"DESCRIPTOR" places on the Evaluation Stack, a descriptor which describes the descriptor of a <SIMPLE IDENTIFIER> or an <ARRAY IDENTIFIER>. The descriptor function may appear as the object of a replacement, thereby providing easy access to any part of a descriptor.

EXAMPLE:

1. SUBBIT(DESCRIPTOR(X),4,2) := 2;

2. DESCRIPTOR(X) := DESCRIPTOR(Y);

Example (2) forces both X and Y to describe the same data name. Note, however, that if X and Y are not either both simple items or both arrays, the result will be incorrect.

MAKE_DESCRIPTOR

<DESCRIPTOR-GENERATOR DESIGNATOR> ::= MAKE_DESCRIPTOR(<EXPRESSION>)

The value which is generated by the <EXPRESSION> is assumed to be a descriptor. This descriptor replaces on the Evaluation Stack, the descriptor representing that <EXPRESSION>. If the name-value bit of the expression's descriptor on the Evaluation Stack is set, then the value of the <EXPRESSION> is removed from the Value Stack.

A <DESCRIPTOR GENERATOR DESIGNATOR> may appear as the object of a replacement, however the programmer is responsible to see that the descriptor built generates an address. There is no syntax check for this.

The following examples illustrate the relationships between the descriptor functions:

DESCRIPTOR(X)=VALUE_DESCRIPTOR(X),
where X is non-self-relative

MAKE_DESCRIPTOR (DESCRIPTOR(X)) = X,
where X is non-self-relative

MAKE_DESCRIPTOR (VALUE_DESCRIPTOR(E)) = E,
where E is an <ADDRESS GENERATOR>

VALUE_DESCRIPTOR (MAKE_DESCRIPTOR(E)) = E,
where the value of E is a valid <ADDRESS GENERATOR>

NEXT_ITEM, PREVIOUS_ITEM

<ADDRESS-MODIFIER DESIGNATOR> ::= <ADDRESS-MODIFIER FUNCTION IDENTIFIER> (<SIMPLE IDENTIFIER>)

<ADDRESS-MODIFIER FUNCTION IDENTIFIER> ::= NEXT_ITEM
| PREVIOUS_ITEM

The NEXT_ITEM function causes the length field of the descriptor represented by the <SIMPLE IDENTIFIER> to be added to the address field of that descriptor. This modified descriptor is put back onto the Name Stack, and also moved to the top of the Evaluation Stack. Moving the modified descriptor to the Evaluation Stack is, in effect, a load address of the new item described by the <SIMPLE IDENTIFIER>. Hence, "NEXT_ITEM" may be used as the object of a replacement. For example, the following statements:

```
DECLARE 01 CHAR_STRING CHARACTER(1000),
        02 NEXT_CHAR CHARACTER(1);
NEXT_ITEM (NEXT.CHAR)!"D";
```

have the effect of storing "D" into the second character of CHAR_STRING, which is:

```
SUBSTR(CHAR_STRING,1,1);
```

The PREVIOUS_ITEM function is identical to NEXT_ITEM except that a subtraction (of length from address) is performed.

NULL

This function generates an address of type character with zero length. A store into this address is essentially a no-op. NULL is used primarily in conjunction with the REFER statement.

ADDRESS GENERATORS

```
<ADDRESS
GENERATOR LIST> ::=          <ADDRESS GENERATOR>
                           | <ADDRESS GENERATOR>,
                             <ADDRESS GENERATOR LIST>

<ADDRESS GENERATOR> ::=      <ADDRESS VARIABLE>
                           | <BUMPOR>
                           | <DECREMENTOR>
                           | <CONDITIONAL ADDRESS GENERATOR>
                           | <CASE ADDRESS GENERATOR>
                           | <ADDRESS-GENERATING ASSIGNOR>

<BUMPOR> ::=                 See "BUMPOR"

<DECREMENTOR> ::=            See "DECREMENTOR"

<CONDITIONAL ADDRESS
GENERATOR> ::=               IF <EXPRESSION>
                             THEN <ADDRESS GENERATOR>
                             ELSE <ADDRESS GENERATOR>

<CASE ADDRESS
GENERATOR> ::=               CASE <EXPRESSION>
                             OF (<ADDRESS GENERATOR LIST>)

<ADDRESS-GENERATING
ASSIGNOR> ::=                <ADDRESS VARIABLE>
                             <REPLACE, DELETE LEFT PART>
                             <ADDRESS GENERATOR>
                           | <ADDRESS VARIABLE>
                             <REPLACE, DELETE RIGHT PART>
                             <EXPRESSION>
```

The <ADDRESS GENERATOR> includes any primary which leaves an address on the top of the Evaluation Stack. See "PRIMARY ELEMENTS OF THE EXPRESSION" for more explicit detail.

VALUE VARIABLES

```
<VALUE VARIABLE> ::=          <VALUE-GENERATING FUNCTION DESIGNATOR>
                            | <TYPED PROCEDURE DESIGNATOR>
                            | (<ADDRESS VARIABLE>)
                            | <FILE DESIGNATOR>

<FILE DESIGNATOR> ::=         <FILE IDENTIFIER>
                            | <SWITCH FILE IDENTIFIER>(<EXPRESSION>)

<TYPED PROCEDURE
DESIGNATOR> ::=               <TYPED PROCEDURE IDENTIFIER>
                              <ACTUAL PRAMETER PART>

<TYPED PROCEDURE
IDENTIFIER> ::=               <IDENTIFIER>

<ACTUAL PARAMETER PART> ::=   <EMPTY>
                            | (<ACTUAL PARAMETER LIST>)

<ACTUAL PARAMETER LIST> ::=   <ACTUAL PARAMETER>
                            | <ACTUAL PARAMETER>,
                              <ACTUAL PARAMETER LIST>

<ACTUAL PARAMETER> ::=        <EXPRESSION>
                            | <ARRAY DESIGNATOR>

<ARRAY DESIGNATOR> ::=        <ARRAY IDENTIFIER>

<ARRAY IDENTIFIER> ::=        <IDENTIFIER>
```

Notice from the above syntax that any <ADDRESS VARIABLE> enclosed in parens, such as (SUBBIT (A,I,J)), will be treated as a value variable.

The value generated by a <FILE DESIGNATOR> is the FPB number of the specified file. A warning message will be issued when a <FILE DESIGNATOR> is used as a value, i.e., not in an I/O statement.

TYPED PROCEDURES

The TYPED procedure (a procedure which returns a value) is invoked within an expression according to the above syntax. The procedure identifier, followed by its parameters (if any), enclosed within parens, is treated as an operand in the expression. For details on passing parameters, see ADDRESS AND VALUE PARAMETERS. The procedure is evaluated and the returned value replaces the <TYPED PROCEDURE DESIGNATOR>. For example:

```
DECLARE Z FIXED;
PROCEDURE X(A,B) FIXED;
     FORMAL (A,B) FIXED;
               .
               .
               .
END X;
Z := X(BUMP M,R)+1;
```

ADDRESS AND VALUE PARAMETERS

Actual parameters may be passed to a procedure either by address (which passes the address of the actual parameter) or by value (which passes a duplicate copy of the actual parameter).

If an <ACTUAL PARAMETER> (See VALUE VARIABLES and EXECUTE-PROCEDURE STATEMENT) is passed by address, then any change to the corresponding <FORMAL PARAMETER> in the procedure will result in a change to the original value of the <ACTUAL PARAMETER>.

If a parameter is passed by value, then only the duplicate copy of the <ACTUAL PARAMETER> can be changed. The original value remains unaltered, and the duplicate copy is erased when the procedure is exited.

An <ACTUAL PARAMETER> may be any expression or an <ARRAY IDENTIFIER>. SDL has specified that array identifiers may only be passed by address. An array element, however, may be passed either by address or by value.

Expressions may be divided into two groups:

1. Those which may be passed either by address or by value, and

2. Those which may only be passed by value.

An <ADDRESS GENERATOR> is passed by address unless it is enclosed within parentheses, or unless the formal parameter to which it corresponds has been declared as FORMAL_VALUE. In these two cases <ADDRESS GENERATOR>s will be loaded by value. All other expressions are loaded by value only.

Examples of parameters passed by address:

```
P(BUMP X, A)
P(B(BUMP M), SUBBIT(X,5))
P(NEXT_ITEM(8), A:IC+D)
```

Examples of parameters passed by value:

```
P((BUMP X), (A), 3)
P((B(BUMP M)), A+B)
P(SWAP(A,D), (SUBSTR(A,5,3)))
```

VALUE GENERATING FUNCTIONS

```
<VALUE-GENERATING
FUNCTION DESIGNATOR> ::=    <BASE REGISTER DESIGNATOR>
                          | <BINARY CONVERSION DESIGNATOR>
                          | <BINARY_SEARCH DESIGNATOR>
                          | <COMMUNICATE WITH GISMO FUNCTION>
                          | <CONSOLE SWITCHES DESIGNATOR>
                          | <CONTROL STACK BITS DESIGNATOR>
                          | <CONTROL STACK TOP DESIGNATOR>
                          | <CONVERT DESIGNATOR>
                          | <DATA ADDRESS DESIGNATOR>
                          | <DATA_LENGTH DESIGNATOR>
                          | <DATA_TYPE DESIGNATOR>
                          | <DATE FUNCTION DESIGNATOR>
                          | <DECIMAL CONVERSION DESIGNATOR>
                          | <DELIMITED TOKEN DESIGNATOR>
                          | <DISPATCH DESIGNATOR>
```

| <DISPLAY BASE DESIGNATOR>
| <DYNAMIC MEMORY BASE DESIGNATOR>
| <EVALUATION STACK TOP DESIGNATOR>
| <EXECUTE OPERATOR FUNCTION>
| <EXTENDED ARITHMETIC FUNCTION>
| <HASH CODE DESIGNATOR>
| <INTERROGATE INTERRUPT STATUS DESIGNATOR>
| <LAST LIO STATUS DESIGNATOR>
| <LENGTH DESIGNATOR>
| <LIMIT REGISTER DESIGNATOR>
| <LOCATION DESIGNATOR>
| <NAME-OF-DAY FUNCTION DESIGNATOR>
| <NAME STACK TOP DESIGNATOR>
| <NEXT TOKEN DESIGNATOR>
| <PARITY_ADDRESS DESIGNATOR>
| <PROCESSOR_TIME FUNCTION DESIGNATOR>
| <PROGRAM_SWTICHES DESIGNATOR>
| <SEARCH_LINKED_LIST DESIGNATOR>
| <SEARCH_SDL_STACKS DESIGNATOR>
| <SEARCH SERIAL LIST DESIGNATOR>
| <MEMORY SIZE DESIGNATOR>
| <SORT_SEARCH DESIGNATOR>
| <SORT_STEP_DOWN DESIGNATOR>
| <SORT_UNBLOCK DESIGNATOR>
| <SPO INPUT PRESENT DESIGNATOR>
| <SUB_STRING VALUE DESIGNATOR>
| <SWAP DESIGNATOR>
| <TIME FUNCTION DESIGNATOR>
| <TIMER DESIGNATOR>
| <DESCRIPTOR_VALUE_GENERATOR DESIGNATOR>
| <WAIT FUNCTION>

BASE_REGISTER

<BASE REGISTER DESIGNATOR> ::= BASE_REGISTER

A value of type BIT(24) is returned. The value is the absolute address of the base of the program. It should be noted that two separate executions of BASE_REGISTER may not yield the same results, since the MCP may have moved the program in memory.

BINARY CONVERSION

<BINARY CONVERSION DESIGNATOR> ::= BINARY (<EXPRESSION>)

The <BINARY CONVERSION DESIGNATOR> returns a fixed value which is the binary representation of the <EXPRESSION>. The <EXPRESSION> is assumed to be a character string containing decimal digits. Only the low-order 8 characters will be converted. Zone bits are ignored.

If the conversion results in a binary value greater than 24 bits (i.e., if the decimal number is greater than 16,777,215), then the left-most bits will be truncated.

If the decimal number is greater than 8,388,607 (i.e., (2 exp 23)-1), then the returned value will appear to be negative (i.e., the high-order bit is 1).

BINARY SEARCH

<BINARY_SEARCH FUNCTION>::= BINARY_SEARCH
(<START_RECORD>, <COMPARE_FIELD>,
<COMPARE_VALUE>, <NUMBER_OF_RECORDS>)

<START_RECORD>::= <EXPRESSION>
<COMPARE_FIELD>::= <TEMPLATE>
<COMPARE_VALUE>::= <EXPRESSION>
<NUMBER_OF_RECORDS>::= <ADDRESS GENERATOR>

BINARY_SEARCH searches an ordered list of items that start at (START_RECORD> for <NUMBER_OF_RECORDS> for a match.

The occurrence number of the entry that matches will be returned, or if there is no match, the occurrence number of the first entry that is greater will be returned.

Note: The comparison is always left justified and uses the length of <COMPARE VALUE>.

COMMUNICATE_WITH_GISMO

<COMMUNICATE WITH GISMO FUNCTION> ::= COMMUNICATE_WITH_GISMO (<EXPRESSION>)

The value of the operand is made non-self-relative by pushing its value to the Value Stack, if necessary. The absolute address of the value is copied into the T-register, and the length is copied into the L-register. The proper swapper value is put into the X-register and control is passed to GISMO. Any value returned by GISMO will be described by the same descriptor on the Evaluation Stack as was used to pass a value to GISMO. COMMUNICATE_WITH_GISMO may be used either as a statement or as a function.

CONSOLE_SWITCHES

<CONSOLE SWITCHES DESIGNATOR>::= CONSOLE_SWITCHES

Note: This function has meaning only B1720-series systems. It leaves on the top of the Evaluation Stack a 24-bit, self-relative value of the 24 console switches.

CONTROL_STACK_BITS

<CONTROL STACK BITS DESIGNATOR>::= CONTROL_STACK_BITS

This function leaves on the top of the Evaluation Stack a 24-bit, self-relative value of type bit which is the number of bits left in the control stack until overflow.

CONTROL_STACK_TOP

<CONTROL STACK TOP DESIGNATOR> ::= CONTROL_STACK_TOP

A value of type BIT(24) is returned. The value is the base relative address of the next entry to be placed on the control stack.

CONVERT

```
<CONVERSION DESIGNATOR> ::=  CONVERT (<EXPRESSION>,
                             <CONVERSION PART>)
                           | CONV (<EXPRESSION>,
                             <CONVERSION PART>)

<CONVERSION PART> ::=        <CONVERSION TYPE>
                           | <CONVERSION TYPE>,
                             <BIT GROUP SIZE>

<CONVERSION TYPE> ::=        BIT | CHARACTER | FIXED

<BIT GROUP SIZE> ::=         1 | 2 | 3 | 4
```

The <EXPRESSION>, which may be of any data type, will be converted as specified by the <CONVERSION TYPE>. The converted <EXPRESSION> will be returned as a value.

The <BIT GROUP SIZE> is used only with bit-to-character or character-to-bit conversions. It specifies the number of bits (of the bit string) which correspond to a character in the character string.

Note: Bit-to-character conversion does not yield decimal digits. If a bit string is to be converted to decimal digits, it should be stored in a fixed variable, and the fixed variable converted.

The following table shows the possible conversion combinations:

INPUT: \ OUTPUT:	BIT	CHARACTER	FIXED
BIT	No change	Convert to CHAR. under control of <BIT GROUP SIZE>; if omitted use 4	Return 24 BITS providing lead- ing zeroes or left truncation, as necessary.
CHARAC- TER	Convert to bits under control of <BIT GROUP SIZE>; if omitted use 4	No change	See Note.
FIXED	Change type to BIT	Decimal conver- sion w/ leading zeros & sign not suppressed. (7 digits + SIGN).	No change

Note: The character string may have leading blanks, sign (or none), more blanks, and decimal digits. A plus sign is ignored. The decimal digits (only the low-order 7) are converted to a binary number that is right-justified in a 24-bit field. If the sign was minus, then the 2's complement of the 24-bit field is returned.

EXAMPLES:

```
CONVERT ("-72581",FIXED)                 returns      -72581
CONVERT (@(3)752@,CHARACTER,4)                         "1E4"
CONVERT (@(1)11011@,FIXED)                              27
CONVERT ("132",BIT,2)                                   @(2)132@
CONVERT ("132",BIT,4)                                   @(4)132@
CONVERT ("2",BIT)                                       @(4)2@
```

DATA_ADDRESS

<DATA ADDRESS DESIGNATOR> ::= DATA_ADDRESS (<ADDRESS GENERATOR>)

<ADDRESS GENERATOR> ::= See ADDRESS GENERATORS

The <DATA ADDRESS DESIGNATOR> returns a value of type BIT(24) which is the base relative address generated by the <ADDRESS GENERATOR>.

DATA_LENGTH

<DATA_LENGTH DESIGNATOR>::= DATA_LENGTH (<EXPRESSION>)

Returns the length in bits of <EXPRESSION>, regardless of the data type.

DATA_TYPE

<DATA_TYPE DESIGNATOR>::= DATA_TYPE (<EXPRESSION>)

Returns the type bits of <EXPRESSION>.

DATE

<DATE FUNCTION DESIGNATOR> ::= DATE
| DATE (<DATE FORMAT>, <REPRESENTATION>)

<DATE FORMAT> ::= JULIAN | MONTH | DAY | YEAR

<REPRESENTATION> ::= BIT | DIGIT | CHARACTER

The <DATE FUNCTION DESIGNATOR> returns a bit or character string which is the date of the execution of the function.

DATE and DATE (MONTH,CHARACTER) are equivalent.

The formats (in bits) of the returned strings are:

	BIT	DIGIT	CHARACTER
JULIAN (YYDDD)	7+9=16	8+12=20	16+24=40
MONTH (MMDDYY)	4+5+7=16	8+8+8=24	16+16+16=48
DAY (DDMMYY)	5+4+7=16	8+8+8=24	16+16+16=48
YEAR (YYMMDD)	7+4+5=16	8+8+8=24	16+16+16=48

Example: DECLARE D CHARACTER(5);
D := DATE (JULIAN,CHARACTER);

DECIMAL CONVERSION

<DECIMAL CONVERSION DESIGNATOR> ::= DECIMAL (<EXPRESSION>, <DECIMAL STRING SIZE>)

<DECIMAL STRING SIZE> ::= <EXPRESSION>

The value of the first <EXPRESSION> following the reserved word DECIMAL is converted to a string of decimal characters. If the value of the <EXPRESSION> generates more than 24 bits, then only the low-order 24 bits are used.

The number of characters returned is given by the value of the <DECIMAL STRING SIZE>. A maximum of 8 decimal characters will be returned, even if the value of the <DECIMAL STRING SIZE> is greater. If the <DECIMAL STRING SIZE> is less than the number of decimal characters, then characters are truncated from the left.

DELIMITED_TOKEN

<DELIMITED TOKEN DESIGNATOR>::= DELIMITED_TOKEN (<FIRST CHARACTER>, <DELIMITERS>, <RESULT>)

<FIRST CHARACTER>::= <IDENTIFIER>

<DELIMITERS>::= <CHARACTER STRING>
| <BIT STRING>

<RESULT>::= <IDENTIFIER>

The <FIRST CHARACTER> is a simple identifier which describes the first character to be examined. <DELIMITERS> will generate 16 bits of information, each of the 8-bit bytes being used as a delimiter. For SDL, <DELIMITERS> will be %; for COBOL, @7F03@ (Quote followed by ETX).

DELIMITED_TOKEN will leave on the top of the Evaluation Stack the descriptor of the string of characters from (and including) <FIRST CHARACTER> up to (but not including) whichever delimiter was found. The descriptor of <RESULT> will be replaced by this descriptor. The address field of <FIRST CHARACTER> will be changed to point to the delimiter which stopped the scan.

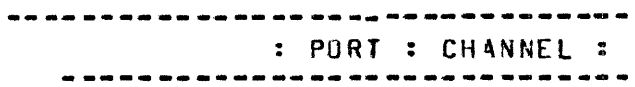
DISPATCH

<DISPATCH DESIGNATOR> ::= DISPATCH(<PORT,CHANNEL>, <I/O DESCRIPTOR ADDRESS>)

<PORT,CHANNEL> ::= <EXPRESSION>

<I/O DESCRIPTOR ADDRESS> ::= <EXPRESSION>

The rightmost seven bits of the value of <PORT, CHANNEL> contain the following information from left to right:



The rightmost 24 bits of the value of the <I/O DESCRIPTOR ADDRESS> is the absolute address of the I/O descriptor.

Using these two values, an I/O operation is initiated. A bit value with the following meanings is returned:

0 = DISPATCH REGISTER LOCK BIT SET
1 = SUCCESSFUL DISPATCH
2 = SUCCESSFUL DISPATCH, BUT MISSING DEVICE

DISPLAY_BASE

<DISPLAY BASE DESIGNATOR> ::= DISPLAY_BASE

This function leaves on the top of the Evaluation Stack a 24-bit, self-relative value of type bit which is the base-relative address of the base of the Display Stack.

DYNAMIC_MEMORY_BASE

<DYNAMIC MEMORY BASE DESIGNATOR> ::= DYNAMIC_MEMORY_BASE

The <DYNAMIC MEMORY BASE DESIGNATOR> returns a 24-bit value which is the base relative address of the program's dynamic memory. Refer to the SDL S-Language documentation for discussion of the use of dynamic memory.

EVALUATION_STACK_TOP

<EVALUATION STACK TOP DESIGNATOR> ::= EVALUATION_STACK_TOP

This function leaves on the top of the Evaluation Stack a 24-bit, self-relative value of type bit which is the base-relative address of the top of the Evaluation Stack (before execution of this function).

EXECUTE

<EXECUTE OPERATOR FUNCTION> ::= EXECUTE (<EXPRESSION LIST>)

<EXPRESSION LIST> ::= <EXPRESSION>
| <EXPRESSION LIST>, <EXPRESSION>

Note: The EXECUTE function is intended only for use by interpreter writers in the experimental design of new opcodes.

The value of the last expression may be expected to be an opcode which may then be executed by the interpreter. EXECUTE may be used as a statement as well as a <VALUE GENERATING FUNCTION DESIGNATOR>.

This statement or <VALUE GENERATING FUNCTION DESIGNATOR> when used with released interpreters will result in a "BRANCH TO INVALID OP CODE" condition.

EXTENDED ARITHMETIC FUNCTIONS

<EXTENDED ARITHMETIC FUNCTION>::= <EXTENDED ARITHMETIC FUNCTION DESIGNATOR> (<EXPRESSION>, <EXPRESSION>)

<EXTENDED ARITHMETIC FUNCTION DESIGNATOR>: X_ADD | X_SUB | X_MUL | X_DIV | | X_MOD

The indicated operation is performed on the two operands, which are treated as bit strings. The operation is performed on the full length of the operands, not just the low-order 24 bits. The length of the result is derived as described below:

Addition (Subtraction): If the two operands are of different lengths, then the shorter is padded on the left with binary zeroes. The length of the sum (difference) will be equal to the length of the longer of the two operands. The result will be in two's complement notation.

Multiplication: The length of the product will be the sum of the lengths of the two operands. (This sum may not exceed 65,535 bits.)

Division (Modulo): The length of the quotient (residue) will be length of the dividend (modulus).

For X_SUB, X_DIV, and X_MOD, the second argument represents the subtrahend, divisor, and modulus, respectively.

HASH_CODE

<HASH CODE DESIGNATOR>::= HASH_CODE (<TOKEN>)

<TOKEN>::= <EXPRESSION>

The HASH.CODE will leave on the Evaluation Stack a descriptor of type BIT and length 24. The value will be computed from the characters of <TOKEN> and the length of <TOKEN>. (If <TOKEN> is longer than 15 characters, only the first 15 are considered.) To be effective, the value generated by HASH.CODE must be used modulo a prime number (which is then the hash table size).

INTERROGATE_INTERRUPT_STATUS

<INTERROGATE INTERRUPT STATUS DESIGNATOR> ::= INTERROGATE_INTERRUPT_STATUS

A 24-bit data item of type bit is returned. The value represents the interrupt bits of the M-machine. The applicable M-machine interrupt bits are reset. Note that the INCN bits will not be reset.

LAST_LIO_STATUS

<LAST LIO STATUS DESIGNATOR>::= LAST_LIO_STATUS

Returns the last logical I/O status as type bit with a length of RS_LAST_LIO_STATUS_SIZE.

LENGTH

<LENGTH DESIGNATOR> ::= LENGTH (<EXPRESSION>)

The <LENGTH DESIGNATOR> returns a 24-bit, type bit field containing the number of units in the <EXPRESSION>. If the <EXPRESSION> is type character, then each character is a unit. Otherwise, each bit is a unit.

LIMIT_REGISTER

<LIMIT REGISTER DESIGNATOR> ::= LIMIT_REGISTER

The <LIMIT REGISTER DESIGNATOR> returns a value of type BIT(24) which is the base relative address of the program's Run Structure Nucleus. For further explanation, please refer to the B1700 MCP Manual.

LOCATION

```
<LOCATION DESIGNATOR> ::=   LOCATION (<PROCEDURE IDENTIFIER>)
                          | LOCATION (<SIMPLE IDENTIFIER>)
                          | LOCATION (<ARRAY IDENTIFIER>)

<PROCEDURE IDENTIFIER> ::=  <IDENTIFIER>

<SIMPLE IDENTIFIER>::=      <IDENTIFIER>

<ARRAY IDENTIFIER>::=       <IDENTIFIER>
```

For procedures, the <LOCATION DESIGNATOR> returns a 33-bit value (typed BIT) containing, from left to right:

ADDRESS TYPE, CONTAINING @(3)6@	4 BITS
SEGMENT NUMBER	6 BITS
PAGE NUMBER	6 BITS
DISPLACEMENT	20 BITS

This 33-bit value is the address of the procedure in question.

A forward declaration is required only during recompilation or Create-Master for any procedure on which a location is performed. An error is given if this is not done .

For simple and array identifiers, the <LOCATION DESIGNATOR> returns a 16-bit value (typed BIT) containing, from left to right:

ADDRESS TYPE CONTAINING @(2)0@	2 BITS
LEXIC LEVEL	4 BITS
OCCURRENCE NUMBER	10 BITS

NAME_OF_DAY

```
<NAME OF DAY FUNCTION
DESIGNATOR> ::=             NAME_OF_DAY
```

A character string, which is the name of the day of the week, is returned as a 9-character string. The name is left justified.

```
Example: DECLARE DAY CHARACTER(9);
         DAY:NAME_OF_DAY
```

NAME_STACK_TOP

<NAME.STACK TOP DESIGNATOR>::= NAME_STACK_TOP

This function leaves on the top of the Evaluation Stack a 24-bit, self-relative value of type bit which is the base-relative address of the top of the Name Stack.

NEXT_TOKEN

<NEXT TOKEN DESIGNATOR>::= NEXT_TOKEN (<FIRST CHARACTER>, <SEPARATOR>, <NUMERIC-TO-ALPHA INDICATOR>, <RESULT>)

<FIRST CHARACTER>::= <IDENTIFIER>

<SEPARATOR>::= <CHARACTER STRING>

<NUMERIC-TO-ALPHA INDICATOR>::= SET | RESET

The <FIRST CHARACTER> is a simple identifier which describes the first character to be examined. This will usually be the first character of the token. The <SEPARATOR> is the token separator: "_" for SDL, "-" for COBOL, etc. It must be a single character; if none is needed, use "A". <NUMERIC-TO-ALPHA INDICATOR> is set if symbols such as 2354B are allowed. It is RESET otherwise.

NEXT_TOKEN will leave on the top of the Evaluation Stack the descriptor of the next token. This token will be an identifier, a number, or a special character. The descriptor of <RESULT> will also be replaced by this descriptor. The address field of <FIRST CHARACTER> will be changed to point to the character following this token. NEXT_TOKEN assumes that <FIRST CHARACTER> describes a non-blank character.

PARITY_ADDRESS

<PARITY ADDRESS DESIGNATOR> ::= PARITY_ADDRESS

For MCP use only.

The <PARITY_ADDRESS DESIGNATOR> returns a 24-bit value which is the address of the first parity error encountered in S-Memory. If no parity error is found, @FFFFFF@ is returned.

PROCESSOR_TIME

<PROCESSOR_TIME FUNCTION GENERATOR> ::= PROCESSOR_TIME

PROCESSOR_TIME will yield the accumulated processor time since BOJ in tenths of a second as a BIT(20) data item.

Example:

```
DECLARE (PROC_TIME,HOURS,MINUTES,SECONDS,TENTHS) BIT(20);
                /*    E A R L Y      C O D E    /*
PROC_TIME := PROCESSOR_TIME;
          /*    C O D E    T O    B E    T I M E D     */
PROC_TIME :=  PROCESSOR_TIME - PROC_TIME;
HOURS     :=  PROC_TIME / 36000;
MINUTES   :=  PROC_TIME MOD 36000 / 600;
SECONDS   :=  PROC_TIME MOD 600  / 10;
TENTHS    :=  PROC_TIME MOD 10;
              /*   L A T E    C O D E    */
```

PROGRAM SWITCHES

<PROGRAM_SWITCHES DESIGNATOR> ::= PROGRAM_SWITCHES
| PROGRAM_SWITCHES (<EXPRESSION>)

This function is used to read the program switches which have been specified by the Program's Parameter Block (PPB), a control card or a SPO input. If a parameter is specified, the corresponding switch (0 through 9) is returned as a 4-bit quantity. A parameter which is less than zero or greater than nine will yield a run time error of invalid substring. If no parameter is specified, all ten switches are returned as a 40-bit result. SDL provides no means to modify the program switches.

SEARCH_LINKED_LIST

```
<SEARCH_LINKED_LIST
DESIGNATOR> ::=          SEARCH_LINKED_LIST
                         (<START RECORD>,<COMPARE FIELD>,
                         <COMPARE VALUE>,<RELATION>,
                         <LINK FIELD>)

<START RECORD> ::=       <EXPRESSION>

<COMPARE FIELD> ::=      <TEMPLATE>

<COMPARE VALUE> ::=      <EXPRESSION>

<RELATION> ::=           < | <= | = | /= | >= | > |
                         LSS | LEQ | EQL | NEQ |
                         GEQ | GTR |

<LINK FIELD> ::=         <TEMPLATE>

<TEMPLATE> ::=           <ADDRESS GENERATOR>
```

1. The <START RECORD> is the first structure to be examined. Typically, it is an <ADDRESS GENERATOR>, but an <EXPRESSION> is allowed.

2. The <COMPARE FIELD> is a template which gives the relative offset and size in the structure, of the 24 (or less) bit field being compared with the <COMPARE VARIABLE>.

3. The <COMPARE VALUE> is the value against which the specified field in the structure is compared. <COMPARE VALUE> is considered "on the left" of the relation.

4. The <RELATION> specifies the desired relation in the comparison of the two values.

5. The <LINK FIELD> is a template which gives the relative offset and size in the structure, of the 24 (or less) bit field containing the address of the

next structure to be examined (if comparison with the current structure fails).

A template is an address generator whose address is relative to the beginning of a structure rather than base relative. A field in a structure declared REMAPS BASE has such an address.

The last structure in the linked list contains all 1 bits in the field described by the <LINK FIELD>.

The linked list is searched until the desired comparison succeeds or until the comparison fails with the last structure.

If the search succeeds, the base-relative address of the current structure is returned as a 24-bit value. If the search fails, @FFFFFF@ is returned.

SEARCH_SDL_STACKS

<SEARCH_SDL_STACKS DESIGNATOR>::=	SEARCH_SDL_STACKS (<STACK BASE>, <STACK TOP>, <COMPARE BASE>, <COMPARE TOP>)
<STACK BASE>::=	<EXPRESSION>
<STACK TOP>::=	<EXPRESSION>
<COMPARE BASE>::=	<EXPRESSION>
<COMPARE TOP>::=	<EXPRESSION>

The four parameters are expected to generate values which are base-relative addresses of the base and top of a stack of SDL descriptors and of an address range, respectively. The stack will be searched for a non-array, non-self-relative SDL descriptor whose address is within the given range. If the search is successful @(1)1@ will be returned; otherwise, @(1)0@ will be returned.

SEARCH_SERIAL_LIST

```
<SEARCH SERIAL
LIST DESIGNATOR> ::=        SEARCH_SERIAL_LIST (<SSL COMPARE VALUE>,
                            <SSL COMPARE TYPE>, <SSL COMPARE FIELD>,
                            <SSL FIRST ITEM>, <SSL TABLE LENGTH>,
                            <SSL RESULT VARIABLE>)

<SSL COMPARE VALUE> ::=     <EXPRESSION>

<SSL COMPARE TYPE> ::=      < | <= | = | /= | >= | >
                          | LSS | LEQ | EQL | NEQ | GEQ | GTR

<SSL COMPARE FIELD>:::=     <TEMPLATE>

<SSL FIRST ITEM>:::=        <ADDRESS GENERATOR>

<SSL TABLE LENGTH>:::=      <EXPRESSION>

<SSL RESULT VARIABLE>:::=   <ADDRESS GENERATOR>

<TEMPLATE> ::=              <ADDRESS GENERATOR>
```

SEARCH_SERIAL_LIST searches a serial list of items beginning with the structure described by <SSL FIRST ITEM>. <SSL COMPARE VALUE> is compared (as specified by <SSL COMPARE TYPE>) against the field of the field described by <SSL COMPARE FIELD> (<SSL COMPARE FIELD> is a TEMPLATE) until a match has been found, or until <SSL TABLE LENGTH> number of bits has been searched.

When the relation is non-commutative, the comparisons are made as though <SSL COMPARE VALUE> was "on the left" of the relation.

If the search succeeds, the base relative address of the item containing the successful <SSL COMPARE FIELD> is stored in <SSL RESULT VARIABLE> and a 3(1)1@ is returned.

If the search fails, then the end address of the table if stored in <SSL RESULT VARIABLE> and a 3(1)0@ is returned.

S_MEM_SIZE, M_MEM_SIZE

<MEMORY SIZE DESIGNATOR> ::= S_MEM_SIZE | M_MEM_SIZE

The requested memory size is returned as a 24-bit data item of type bit.

SORT_SEARCH

<SORT_SEARCH DESIGNATOR> ::= SORT_SEARCH (<TABLE ADDRESS>,<LIMIT>)

<TABLE ADDRESS> ::= <ADDRESS GENERATOR>

<LIMIT> ::= <EXPRESSION>

For use by sort only.

The <SORT SEARCH DESIGNATOR> provides the information to evaluate a record for sorting purposes. The <TABLE ADDRESS> contains the address, in an array of records, of the first record to be examined and the condition under which records will be selected.

The <LIMIT> specifies the last record to be examined.

SORT_STEP_DOWN

<SORT_STEP_DOWN DESIGNATOR> ::= SORT_STEP_DOWN (<RECORD 1>,<RECORD 2>, <KEY TABLE ADDRESS>

<RECORD 1> ::= <EXPRESSION>

<RECORD 2> ::= <EXPRESSION>

<KEY TABLE ADDRESS> ::= <EXPRESSION>

For use by sort only.

The <SORT_STEP_DOWN DESIGNATOR> provides the information necessary to compare two records. <RECORD 1> and <RECORD 2> are, respectively, the first and second records which are to be compared. The <KEY TABLE ADDRESS> specifies the sort key used in the comparison.

SORT_UNBLOCK

<SORT_UNBLOCK DESIGNATOR> ::= SORT_UNBLOCK (<MINI FIB ADDRESS>, <LENGTH>,<SOURCE>,<DESTINATION>)

<MINI FIB ADDRESS> ::= <ADDRESS GENERATOR>

<LENGTH> ::= <EXPRESSION>

<SOURCE> ::= <EXPRESSION>

<DESTINATION> ::= <EXPRESSION>

For use by SORT only.

The <SORT_UNBLOCK DESIGNATOR> moves a record to or from a buffer, updating the buffer pointer and block count. It normally returns a zero. When the block count goes to zero, it restores the original buffer pointer and block count, and returns a 1, signalling the need for an I/O.

A bit on the mini-FIB signals SORT_UNBLOCK to create sort tags. For this function, it uses the sort key table and selects only the key information to move from the buffer. A value in the mini-FIB represents the length of the receiving field.

SPO_INPUT_PRESENT

<SPO INPUT PRESENT DESIGNATOR>::= SPO_INPUT_PRESENT

A special, SPO_INPUT_PRESENT, has been added to allow the presence of SPO input to be tested before having to perform an accept to the MCP.

SUBBIT AND SUBSTR

```
<SUB-STRING VALUE
DESIGNATOR> ::=          <SUB-STRING FUNCTION IDENTIFIER>
                         (<STRING VALUE>,<OFFSET PART>)
                       | <SUB-STRING FUNCTION IDENTIFIER>
                         (<STRING VALUE>,<OFFSET PART>,
                         <LENGTH PART>)

<SUB-STRING FUNCTION
IDENTIFIER> ::=          SUBBIT | SUBSTR

<STRING VALUE> ::=       <EXPRESSION>

<OFFSET PART> ::=        <EXPRESSION>

<LENGTH PART> ::=        <EXPRESSION>
```

The <SUB-STRING VALUE DESIGNATOR> and the <SUB-STRING ADDRESS DESIGNATOR> are identical except that the former returns a value if its <STRING VALUE> is not an <ADDRESS GENERATOR>. Please see SUBBIT AND SUBSTR under ADDRESS VARIABLES for the specifics of the function.

The following examples illustrate some of the uses of the <SUB-STRING VALUE DESIGNATOR>:

```
X:SUBSTR(A CAT B,5,10);
MAKE_DESCRIPTOR(@48@ CAT SUBBIT(A OR B, 0, 16) CAT X) [...;
IF SUBSTR(@06@ CAT ABC, 0) = Y THEN ...;
```

SWAP

<SWAP DESIGNATOR> ::= SWAP (<ADDRESS GENERATOR>,<EXPRESSION>)

The length of the value described by the <ADDRESS GENERATOR> is used as the length, L, of the data to be SWAPped. However, if the length of the value is greater than 24 bits, L will be 24 bits, and only the low-order 24 bits of the <ADDRESS GENERATOR> will be modified.

SWAP is indeed a true swap operation: that is, the items are exchanged in one "virtual" memory cycle. This is necessary for the synchronization of independent processes (e.g., MCP and GISMO).

The rightmost L bits of the value described by the <ADDRESS GENERATOR> are isolated, and become the destination field.

The rightmost L bits of the value generated by the <EXPRESSION> are isolated. Leading zeroes are supplied if the length of the value is less than L bits long. This field is known as the source field.

The source field is stored into the destination field, the original value of which is the value returned. The returned value is of type bit and of length L.

Example:

```
A10;
IF SWAP (A,1) THEN DO ... END;
              ELSE DO ... END;
```

In the above example, the ELSE part of the statement is evaluated, since A was originally set to 0 (i.e., false). At the end of the SWAP, 1 has been stored into A, and 0 returned to the top of the Evaluation Stack.

TIME

<TIME FUNCTION DESIGNATOR> ::= TIME
| TIME (<TIME FORMAT>,<REPRESENTATION>)

<TIME FORMAT> ::= COUNTER | MILITARY | CIVILIAN

<REPRESENTATION> ::= BIT | DIGIT | CHARACTER

The <TIME FUNCTION DESIGNATOR> returns a bit or character string which is the time of the function's execution. The <TIME FORMAT> may have three basic formats:

COUNTER: Returns the time of day in tenths of seconds.

MILITARY: Returns the time of day in the following form -- HHMMSST (Where T=Tenths of seconds).

CIVILIAN: Returns HHMMSSTAP(Where AP=AM OR PM).

The time of day may be represented in either bits, digits, or characters in the following formats:

	BIT	DIGIT	CHARACTER
COUNTER	20 BITS	24 BITS	48 BITS
MILITARY	5+6+6+4=21	8+8+8+4=28	16+16+16+8=56
CIVILIAN	4+6+6+4+16=36	8+8+8+4+16=44	16+16+16+8+16=72

NOTE: TIME and TIME (CIVILIAN,CHARACTER) are equivalent.

TIMER

<TIMER DESIGNATOR>::= TIMER

A value of type BIT(24) is returned. The value is the current setting of the TIME register.

VALUE_DESCRIPTOR

<DESCRIPTOR-VALUE GENERATOR DESIGNATOR> ::= VALUE_DESCRIPTOR (<ADDRESS GENERATOR>)

<ADDRESS GENERATOR> ::= See ADDRESS GENERATORS

The <ADDRESS GENERATOR> is represented by a descriptor at the top of the Evaluation Stack. This descriptor is moved to the Value Stack. In its place on the Evaluation Stack is left a descriptor describing the one just moved to the Value Stack.

The Name-Value bit is set in the descriptor left in the Evaluation Stack.

WAIT

<WAIT FUNCTION> ::= WAIT <START POSITION> (<EVENT LIST>)

<START POSITION> ::= [<EXPRESSION>] | <EMPTY>

<EVENT LIST> ::= <EVENT> | <EVENT LIST>, <EVENT>

<EVENT> ::= <SIMPLE EVENT> | <QUALIFIED EVENT>

<QUALIFIED EVENT> ::= <SIMPLE EVENT> WHEN <EXPRESSION>

<SIMPLE EVENT> ::= TIME_TENTHS (<EXPRESSION>)
| SPO_INPUT_PRESENT
| SPO_INPUT_PRESENT
| DC_IO_COMPLETE
| READ_OK (<FILE SPECIFIER>)
| WRITE_OK (<FILE SPECIFIER>)
| Q_WRITE_OCCURRED (<FILE IDENTIFIER>)

<FILE SPECIFIER> ::= <FILE IDENTIFIER>
| <FILE IDENTIFIER> [<EXPRESSION>]

The WAIT function returns a fixed value which is the ordinal position of a true event in the <EVENT LIST>. If no event is true, the process will be blocked until one of the events occurs. If more than one is true, the value that is returned is the position of the first event found true in a left to right circular scan starting from <START POSITION>. If <START POSITION> is empty, zero is assumed. If <START POSITION> is

greater than or equal to the number of items in the <EVENT LIST>, the MCP will terminate the job. In the case of a <QUALIFIED EVENT>, the event will never become true unless the qualifying <EXPRESSION> evaluates to true, i.e., its lowest order bit is a one.

The various events are true when the condition(s) below are satisfied:

EVENT	CONDITION(S)
TIME_TENTHS (<EXPRESSION>)	The specified number of tenths of seconds have elapsed since the WAIT began execution.
SPO_INPUT_PRESENT	A message from the operator has been queued for the WAITing program.
DC_IO_COMPLETE	A previously initiated data communications IO has been completed.
READ_OK (<FILE SPECIFIER>)	The buffer for the specified file contains a record waiting to be read. If a [<EXPRESSION>] is specified, it is taken to be a subscript of a queue file family. If the file is a queue file family and no subscript is specified, the event is always true.
WRITE_OK (<FILE SPECIFIER>)	A buffer for the specified file is empty, waiting for a write operation. See above for queue file families.
G_WRITE_OCCURRED (<FILE IDENTIFIER>)	A write operation has been done (by another process) on a member of a queue file family named in the time since the WAIT began execution. This event will be correct only when preceded by MESSAGE COUNT.

Restrictions:

1. If TIME.TENTHS is in the list, it must be at the extreme left.
2. The maximum number of tenths of seconds is 864,000, i.e., 24 hours.

I/O CONTROL STATEMENTS

```
<I/O CONTROL STATEMENT> ::=  <OPEN STATEMENT>
                           | <CLOSE STATEMENT>
                           | <READ STATEMENT>
                           | <WRITE STATEMENT>
                           | <SEEK STATEMENT>;
                           | <ACCEPT STATEMENT>;
                           | <DISPLAY STATEMENT>;
                           | <SPACE STATEMENT>
                           | <SKIP STATEMENT>;
```

Each file is numbered sequentially, beginning with zero. This number is the <FILE NUMBER> and will eventually be used as an index into the FIB dictionary. The file declaration will be used to construct an FPB in the code file.

OPEN STATEMENT

```
<OPEN STATEMENT>::=          <OPEN PART>;
                           | <OPEN PART>; <FILE MISSING PART>
                           | <OPEN PART>; <FILE LOCKED PART>
                           | <OPEN PART>; <FILE MISSING PART>
                             <FILE LOCKED PART>

<OPEN PART>::=               OPEN <FILE DESIGNATOR>
                             <OPEN ATTRIBUTE PART>

<FILE DESIGNATOR>::=         <FILE IDENTIFIER>
                           | <SWITCH FILE IDENTIFIER> (<EXPRESSION>)

<OPEN ATTRIBUTE PART> ::=    <EMPTY>
                           | <OPEN ATTRIBUTE LIST>
                           | WITH <OPEN ATTRIBUTE LIST>

<OPEN ATTRIBUTE LIST> ::=    <OPEN ATTRIBUTE>
                           | <OPEN ATTRIBUTE> <ATTRIBUTE SEPARATOR>
                             <OPEN ATTRIBUTE LIST>

<ATTRIBUTE SEPARATOR>::=     , | <SLASH> | <EMPTY>

<OPEN ATTRIBUTE> ::=         <INPUT-OUTPUT MODE>
                           | <LOCK MODE>
                           | <OPEN ACTION MODE>
                           | <MFCU MODE>
                           | <ON BEHALF OF MODE>

<INPUT-OUTPUT MODE> ::=      INPUT | OUTPUT | NEW

<LOCK MODE> ::=              LOCK | LOCK.OUT

<OPEN ACTION MODE> ::=       NO_REWIND | REVERSE

<MFCU MODE>::=               PUNCH | PRINT |
                             INTERPRET | STACKERS

<ON BEHALF OF MODE>::=       ON_BEHALF_OF <EXPRESSION>

<FILE MISSING PART>::=       ON FILE_MISSING <EXECUTABLE STATEMENT>

<FILE LOCKED PART>::=        ON FILE_LOCKED <EXECUTABLE STATEMENT>
```

FORMAT OPTIONS:

1. OPEN DECLARED_FILE;

   If no attributes are specified, INPUT is assumed.

	FOLLOWED BY:	AND/OR:
		LOCK
	INPUT	LOCK_OUT
2. OPEN DECLARED_FILE	OUTPUT	NO_REWIND
	NEW *	REVERSE
	INPUT, OUTPUT	LOCK, NO_REWIND
3. OPEN DECLARED_FILE WITH	OUTPUT, NEW	LOCK, REVERSE
	INPUT, OUTPUT, NEW	LOCK_OUT, NO_REWIND
		LOCK_OUT, REVERSE

* NEW alone assumes OUTPUT, NEW.

Note: The combination INPUT, NEW results in a syntax error.

If the <OPEN ATTRIBUTE>s have been explicitly or implicitly included in the file declaration, then the file need not be explicitly opened here.

CLOSE STATEMENT

```
<CLOSE STATEMENT>::=          CLOSE <FILE DESIGNATOR>
                              <CLOSE ATTRIBUTE PART>;

<FILE DESIGNATOR> ::=         <FILE IDENTIFIER>
                            | <SWITCH FILE IDENTIFIER> (<EXPRESSION>)

<CLOSE ATTRIBUTE PART> ::=    <EMPTY>
                            | <CLOSE ATTRIBUTE LIST>
                            | WITH <CLOSE ATTRIBUTE LIST>

<CLOSE ATTRIBUTE LIST> ::=    <CLOSE ATTRIBUTE>
                            | <CLOSE ATTRIBUTE> <ATTRIBUTE SEPARATOR>
                              <CLOSE ATTRIBUTE LIST>

<ATTRIBUTE SEPARATOR> ::=     , | <SLASH> | <EMPTY>

<CLOSE ATTRIBUTE> ::=         <CLOSE MODE>
                            | CRUNCH | ROLLOUT | PURGE | REMOVE

<CLOSE MODE> ::=              REEL | RELEASE | PURGE | REMOVE
                            | NO_REWIND | LOCK
```

FORMAT OPTIONS:

1. CLOSE DECLARED_FILE;

   There is no default. If LOCK is specified as part of the file attributes, the file is LOCKed if the program terminates abnormally. Otherwise, the file is not LOCKed.

	FOLLOWED BY	AND/OR ONE OF: *
	0 OR ONE OF:	
2. CLOSE DECLARED_FILE	ROLLOUT CRUNCH IF_NOT_CLOSED	REEL RELEASE PURGE REMOVE NO_REWIND LOCK

* If more than one option is specified, only the final one is used by the compiler.

Files need not be explicitly closed. However, closing a file when finished with it will free memory space for other uses.

READ STATEMENT

```
<READ STATEMENT> ::=          <READ PART>;
                            | <READ PART>;<ON SEQUENCE>
                            | <READ PART><RESULT MASK>; <ON SEQUENCE>

<READ PART> ::=               <READ SPECIFIER>
                            | <DISK READ SPECIFIER>
                            | <REMOTE READ SPECIFIER>
                            | <QUEUE READ SPECIFIER>

<READ SPECIFIER> ::=          READ <FILE DESIGNATOR>
                              (<ADDRESS GENERATOR>)

<FILE DESIGNATOR> ::=         <FILE IDENTIFIER>
                            | <SWITCH FILE IDENTIFIER> (<EXPRESSION>)

<DISK READ SPECIFIER> ::=     READ
                              <FILE DESIGNATOR>
                              <RECORD ADDRESS PART>
                              (<ADDRESS GENERATOR>)

<RECORD ADDRESS PART> ::=     <EMPTY>
                            | [<RECORD ADDRESS>]

<RECORD ADDRESS> ::=          <EXPRESSION>

<REMOTE READ SPECIFIER> ::=   READ <FILE DESIGNATOR>
                              <REMOTE KEY PART>
                              (<ADDRESS GENERATOR>)

<REMOTE KEY PART> ::=         <EMPTY>
                            | [<REMOTE KEY>]

<REMOTE KEY> ::=              <ADDRESS GENERATOR>

<QUEUE READ SPECIFIER> ::=    READ <FILE DESIGNATOR>
                              <QUEUE FAMILY MEMBER PART>
                              (<ADDRESS GENERATOR>)

<QUEUE FAMILY
MEMBER PART> ::=              <EMPTY>
                            | [<QUEUE FAMILY MEMBER>]

<QUEUE FAMILY MEMBER> ::=     <EXPRESSION>

<RESULT MASK>::=              WITH RESULT_MASK <ADDRESS GENERATOR>
```

The <READ STATEMENT> provides the necessary information to read a file: A file identifier, record address, data information, and instructions to be executed if an end-of-file or a parity error is detected.

The <READ STATEMENT> separates files into four categories: disk files, remote files, queue files, and all others (card, tape, papertape, etc.). If the file attributes indicate a random disk file, the user may specify <RECORD ADDRESS>. In all cases, the user need only give the <FILE DESIGNATOR> and <ADDRESS GENERATOR>.

If the file is of type REMOTE, and the REMOTE_KEY ATTRIBUTE is set then a <REMOTE KEY> may be used. (For the format of this, see the discussion under REMOTE_KEY in the FILE DECLARATION SECTION.) If the REMOTE_KEY attribute is not set, then a <REMOTE KEY> may not be used. After performing the read, the REMOTE KEY will have been stored in the field specified as the <REMOTE KEY>.

If the file is of type QUEUE and is a multi-queue family, then a <QUEUE FAMILY MEMBER> may be used. This is an expression whose value will specify which member of the family to read from. If this is omitted, then the oldest message in all of the queues will be read.

If the <RESULT MASK> option is used, the occurrence of an exception in the mask is signalled by the ON EXCEPTION sequence.

WRITE STATEMENT

```
<WRITE STATEMENT> ::=          <WRITE PART>;
                             | <WRITE PART>;<ON SEQUENCE>
                             | <WRITE PART> <RESULT MASK>;
                               <ON SEQUENCE>

<WRITE PART> ::=               <WRITE SPECIFIER>
                             | <DISK WRITE SPECIFIER>
                             | <REMOTE WRITE SPECIFIER>
                             | <QUEUE WRITE SPECIFIER>

<WRITE SPECIFIER> ::=          WRITE <FILE DESIGNATOR>
                               <CARRIAGE CONTROL PART>
                               (<EXPRESSION>)
                             | WRITE <FILE IDENTIFIER>
                               <CARRIAGE CONTROL PART>

<FILE DESIGNATOR> ::=          <FILE IDENTIFIER>
                             | <SWITCH FILE IDENTIFIER> (<EXPRESSION>)

<CARRIAGE CONTROL PART> ::=    <EMPTY>
                             | <CARRIAGE CONTROL SPECIFIER>

<CARRIAGE CONTROL
SPECIFIER> ::=                 NO | SINGLE | DOUBLE | PAGE
                             | <SKIP-TO-CHANNEL> | NEXT

<SKIP-TO-CHANNEL> ::=          <CHANNEL NUMBER>

<CHANNEL NUMBER> ::=           1 | 2 | 3 | ... | 11 | 12

<DISK WRITE SPECIFIER> ::=     WRITE
                               <FILE DESIGNATOR>
                               <RECORD ADDRESS PART>
                               (<EXPRESSION>)

<RECORD ADDRESS PART> ::=      <EMPTY>
                             | [<RECORD ADDRESS>]

<RECORD ADDRESS> ::=           <EXPRESSION>

<REMOTE WRITE
SPECIFIER>:::=                 WRITE <FILE DESIGNATOR>
                               <REMOTE KEY PART>
                               (<EXPRESSION>)

<REMOTE KEY PART>:::=          <EMPTY>
                             | [<REMOTE KEY>]

<REMOTE KEY>:::=               <ADDRESS GENERATOR>

<QUEUE WRITE
```

```
SPECIFIER>::=                    WRITE <FILE DESIGNATOR>
                                 <QUEUE FAMILY MEMBER PART> <TOP>
                                 (<ADDRESS GENERATOR>)

<FILE DESIGNATOR>::=               <FILE IDENTIFIER>
                                 | <SWITCH FILE IDENTIFIER> (<EXPRESSION>)

<TOP> ::=                          <EMPTY> | TOP

<QUEUE FAMILY
MEMBER PART>::=                    <EMPTY>
                                 | [<QUEUE FAMILY MEMBER>]

<QUEUE FAMILY MEMBER>::=         <EXPRESSION>

<RESULT MASK>::=                 WITH RESULT_MASK <ADDRESS GENERATOR>
```

The <WRITE STATEMENT> provides the necessary information to write a file. The <WRITE STATEMENT> treats disk files separately from other file types by allowing the user the option of specifying <RECORD ADDRESS> on his random disk files. The <CARRIAGE CONTROL PART> is intended for use with a printer file.

If the file is of type REMOTE, and the REMOTE_KEY attribute is set then a <REMOTE KEY> may be used. (For the format of this, see the discussion under REMOTE_KEY in the FILE DECLARATION section.) If the REMOTE_KEY attribute is not set, then a <REMOTE KEY> may not be used. The <REMOTE KEY> will specify the terminal to which the write is to be performed.

If <DISK WRITE SPECIFIER> is used when the actual device is a data recorder, the <RECORD ADDRESS> will be used to select a stacker.

If the file is of type QUEUE and is a multi-queue family, then a <QUEUE FAMILY MEMBER> may be used. This is an expression whose value will specify which member of the family to write to. If TOP is specified, the message will be written to the front of the queue.

If the <END-OF-PAGE PART> is set in the file attributes, then when end-of-page is detected on a printer file, the <EOF PART> will be executed. This facilitates, for example, printing totals and/or headings without keeping a line counter.

If the <RESULT MASK> option is used, the occurrence of an exception in the mask is signalled by the ON EXCEPTION sequence.

EXAMPLE:

```
WRITE PRINTOUT SINGLE (PRINT_LINE);
     ON EOF DO;
               WRITE PRINTOUT; % SKIP A LINE:
               WRITE PRINTOUT PAGE (TOTALS);
               WRITE PRINTOUT DOUBLE (HEADER);
          END;
```

SEEK STATEMENT

```
<SEEK STATEMENT> ::=          SEEK
                              <FILE DESIGNATOR>
                              [<RECORD ADDRESS>]

<FILE DESIGNATOR>::=          FILE IDENTIFIER>
                            | <SWITCH FILE IDENTIFIER> (<EXPRESSION>)

<RECORD ADDRESS> ::=          <EXPRESSION>
```

The <SEEK STATEMENT> calls up a record from a random disk file in preparation for a read on that record. This statement should only be used with disk files that are being read using a random access technique.

A <SEEK STATEMENT> performed immediately prior to a <READ STATEMENT> is less effective than merely reading the record.

ACCEPT STATEMENT

<ACCEPT STATEMENT> ::= ACCEPT <ADDRESS GENERATOR>

The <ACCEPT STATEMENT> causes the execution of a program to halt until the appropriate information is entered via the SPO by the operator. The message keyed in will be read into the area specified by the <ADDRESS GENERATOR> following the reserved word ACCEPT.

See ADDRESS VARIABLES for the syntax of the <ADDRESS GENERATOR>.

DISPLAY STATEMENT

```
<DISPLAY STATEMENT> ::=     DISPLAY <EXPRESSION>
                            <CRUNCH SPECIFIER>

<CRUNCH SPECIFIER> ::=      <EMPTY>
                          | , CRUNCHED
```

The <DISPLAY STATEMENT> prints an output message on the SPO. As noted, the <CRUNCH SPECIFIER> is optional. If , CRUNCHED is specified, the system will delete trailing blanks and substitute one blank for each occurrence of multiple embedded blanks.

SPACE STATEMENT

```
<SPACE STATEMENT> ::=      <SPACE PART>;
                         | <SPACE PART>; <ON SEQUENCE>

<SPACE PART> ::=           SPACE <FILE DESIGNATOR>
                           <SPACING SPECIFIER>

<FILE DESIGNATOR> ::=      <FILE IDENTIFIER>
                         | <SWITCH FILE IDENTIFIER>(<EXPRESSION>)

<SPACING SPECIFIER> ::=    <EXPRESSION | TO <EXPRESSION>
                         | TO_EOF
```

The <SPACE STATEMENT> allows the user to skip over certain records in a sequential file.

The <SPACING SPECIFIER> may take three forms. An <EXPRESSION> alone will indicate the number of records to be spaced. It may be a negative number indicating reverse spacing. TO <EXPRESSION> will always be a positive number and indicates the number of the record to space to. TO_EOF will cause the file to space to its current end.

SKIP STATEMENT

<SKIP STATEMENT> ::= SKIP <FILE IDENTIFIER> TO <CHANNEL NUMBER>

<FILE DESIGNATOR> ::= <FILE IDENTIFIER>
| <SWITCH FILE IDENTIFIER> (<EXPRESSION>)

<CHANNEL NUMBER> ::= 1 | 2 | 3 | ... | 11 | 12

The <SKIP STATEMENT> causes the line printer to skip to a specified channel number on its carriage tape. The channel numbers control the vertical spacing of data on a printed page and are defined by the carriage tape on the device.

ON SEQUENCE

```
<ON SEQUENCE> ::=      <ON CLAUSE> <EXECUTABLE STATEMENT>
                     | <ON SEQUENCE> <ON CLAUSE> <EXECUT-
                       TABLE STATEMENT>

<ON CLAUSE> ::=        ON EOF | ON INCOMPLETE_IO
                     | ON EXCEPTION | ON EXCEPTION
                       (<STATUS>)

<STATUS> ::=           <ADDRESS GENERATOR>
```

An ON SEQUENCE is used to examine the status of the I/O requested by the preceding statement. When any of the <ON CLAUSE>s are true, the corresponding <EXECUTABLE STATEMENT> will be executed before proceeding. Only one condition will be true. If <STATUS> is requested in ON EXCEPTION, a 24-bit result describing the exact exception will be assigned to the given <ADDRESS GENERATOR>.

The <EXECUTABLE STATEMENT>s of the <ON SEQUENCE> are considered subordinate to the <WRITE STATEMENT>. Therefore, segmentation of these statements is temporary (See THE SEGMENT STATEMENT).

Note: Exceptions may be masked by the EXCEPTION_MASK clause in the file declaration.

EXECUTABLE STATEMENTS

```
<EXECUTABLE STATEMENT
LIST> ::=                       <EXECUTABLE STATEMENT>
                              | <EXECUTABLE STATEMENT>
                                <EXECUTABLE STATEMENT LIST>

<EXECUTABLE STATEMENT> ::=      <DO GROUP>;
                              | <GROUP TERMINATION STATEMENT>;
                              | <IF STATEMENT>;
                              | <CASE STATEMENT>;
                              | <ASSIGNMENT STATEMENT>;
                              | <REFER STATEMENT>;
                              | <REDUCE STATEMENT>;
                              | <EXECUTE-PROCEDURE STATEMENT>;
                              | <EXECUTE-FUNCTION STATEMENT>;
                              | <I/O CONTROL STATEMENT>
                              | <MODIFY INSTRUMENTS>;
                              | <NULL STATEMENT>
                              | <FILE ATTRIBUTE STATEMENT>;
                              | <STOP STATEMENT>;
                              | <ZIP STATEMENT>;
                              | <SEARCH STATEMENT>;
                              | <ACCESS FILE HEADER STATEMENT>;
                              | <ARRAY PAGE TYPE STATEMENT>;
                              | <COROUTINE STATEMENT>;
                              | <SEGMENT STATEMENT>
                                <EXECUTABLE STATEMENT>

<ASSIGNMENT STATEMENT> ::=      SEE ASSIGNMENT STATEMENTS
                                AND EXPRESSIONS

<I/O CONTROL STATEMENT> ::=     SEE I/O CONTROL STATEMENTS

<SEGMENT STATEMENT> ::=         SEE THE SEGMENT STATEMENT
```

DO GROUPS

```
<DO GROUP> ::=                  <GROUP HEAD>
                                <GROUP BODY>

<GROUP HEAD> ::=                <GROUP NAME>
                                <FOREVER PART>;

<GROUP NAME> ::=                  DO
                                | DO <GROUP IDENTIFIER>

<FOREVER PART> ::=                <EMPTY>
                                | FOREVER

<GROUP IDENTIFIER> ::=            <IDENTIFIER>

<GROUP BODY> ::=                  <EXECUTABLE STATEMENT LIST>
                                  <GROUP ENDING>

<GROUP ENDING> ::=                END
                                | END <GROUP IDENTIFIER>
```

The <DO GROUP> is a collection of <EXECUTABLE STATEMENT>s which functions as a routine. It is executed once unless FOREVER appears after the <GROUP NAME>.

If FOREVER is present, the <DO GROUP> will be executed iteratively until a specific condition is met. Only a <GROUP TERMINATION STATEMENT> (UNDO) or a <TYPED PROCEDURE RETURN STATEMENT> (RETURN) can get the program out of this loop. See the following example:

```
DO THIS FOREVER;
   READ CARD (A); ON EOF UNDO;
   IF 55 GTR BUMP X
      THEN WRITE PRINTER (A);
      ELSE DO;
           XII;
           WRITE PRINTER PAGE (A);
           END;
END THIS;
```

If it is necessary to execute the statements in a <DO GROUP> from different points in the program, more efficient code is generated by making the body of the group a procedure rather than by repeating the <DO GROUP>.

RESTRICTIONS:

1. If a <GROUP IDENTIFIER> is included in the <GROUP NAME>, it must also appear in the <GROUP ENDING>.

2. If the <GROUP NAME> does not include an identifier, the <GROUP ENDING> must not contain one.

3. FOREVER is not a reserved word and may appear as the <GROUP IDENTIFIER>. DO FOREVER; is considered to be the <GROUP HEAD> of an un-named, iterative <DO GROUP>. DO FOREVER FOREVER is a legal heading for a named, iterative group.

4. Nested <DO GROUP>s may not have duplicate identifiers. If this occurs, a warning message will appear on the program listing.

5. <DO GROUP>s may be nested 32 levels deep. However, a <GROUP TERMINATION STATEMENT> can UNDO only a maximum of 16 levels.

UNDO

<GROUP TERMINATION STATEMENT> ::= UNDO
| UNDO <GROUP IDENTIFIER>

<GROUP IDENTIFIER> ::= <IDENTIFIER>

The <GROUP TERMINATION STATEMENT> will cause the execution of a <DO GROUP> to cease, and will transfer control to the next statement following the <DO GROUP> which has teen UNDONE. The statement may take one of three forms:

1. UNDO will transfer control out of the <DO GROUP> which contains the statement.

2. UNDO <GROUP IDENTIFIER> takes control out of the <DO GROUP> specified by the identifier.

3. Another form, UNDO(*), is now considered obsolete. It transferred control out of the outermost <DO GROUP>.

Note: UNDO <IDENTIFIER> can undo a maximum of 16 levels.

EXAMPLE:

```
1.      DO ONE;
2.         DO TWO FOREVER;
3.            IF <EXPRESSION> THEN
4.               DO THREE;
5.                  CASE <EXPRESSION>;
6.                       UNDO; /* SAME AS UNDO THREE; */
7.                       UNDO TWO;
8.                  END CASE;
9.               END THREE;
10.        END TWO;
11.     END ONE;
```

Execution of line 6 transfers control to line 10.
Execution of line 7 transfers control to the statement following line 11.

IF STATEMENT

```
<IF STATEMENT> ::=      <IF CLAUSE>
                        <EXECUTABLE STATEMENT>
                      | <IF CLAUSE>
                        <EXECUTABLE STATEMENT>
                        ELSE <EXECUTABLE STATEMENT>

<IF CLAUSE> ::=         IF <EXPRESSION> THEN
```

The <EXPRESSION> is evaluated. If the low-order bit of the result is 1 (i.e., true), the statement following THEN is executed. If the low-order bit is 0 (i.e., false), the statement following ELSE (if present) is executed. If the result of the<EXPRESSION> is false, and the ELSE part is omitted, control is transferred to the next statement after the <IF STATEMENT>.

<IF STATEMENT>s may be nested. The outermost <IF CLAUSE> and the corresponding ELSE, if any, are on Nesting Level 0. The <EXECUTABLE STATEMENT>s following THEN and ELSE are on Nesting Level 1. Nesting may be no deeper than 32 levels.

When using nested <IF STATEMENT>s, the user must maintain correspondence between the delimiters THEN and ELSE on each level. The innermost ELSE will always be associated with the innermost THEN. From this point continues an outward progression (i.e., from highest nesting level to lowest) of THEN-ELSE association.

Thus, if an <IF STATEMENT> on Nesting Level N is to have an ELSE associated with it, then every <IF STATEMENT> on a nesting level greater than N must also have ELSEs associated with them. If the user wishes to execute nothing on a false condition, then ELSE followed by a <NULL STATEMENT> may be used.

EXAMPLE:

Let E-1, E-2, E-3, and E-4 be <EXPRESSION>s, and let S-2, S-3, and S-4 be <EXECUTABLE STATEMENT>s.

```
IF E-1
THEN IF E-2
     THEN IF E-3
          THEN IF E-4
               THEN S-4;
               ELSE;
          ELSE S-3;
     ELSE S-2;
```

All statements here are the IF-THEN-ELSE type, except the first IF which has no corresponding ELSE.

CASE STATEMENT

<CASE STATEMENT> ::= <CASE HEAD>
<CASE BODY>

<CASE HEAD> ::= CASE <EXPRESSION>

<CASE BODY> ::= <EXECUTABLE STATEMENT LIST>
<CASE ENDING>

<CASE ENDING> ::= END CASE

The <EXPRESSION> serves as an index into the list of <EXECUTABLE STATEMENT>s. The statement selected is executed, and the others ignored. Control is then transferred to the statement following the <CASE ENDING> unless, of course, the statement causes a RETURN or an UNDO to some other location.

If there are N number of statements in the list, then the range of the value of the <EXPRESSION> may be from 0 through N-1.

The statements in the list may be any legal <EXECUTABLE STATEMENT> allowed in SDL. If the user wishes to execute nothing in a given case, the <NULL STATEMENT> is an appropriate statement.

REFER STATEMENT

<REFER STATEMENT> ::= REFER <REF VAR> TO <ADDRESS GENERATOR>

<REF VAR>::= <IDENTIFIER>

The statement will make <ADDRESS GENERATOR> become the new referent of <REF VAR>. Since an <ADDRESS GENERATOR> in SDL can locate any arbitrary area of memory (using MAKE.DESCRIPTOR, indexing, etc), the reference variable may do likewise, but in UPL the restriction to a safe subset of <ADDRESS GENERATOR>'s also guarantees the safety of reference variables.

The only exception to this safety is the classic dangling reference problem: Suppose, while executing a lexic level one procedure, that a reference variable declared at lexic level zero is bound to a locally declared referent. If that reference variable is then used after the procedure is exited, its referent will not exist and an unpredictable piece of data or garbage will be accessed.

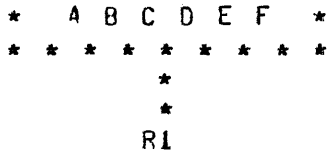
Technically, this error can only be detected at run time, but its occurrence can be precluded altogether by making a strong restriction in the syntax: the lexic level of the <ADDRESS GENERATOR> may not be greater than that of <REF VAR>. This cannot be checked for some <ADDRESS GENERATOR>s, notably MAKE.DESCRIPTOR, but it can be checked in all cases for UPL.

An <ADDRESS GENERATOR>, NULL, is available so that reference variables may be re-bound to such. Testing for NULL is done by checking for length of zero.

REDUCE STATEMENT

```
<REDUCE STATEMENT> ::= REDUCE <OBJECT REFERENCE> <SETTING
                       RESULT REFERENCE PART> UNTIL
                       <FIRST OR LAST> <EQL OR NEQ OR IN>
                       <EXPRESSION>
                       <ON EOS_CYCLE PART>

<OBJECT REFERENCE ::= <IDENTIFIER>

<SETTING RESULT REFERENCE PART> ::=  <EMPTY> | SETTING <RESULT
                                     REFERENCE>

<RESULT REFERENCE> ::=  <IDENTIFER>

<FIRST OR LAST> ::=  FIRST | LAST

<EQL OR NEG OR IN> ::=  EQL | NEQ | IN | = | /=

<ON EOS_CYCLE PART> ::=  <EMPTY> | ON EOS_CYCLE <EXECUTABLE STATEMENT> |
                         ON EOS <EXECUTABLE STATEMENT>
```

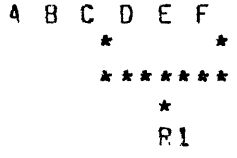
Reduction is a flexible and efficient means for scanning character strings which uses reference variables rather than integers as pointers which select substrings. The basic function of reduction is to truncate a reference variable from the left until its first character satisfies some condition. No change is actually made to the data; the reference variable is simply rebound to a substring of its former referent. For example, the original referent of R1 is a string "ABCDEF".



After the statement

```
REDUCE R1 UNTIL FIRST = "D";
```

is executed the referent of R1 is "DEF".



If the character string deleted is of interest, another reference may be referenced to it by the variation:

```
REDUCE R1 SETTING R2 UNTIL FIRST = "D";
```

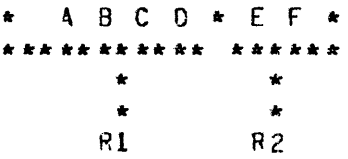
Starting with R1's original referent, "ABCDEF", this leaves

thus dividing the original string according to the condition FIRST = "D".

The entire operation may also be done in reverse (scanning right to left) in which case the last character of R1 must satisfy the condition.

```
REDUCE R1 SETTING R2 UNTIL LAST = "D";
```

results in the new binding



Three types of conditions may be specified:

= scans for a character which is the same as the specified character.

/= scans for a character which is different from the specified character.

IN scans for a character which, when translated to by the specified bit table, yields a @(1)1@. See CHAR-TABLE for a convenient means for specifying bit table constants.

In the first two cases, a single character must be given as a scan argument. In the third case, a bit string of length 256 bits must be given as a table.

The <EXPRESSION> must evaluate to either CHARACTER(1) or BIT(8) or BIT(256) depending upon the condition type. Improper type on this <EXPRESSION> is the only possible run-time error from reduction.

END OF STRING

The REDUCE statement terminates when either a character satisfying the condition is found or the length of the <OBJECT REFERENCE> has been reduced to zero, i.e., it is NULL. Since the latter termination is often of separate interest its occurrence may be detected using syntax analogous to that for detection of special conditions on I/O statements. The syntax was shown above. The <EXECUTABLE STATEMENT> is executed if and only if the original reference has been reduced to NULL. (If a <RESULT REFERENCE> was specified, it will then refer to the original referent of the <OBJECT REFERENCE>.)

Frequently, the end-of-string code will reset the <OBJECT REFERENCE> to some new data, perhaps by reading a new card. In this case, control returns from the EOS_CYCLE back to the REDUCE, thus effecting scanning over record boundaries without additional coding. If the <OBJECT REFERENCE> remains NULL after execution of the EOS_CYCLE code, control passes to the following statement as usual. These semantics may seem awkward at first, but they have the desirable effect of guaranteeing the proper exit conditions of a REDUCE statement--either the condition is satisfied by the first (or last) character of the <OBJECT REFERENCE> or the <OBJECT REFERENCE> is NULL--regardless of whether or not an EOS_CYCLE has been specified. This principle can be violated only by a branch instruction (UNDO, RETURN) in the EOS code.

If ON_EOS is used in place of EOS_CYCLE, then control always passes to the next statement.

MODIFY STATEMENTS (CLEAR, BUMP, DECREMENT)

```
<MODIFY INSTRUCTION> ::=      <CLEAR STATEMENT>
                            | <BUMP STATEMENT>
                            | <DECREMENT STATEMENT>

<CLEAR STATEMENT> ::=         CLEAR <ARRAY IDENTIFIER LIST>

<ARRAY IDENTIFIER LIST> ::=   <ARRAY IDENTIFIER>
                            | <ARRAY IDENTIFIER>,
                              <ARRAY IDENTIFIER LIST>
```

As the syntax indicates, the <CLEAR STATEMENT> may only clear arrays. If the array has been declared bit or fixed, zeroes are moved to each element. If it was declared as character, blanks are moved to each element. Paged arrays may not be cleared.

```
<BUMP STATEMENT> ::=          BUMP <ADDRESS VARIABLE><MODIFIER>

<ADDRESS VARIABLE> ::=        See ADDRESS VARIABLES

<MODIFIER> ::=                <EMPTY>
                            | BY <EXPRESSION>

<DECREMENT STATEMENT> ::=     DECREMENT <ADDRESS VARIABLE><MODIFIER>
```

The bump and decrement statements perform the same functions as their counterparts in the <EXPRESSION> (BUMPOR and DECREMENTOR). See those sections for specific usage. Since these constructs exist as statements in their own rights, and not merely as parts of the <EXPRESSION>, they are included here.

NULL STATEMENT

<NULL STATEMENT> ::= ;

The semi-colon is considered to be a statement in its own right. It may be used in any construct where the syntax requires that an <EXECUTABLE STATEMENT> be present, but the user wishes to execute nothing. It is most commonly used in the <IF STATEMENT> and the <CASE STATEMENT>, but may also be functional in the read, write, and space statements. Refer to the individual descriptions for more specific details.

EXAMPLE:

```
CASE <EXPRESSION>;
     IF <EXPRESSION> THEN;                    %CASE 0
                     ELSE <STATEMENT>;
     ;                                        %CASE 1
     DO;                                      %CASE 2
     <EXECUTABLE STATEMENT LIST>
     END;
END CASE;
```

Notice that the above <CASE STATEMENT> contains three <EXECUTABLE STATEMENT>s: An <IF STATEMENT>, a <NULL STATEMENT>, and a <DO GROUP>. If the value of the <EXPRESSION> following CASE is 1, then nothing is executed. In addition, the ; following THEN is a <NULL STATEMENT>.

FILE ATTRIBUTE STATEMENT (CHANGE STATEMENT)

```
<FILE ATTRIBUTE
STATEMENT> ::=          CHANGE <FILE DESIGNATOR>
                        TO (<DYNAMIC FILE ATTRIBUTE LIST>)

<FILE DESIGNATOR> ::=     <FILE IDENTIFIER>
                        | <SWITCH FILE IDENTIFIER> (<EXPRESSION>)

<DYNAMIC FILE
ATTRIBUTE LIST> ::=       <DYNAMIC FILE ATTRIBUTE>
                        | <DYNAMIC FILE ATTRIBUTE>,
                          <DYNAMIC FILE ATTRIBUTE LIST>

<DYNAMIC FILE
ATTRIBUTE> ::=            <DYNAMIC MULTI-FILE IDENTIFICATION PART>
                        | <DYNAMIC FILE IDENTIFICATION PART>
                        | <DYNAMIC PACK_ID PART>
                        | <DYNAMIC DEVICE PART>
                        | <DYNAMIC TRANSLATION PART>
                        | <DYNAMIC FILE PARITY PART>
                        | <DYNAMIC VARIABLE RECORD PART>
                        | <DYNAMIC LOCK PART>
                        | <DYNAMIC BUFFERS PART>
                        | <DYNAMIC SAVE FACTOR PART>
                        | <DYNAMIC RECORD SIZE PART>
                        | <DYNAMIC RECORDS-PER-BLOCK PART>
                        | <DYNAMIC REEL NUMBER PART>
                        | <DYNAMIC NUMBER-OF-AREAS PART>
                        | <DYNAMIC BLOCKS-PER-AREA PART>
                        | <DYNAMIC ALL-AREAS-AT-OPEN PART>
                        | <DYNAMIC AREA-BY-CYLINDER PART>
                        | <DYNAMIC EU_SPECIAL PART>
                        | <DYNAMIC EU_INCREMENTED PART>
                        | <DYNAMIC USE_INPUT_BLOCKING
                          DESIGNATOR PART>
```

| <DYNAMIC MULTI-PACK PART>
| <DYNAMIC END-OF-PAGE PART>
| <DYNAMIC OPEN-OPTION PART>
| <DYNAMIC REMOTE-KEY PART>
| <DYNAMIC NUMBER-OF-STATIONS PART>
| <DYNAMIC QUEUE-FAMILY-SIZE PART>
| <DYNAMIC FILE TYPE PART>
| <DYNAMIC WORK FILE PART>
| <DYNAMIC LABEL TYPE PART>
| <DYNAMIC INVALID CHARACTER REPORTING PART>
| <DYNAMIC OPTIONAL FILE PART>
| <DYNAMIC SERIAL NUMBER PART>
| <DYNAMIC EXCEPTION MASK PART>
| <DYNAMIC QUEUE SIZE PART>
| <DYNAMIC HEADER PART>
| <DYNAMIC SOFT TRANSLATE PART>
| <DYNAMIC HOST_NAME PART>
| <DYNAMIC OPEN_ON_BEHALF_OF PART>

The <FILE ATTRIBUTE STATEMENT> allows the user to dynamically change the attributes of his file during the execution of his program. This statement may occur at any point in the program, but the change will not become effective until the file is opened. That is, if the file in question is open when the <FILE ATTRIBUTE STATEMENT> is executed, then the change will not occur until the file is closed and re-opened.

Each <DYNAMIC FILE ATTRIBUTE> should be consistent with the format and restrictions of its counterpart listed in the FILE DECLARATIONS. Exceptions to this are specifically stated below.

If a <DYNAMIC FILE ATTRIBUTE> is omitted, the attribute remains as it was previously set.

It should be noted that the following process is mandatory when changing the attributes of an open file which is to be re-opened:

1. Close the file with an attribute which causes space for the FIB to be returned: i.e., LOCK, RELEASE, etc. (If CLOSE is used without attributes, the FIB will not be rebuilt from the FPB, and the attribute will remain unchanged).

2. Change the desired attributes.

3. Open the file.

<DYNAMIC MULTI-FILE IDENTIFICATION PART> ::= MULTI_FILE_ID := <DYNAMIC MULTI-FILE IDENTIFICATION>

<DYNAMIC MULTI-FILE IDENTIFICATION> ::= <EXPRESSION>

<DYNAMIC FILE IDENTIFICATION PART> ::= FILE_ID := <DYNAMIC FILE IDENTIFICATION>

<DYNAMIC FILE IDENTIFICATION> ::= <EXPRESSION>

<DYNAMIC PACK_ID PART> ::= PACK_ID := <DYNAMIC PACK IDENTIFICATION>

<DYNAMIC PACK IDENTIFICATION> ::= <EXPRESSION>

The <EXPRESSION>s of these four attributes are each assumed to be character strings. If they are bits, however, they will be converted to characters in the following manner:

1. The bits are left justified.

2. Trailing blanks are appended. However, if the bits are not a multiple of 8, then the string will appear to be invalid characters.

EXAMPLE:

```
CHANGE F TO (FILE_ID := @F0E@);

WILL RESULT IN THE <FILE IDENTIFICATION>
BEING EQUAL TO:

@F0E4040404040404040@
```

<DYNAMIC DEVICE PART> ::= DEVICE := <DYNAMIC DEVICE SPECIFIER>

<DYNAMIC DEVICE SPECIFIER> ::= <EXPRESSION>

The low-order 10 bits of the <EXPRESSION> must be coded as follows (where the variant is the high order four bits, and the hardware is the low-order six):

DEVICE	HARDWARE	VARIANT
CARD	21	
TAPE	27	
TAPE_9	28	
TAPE_7	25	
TAPE_PE	26	
TAPE_NRZ	24	
DISK	17	0 = SERIAL 1 = RANDOM
DISK_PACK	16	(SAME AS DISK)
DISK_FILE	12	(SAME AS DISK)
DISK_PACK_CENTURY	15	(SAME AS DISK)
DISK_PACK_CAELUS	14	(SAME AS DISK)
PRINTER	8	0 = BACKUP TAPE OR DISK 1 = BACKUP TAPE 2 = BACKUP DISK 3 = BACKUP TAPE OR DISK 4 = HARDWARE ONLY 5 = BACKUP TAPE ONLY 6 = BACKUP DISK ONLY 7 = BACKUP TAPE OR DISK O
PRINTER FORMS	8	8 + PRINTER VARIANT
CARD_READER	21	
CARD_PUNCH	2	(SAME AS PRINTER)
CARD_PUNCH FORMS	2	(SAME AS PRINTER FORMS)
PUNCH	2	(SAME AS PRINTER)
PUNCH FORMS	2	(SAME AS PRINTER FORMS)
READER_PUNCH_PRINTER	5	(SAME AS PRINTER)
READER_PUNCH_PRINTER FORMS	5	(SAME AS PRINTER FORMS)
PUNCH_PRINTER	5	(SAME AS PRINTER)
PUNCH_PRINTER FORMS	5	(SAME AS PRINTER FORMS)
PAPER_TAPE_PUNCH	20	(SAME AS PRINTER)
PAPER_TAPE_PUNCH FORMS	20	(SAME AS PRINTER FORMS)
PAPER_TAPE_READER	6	
READER_96	19	
SORTER_READER	10	
READER_SORTER	10	
CASSETTE	30	
REMOTE	63	
QUEUE	61	

<DYNAMIC TRANSLATION PART> ::= TRANSLATION := <DYNAMIC TRANSLATION SPECIFIER>

<DYNAMIC TRANSLATION SPECIFIER> ::= <EXPRESSION>

The low-order 3 bits of the <EXPRESSION> determines the translation as follows:

000 = EBCDIC

001 = ASCII

010 = BCL

<DYNAMIC OPEN-OPTION PART>::= OPEN_OPTION := <DYNAMIC OPEN_OPTION SPECIFIER>

<DYNAMIC OPEN-OPTION SPECIFIER>::= <EXPRESSION>

The low-order 12 bits of the expression determine the type of open as follows (bits are numbered from left to right within the 12):

BIT		FUNCTION (IF 1)
0	=	INPUT
1	=	OUTPUT
2	=	NEW
3	=	PUNCH
4	=	PRINT
5	=	NO_REWIND, INTERPRET
6	=	REVERSE, STACKERS
7	=	LOCK
8	=	LOCK_OUT

<DYNAMIC PARITY PART> ::= PARITY := <DYNAMIC PARITY SPECIFIER>

```
<DYNAMIC PARITY
SPECIFIER> ::=                  <EXPRESSION>

<DYNAMIC VARIABLE
RECORD PART> ::=                VARIABLE :=
                                <DYNAMIC VARIABLE RECORD SPECIFIER>

<DYNAMIC VARIABLE
RECORD SPECIFIER> ::=           <EXPRESSION>

<DYNAMIC LOCK PART> ::=         LOCK := <DYNAMIC LOCK SPECIFIER>

<DYNAMIC LOCK
SPECIFIER> ::=                  <EXPRESSION>

<DYNAMIC ALL-AREAS-
AT-OPEN PART> ::=               ALL_AREAS_AT_OPEN :=
                                <DYNAMIC ALL-AREAS-AT-OPEN SPECIFIER>

<DYNAMIC ALL-AREAS-
AT-OPEN SPECIFIER> ::=          <EXPRESSION

<DYNAMIC AREA-BY
CYLINDER PART> ::=              AREA_BY_CYLINDER :=
                                <DYNAMIC AREA-BY-CYLINDER SPECIFIER>

<DYNAMIC AREA-BY-
CYLINDER SPECIFIER> ::=         <EXPRESSION>

<DYNAMIC USE_INPUT_
BLOCKING PART> ::=              USE_INPUT_BLOCKING :=
                                <DYNAMIC USE_INPUT_BLOCKING SPECIFIER>

<DYNAMIC USE_INPUT_
BLOCKING SPECIFIER> ::=         <EXPRESSION>

<DYNAMIC END-OF-
PAGE PART> ::=                  END_OF_PAGE_ACTION :=
                                <DYNAMIC END-OF-PAGE SPECIFIER>

<DYNAMIC END-OF-
PAGE SPECIFIER> ::=             <EXPRESSION>

<DYNAMIC MULTI-
PACK PART>::=                   MULTI_PACK :=
                                <DYNAMIC MULTI-PACK SPECIFIER>

<DYNAMIC MULTI-
PACK SPECIFIER> ::=             <EXPRESSION>

<DYNAMIC REMOTE-
KEY PART>::=                    REMOTE_KEY :=
                                <DYNAMIC REMOTE-KEY SPECIFIER>
```

<DYNAMIC REMOTE-
KEY SPECIFIER>::= <EXPRESSION>

<DYNAMIC WORK
FILE PART>::= WORK_FILE :=
<DYNAMIC WORK FILE SPECIFIER>

<DYNAMIC WORK
FILE SPECIFIER>::= <EXPRESSION>

Only the low-order bit of each of the above <expression>s is used to determine the value of the attribute. The code definitions are as follows:

PARITY	0 = ODD 1 = EVEN
VARIABLE	0 = FIXED 1 = VARIABLE
LOCK	0 = NOT LOCKED 1 = LOCKED
ALL_AREAS_AT_OPEN	0 = ALLOCATE AREAS AS NEEDED 1 = ALLOCATE ALL SPACE AT OPEN TIME
AREA_BY_CYLINDER	0 = PUT AREA ANYWHERE ON DISK 1 = ONE AREA PER CYLINDER AT BEGINNING
USE_INPUT_BLOCKING	0 = TAKE ATTRIBUTES FROM FILE DECLARATION 1 = TAKE ATTRIBUTES FROM DISK FILE HEADER See FILE ATTRIBUTES
END_OF_PAGE_ACTION	0 = NO DETECTION OF END-OF-PAGE 1 = BRANCH TO <EOF PART> OF <WRITE STATEMENT> AT END OF PAGE ON PRINTER FILE
MULTI_PACK	1 = PLACE FILE ON MULTIPLE DISK PACKS 0 = PLACE FILE ON SINGLE DISK PACK
REMOTE KEY	1 = REMOTE KEY IS PRESENT ON ALL READS AND WRITES TO THE FILE 0 = REMOTE KEY IS NOT PRESENT
WORK_FILE	1 = INSERT JOB NUMBER IN FILE IDENTIFIER 0 = LEAVE FILE IDENTIFIER ALONE

<DYNAMIC EU_SPECIAL
PART> ::= EU_SPECIAL :=
<DYNAMIC EU_SPECIAL SPECIFIER>
| EU_SPECIAL :=
<DYNAMIC EU_SPECIAL SPECIFIER>
EU_DRIVE :=
<DYNAMIC EU_SPECIAL SPECIFIER>

<DYNAMIC EU_SPECIAL
SPECIFIER> ::= <EXPRESSION>

<DYNAMIC EU_DRIVE SPECIFIER> ::= <EXPRESSION>

<DYNAMIC EU_INCREMENTED PART> ::= EU_INCREMENTED :=
<DYNAMIC EU_INCREMENTED SPECIFIER>
| EU_INCREMENTED :=
<DYNAMIC EU_INCREMENTED SPECIFIER>,
EU_INCREMENT :=
<DYNAMIC EU_INCREMENT SPECIFIER>

<DYNAMIC EU_INCREMENTED SPECIFIER> ::= <EXPRESSION>

<DYNAMIC EU_INCREMENT SPECIFIER> <EXPRESSION>

The low-order bit of the EU_SPECIAL and EU_INCREMENTED specifiers serves to indicate whether or not the attribute is set (0=Off, 1=On). If the attribute is off, then inclusion of the EU_DRIVE and EU_INCREMENT specifiers is unnecessary.

If these attributes are set on, then the drive and increment parts should be included, and should conform to the specifications in the FILE DECLARATIONS. If omitted, the <DYNAMIC EU_DRIVE SPECIFIER> is not changed. If the <DYNAMIC EU_INCREMENT SPECIFIER> has never been set (i.e., it is 0), then it is set to one; otherwise, it too remains unchanged.

```
<DYNAMIC BUFFERS PART> ::=    BUFFERS := <DYNAMIC NUMBER OF BUFFERS>

<DYNAMIC NUMBER
OF BUFFERS> ::=               <EXPRESSION>

<DYNAMIC SAVE
FACTOR PART> ::=              SAVE := <DYNAMIC SAVE FACTOR>

<DYNAMIC SAVE FACTOR> ::=     <EXPRESSION>

<DYNAMIC RECORD
SIZE PART> ::=                RECORD_SIZE := <DYNAMIC RECORD SIZE>

<DYNAMIC RECORD SIZE> ::=     <EXPRESSION>

<DYNAMIC RECORDS-
PER-BLOCK PART> ::=           RECORDS_PER_BLOCK :=
                              <DYNAMIC RECORDS-PER-BLOCK>

<DYNAMIC RECORDS-
PER-BLOCK> ::=                <EXPRESSION>

<DYNAMIC REEL
NUMBER PART> ::=              REEL := <DYNAMIC REEL NUMBER>

<DYNAMIC REEL NUMBER> ::=     <EXPRESSION>

<DYNAMIC NUMBER-OF-
AREAS PART> ::=               NUMBER_OF_AREAS :=
                              <DYNAMIC NUMBER-OF-AREAS>

<DYNAMIC NUMBER-
OF-AREAS> ::=                 <EXPRESSION>

<DYNAMIC BLOCKS-PER-
AREA PART> ::=                BLOCKS_PER_AREA :=
                              <DYNAMIC BLOCKS-PER-AREA>

<DYNAMIC BLOCKS-PER
AREA> ::=                     <EXPRESSION>

<DYNAMIC QUEUE-FAMILY-
SIZE PART>::=                 QUEUE_FAMILY_SIZE :=
                              <DYNAMIC QUEUE-FAMILY-SIZE>

<DYNAMIC QUEUE-
FAMILY-SIZE>::=               <EXPRESSION>

<DYNAMIC NUMBER-OF-
STATIONS PART>::=             NUMBER_OF_STATIONS :=
                              <DYNAMIC NUMBER-OF-STATIONS SPECIFIER>

<DYNAMIC NUMBER-OF-
STATIONS SPECIFIER>::=        <EXPRESSION>
```

The above <EXPRESSION>s return a bit string which should be consistent with the formats and restrictions listed in the FILE DECLARATIONS_

<DYNAMIC FILE TYPE PART>::= FILE_TYPE :=
<DYNAMIC FILE TYPE SPECIFIER>

<DYNAMIC FILE TYPE SPECIFIER>::= <EXPRESSION>

The value of the expression determines the file type:

VALUE	TYPE
0	DATA
7	INTERPRETER
8	CODE
9	DATA
12	INTRINSIC

<DYNAMIC LABEL TYPE PART>::= LABEL_TYPE :=
<DYNAMIC LABEL TYPE SPECIFIER>

<DYNAMIC LABEL TYPE SPECIFIER>::= <EXPRESSION>

The value of the expression determines the label type.

VALUE	TYPE
0	ANSII
1	UNLABELED
2	BURROUGHS STANDARD

<DYNAMIC INVALID CHARACTER REPORTING> ::= INVALID_CHARACTERS :=
<DYNAMIC INVALID CHARACTER REPORT TYPE>

<DYNAMIC INVALID CHARACTER REPORTING TYPE> ::= <EXPRESSION>

The value of the expression determines the type of reporting:

VALUE	TYPE
0	Report all lines containing invalid characters.
1	Report all lines containing invalid characters and ther stop program.
2	Report once that the file contains invalid characters.
3	Don't report that the file contains invalid characters.

<DYNAMIC OPTIONAL FILE PART> ::= OPTIONAL := <EXPRESSION>

The low-order bit of the expression determines whether or not the file may be optional. If the value is 1, the file may be optional; if 0, it must be present.

<DYNAMIC SERIAL NUMBER PART> ::= SERIAL := <EXPRESSION>

The expression should generate a 6-character string, each of the characters of which are a decimal digit. This number will be used as the tape serial number.

<DYNAMIC EXCEPTION MASK PART> ::= EXCEPTION_MASK := <EXPRESSION>

The low order 24 bits of the value of the expression will be used as the EXCEPTION MASK. See <EXCEPTION MASK PART> under <FILE DECLARATION STATEMENT> in Section 6.

<DYNAMIC QUEUE SIZE PART> ::= QUEUE_MAX_MESSAGES := <EXPRESSION>

Sets size for queue files.

<DYNAMIC HEADER PART> ::= REMOTE_HEADERS := <EXPRESSION>

Sets headers boolean for remote files.

<DYNAMIC SOFT TRANSLATE PART> ::= TRANSLATE := <EXPRESSION>
| TRANSLATE_FILE := <EXPRESSION>

TRANSLATE sets a boolean, turning the translation option on or off while TRANSLATE_FILE changes the file-id of the translate table file.

<DYNAMIC HOST_NAME PART>::= HOST_NAME:= <EXPRESSION>

Sets Host name for BNA.

<DYNAMIC OPEN_ON_BEHALF_OF PART>::= OPEN_ON_BEHALF_OF:= <EXPRESSION>

Turns the OPEN_ON_BEHALF_OF Boolean on or off.

STOP STATEMENT

```
<STOP STATEMENT> ::=    STOP
                      | STOP <EXPRESSION>
```

The <STOP STATEMENT> is a communicate to the MCP that the program has finished. It should not be confused with FINI which is the final statement in the program.

STOP <EXPRESSION> is intended for use by the compilers only. The <EXPRESSION> communicates the number of syntax errors to the MCP.

ZIP STATEMENT

<ZIP STATEMENT> ::= ZIP <EXPRESSION>

The <ZIP STATEMENT> allows the user to pass control instructions to the MCP. The <EXPRESSION> should generate a character string whose value is a valid MCP control statement as defined in the B1700 Software Operational Guide.

SEARCH_DIRECTORY STATEMENT

No on behalf of ?

```
<SEARCH STATEMENT> ::=      <SEARCH PART>; <ON FILE PART>

<SEARCH PART> ::=           SEARCH_DIRECTORY (<SEARCH OBJECT>,
                            <SEARCH RESULT>,<SEARCH RESULT MODE>

<SEARCH OBJECT> ::=         <ADDRESS GENERATOR>

<SEARCH RESULT> ::=         <ADDRESS GENERATOR>

<SEARCH RESULT MODE> ::=    BIT | CHARACTER

<ON FILE PART> ::=          <EMPTY> | ON FILE_MISSING <EXECUTABLE
                             STATEMENT>
                          | ON FILE_LOCKED <EXECUTABLE STATEMENT>
                          | ON FILE_MISSING <EXECUTABLE STATEMENT>;
                            ON FILE_LOCKED <EXECUTABLE STATEMENT>
                          | ON FILE_LOCKED <EXECUTABLE STATEMENT>;
                            ON FILE MISSING <EXECUTABLE STATEMENT>
```

The <SEARCH STATEMENT> allows the user to extract certain information contained in the disk file header specified by the <SEARCH OBJECT>.

The <SEARCH OBJECT> is expected to be 30 characters in length where the first 10 characters are the pack identification, the second 10 characters are the multi-file identification, and the third 10 are the file identification. File names less than 10 characters must be left-justified in their respective fields with trailing blanks appended. If only one file name exists, that name should be left-justified in the multi-file identification field, and the file identification should be blank.

The <SEARCH RESULT> specifies the receiving field and should be 360 bits long if bit mode is specified, or 59 bytes if character mode is specified.

The information is returned in the following format:

ACCESS-DFH @00E@

```
01  FILE_HEADER_FORMAT,
      02  OPEN_TYPE              BIT (24),  %  CHARACTER (1)
      02  NO_USERS               BIT (24),  %  CHARACTER (2)
      02  RECORD_SIZE            BIT (24),  %  CHARACTER (4)
      02  RECORDS_PER_BLOCK      BIT (24),  %  CHARACTER (4)
      02  EOF_POINTER            BIT (24),  %  CHARACTER (8)
      02  SEGMENTS_PER_AREA      BIT (24),  %  CHARACTER (8)
      02  USER_OPEN_OUTPUT       BIT (24),  %  CHARACTER (1)
      02  FILE_TYPE              BIT (24),  %  CHARACTER (2)
      02  PERMANENT_FLAG         BIT (24),  %  CHARACTER (2)
      02  BLOCKS_PER_AREA        BIT (24),  %  CHARACTER (6)
      02  AREAS_REQUESTED        BIT (24),  %  CHARACTER (3)
      02  AREA_COUNTER           BIT (24),  %  CHARACTER (3)
      02  SAVE_FACTOR            BIT (24),  %  CHARACTER (3)
      02  CREATION_DATE          BIT (24),  %  CHARACTER (5)
      02  LAST_ACCESS_DATE       BIT (24),  %  CHARACTER (5)
```

Note: This format may be subject to change.

The <FILE MISSING PART> and <FILE LOCKED PART> allow the user to specify the course of action should either of these conditions arise.

READ_FILE_HEADER, WRITE_FILE_HEADER

```
<ACCESS FILE HEADER
STATEMENT> ::=          <ACCESS FILE HEADER PART>;
                      | <ACCESS FILE HEADER PART>;
                        <FILE MISSING PART>
                      | <ACCESS FILE HEADER PART>;
                        <FILE LOCKED PART>
                      | <ACCESS FILE HEADER PART>;
                        <FILE MISSING PART>
                        <FILE LOCKED PART>

<ACCESS FILE HEADER
PART> ::=               READ_FILE_HEADER
                        (<FILE NAME>, <DESTINATION FIELD>)
                      | WRITE_FILE_HEADER
                        (<FILE NAME>, <SOURCE FIELD>)

<FILE NAME> ::=         <ADDRESS GENERATOR>

<DESTINATION FIELD> ::= <ADDRESS GENERATOR>

<SOURCE FIELD> ::=      <ADDRESS GENERATOR>

<FILE MISSING PART> ::= ON FILE_MISSING <EXECUTABLE STATEMENT>

<FILE LOCKED PART> ::=  ON FILE_LOCKED <EXECUTABLE STATEMENT>
```

The <ACCESS FILE HEADER STATEMENT> is intended for use in systems programs only. It enables the programmer to either read or write a file header.

The <FILE NAME> is expected to be a 30-character field where the first 10 characters are the PACK_ID, the second 10 characters are the MULTI-FILE IDENTIFICATION and the third 10, the FILE IDENTIFICATION. File names less than 10 characters are left-justified in their respective fields. If only one file name exists, it is left-justified in the multi-file identification, and the file identification should be set to blanks.

The <SOURCE FIELD> or <DESTINATION FIELD> specifies, respectively, the sending or receiving field, and is expected to be 576 to 4320 bits in length depending upon the number of areas allocated. Information is passed in the file header format. Refer to the 81700 MCP Manual for specifics.

The <FILE MISSING PART> and <FILE LOCKED PART> enable the programmer to specify the course of action should either of these conditions arise.

Note that extreme caution is advised when writing a file header.

MAKE_READ_ONLY, MAKE_READ_WRITE

```
<ARRAY PAGE TYPE
STATEMENT> ::=          <ARRAY PAGE TYPE DESIGNATOR>
                        (<PAGED ARRAY NAME>,<PAGE NUMBER>)

<ARRAY PAGE TYPE
DESIGNATOR> ::=           MAKE_READ_ONLY
                        | MAKE_READ_WRITE

<PAGED ARRAY NAME> ::=  <IDENTIFIER>

<PAGE NUMBER> ::=       <EXPRESSION>
```

The <ARRAY PAGE TYPE STATEMENT> allows the user to mark certain paged array pages as READ-ONLY. When this is done, a page will not be written out to disk every time it is overlaid.

MAKE_READ_WRITE allows the user to change information on a paged array, and to have that array written on disk when it is overlaid. It is only necessary to specify MAKE_READ_WRITE after a MAKE_READ_ONLY specification.

It is the programmer's responsibility to ensure that the information in a page marked READ-ONLY is not changed. In addition, the user is responsible for guaranteeing correct page number specifications. There is no syntax check for either.

EXAMPLE:

```
DECLARE PAGED (32) P (1024) BIT(30), T1 FIXED
T1 := -1;
DO FOREVER;
   MAKE_READ_ONLY (P, BUMP T1);
   IF T1 = 31 THEN UNDO;
END;
 .
 .
 .
MAKE_READ_WRITE (P, 0);
```

COROUTINE STATEMENT

```
<COROUTINE STATEMENT>::=      <COROUTINE ENTRY STATEMENT>
                            | <COROUTINE EXIT STATEMENT>

<COROUTINE
ENTRY STATEMENT>::=           ENTER_COROUTINE
                              (<COROUTINE TABLE SPECIFIER>)

<COROUTINE
TABLE SPECIFIER> ::=          <ADDRESS GENERATOR>

<COROUTINE
EXIT STATEMENT>::=            EXIT_COROUTINE
                              (<COROUTINE TABLE SPECIFIER>)
```

The <COROUTINE TABLE SPECIFIER> associated with ENTER_COROUTINE and EXIT_COROUTINE is assumed to describe a table with the following format:

```
DECLARE
     01  TABLE
          ,02  NUMBER_OF_ENTRIES BIT(4)
          ,02  ENTRY_ADDRESS BIT(32)
          ,02  PPS_COPY(16) BIT(32)
     ;
```

A. ENTER_COROUTINE: The <COROUTINE TABLE SPECIFIER> is assumed to have the format described above. The current code address is pushed on to the Program Pointer Stack. The number of elements of PPS.COPY that is specified by NUMBER_OF_ENTRIES is pushed onto the Program Pointer Stack. The address of the next instruction is taken from ENTRY_ADDRESS.

B. EXIT_COROUTINE: The <COROUTINE TABLE SPECIFIER> is assumed to describe a table of the format given above. The current nesting level is stored in NUMBER_OF_ENTRIES. The current code address is stored in ENTRY_ADDRESS. The number (as specified by NUMBER_OF_ENTRIES) of entries on the top of the Program Pointer Stack is copied to PPS_COPY(0) through PPS_COPY(NUMBER_OF_ENTRIES-1). If NUMBER_OF_ENTRIES is 0, then nothing is copied. An UNDO is performed, using NUMBER_OF_ENTRIES as the number of entries on top of the Program Pointer Stack.

Note: Upon first execution of ENTER_COROUTINE, the table must already be set up. The easiest way to accomplish this is to make the first executable statement in the coroutine to be entered an EXIT.COROUTINE statement. The first entrance to the coroutine is then accomplished by a call statement.

Note: This is not a general coroutine mechanism--i.e., It is not symmetric. The routine executing the ENTER_COROUTINE is a master to the slave routine which contains the EXIT_COROUTINE'S.

Note: EXIT_COROUTINE can only appear within procedures with no parameters and no local data; i.e., those procedures which do not change the Control Stack.

EXAMPLE:

```
DECLARE I FIXED;                          will display   "000003"  (1)
DECLARE TABLE BIT(4+17*32);                              "000005"  (2)
PROCEDURE SLAVE;                                         "000008"  (3)
    EXIT_COROUTINE(TABLE); %SETS UP TABLE                "000010"  (4)
    DO FOREVER;                                            "
         BUMP I BY 2;                                      "
         DISPLAY DECIMAL(I,6);                             "
         EXIT_COROUTINE(TABLE); %RESETS TABLE              "
    END;                                                   "
END SLAVE;                                                 "
PROCEDURE MASTER;                                        "5*n" (2n)
    SLAVE; %CALL FOR SETUP                               "5*n+3" (2n+1)
    I := 0;                                                "
    DO FOREVER;                                            "
         BUMP I BY 3;                                      "
         DISPLAY DECIMAL(I,6);                             "
         ENTER_COROUTINE(TABLE); %USES TABLE               "
    END;
END MASTER;
```

EXECUTE-PROCEDURE STATEMENT

```
<EXECUTE-PROCEDURE
STATEMENT> ::=                  <NON-TYPED PROCEDURE DESIGNATOR>

<NON-TYPED PROCEDURE
DESIGNATOR> ::=                 <NON-TYPED PROCEDURE IDENTIFIER>
                                <ACTUAL PARAMETER PART>

<NON-TYPED PROCEDURE
IDENTIFIER> ::=                 <IDENTIFIER>

<ACTUAL PARAMETER PART> ::=     <EMPTY>
                              | (<ACTUAL PARAMETER LIST>)

<ACTUAL PARAMETER LIST> ::=     <ACTUAL PARAMETER>
                              | <ACTUAL PARAMETER>,
                                <ACTUAL PARAMETER LIST>

<ACTUAL PARAMETER> ::=          <EXPRESSION>
                              | <ARRAY DESIGNATOR>

<ARRAY DESIGNATOR> ::=          <ARRAY IDENTIFIER>
```

A non-typed procedure, i.e., a procedure which performs a function and does not return a value, is invoked through an <EXECUTE-PROCEDURE STATEMENT>. The name of the procedure is followed by its parameters enclosed in parens. Refer to the section ADDRESS AND VALUE PARAMETERS for information concerning passing parameters.

For a description of the invocation of typed procedures, see VALUE VARIABLES.

EXECUTE-FUNCTION STATEMENT

```
<EXECUTE-FUNCTION
STATEMENT> ::=                  <FUNCTION DESIGNATOR>

<FUNCTION DESIGNATOR> ::=       <ACCESS FILE INFORMATION DESIGNATOR>
                              | <CHANGE STACK SIZE DESIGNATOR>
                              | <CHARACTER FILL DESIGNATOR>
                              | <COMMUNICATE DESIGNATOR>
                              | <COMPILE-CARD-INFO DESIGNATOR>
                              | <DC_INITIATE_IO DESIGNATOR>
                              | <DEBLANK DESIGNATOR>
                              | <DISABLE_INTERRUPTS DESIGNATOR>
                              | <DUMP DESIGNATOR>
                              | <DUMP-FOR-ANALYSIS DESIGNATOR>
                              | <ENABLE_INTERRUPTS DESIGNATOR>
                              | <ERROR COMMUNICATE DESIGNATOR>
                              | <EXECUTE DESIGNATOR>
                              | <FETCH DESIGNATOR>
                              | <FIND DUPLICATE CHARACTERS DESIGNATOR>
                              | <FREEZE-PROGRAM DESIGNATOR>
                              | <GROW DESIGNATOR>
                              | <HALT DESIGNATOR>
                              | <HARDWARE MONITOR DESIGNATOR>
                              | <INITIALIZE_VECTOR DESIGNATOR>
                              | <MESSAGE COUNT DESIGNATOR>
                              | <MONITOR DESIGNATOR>
                              | <OVERLAY DESIGNATOR>
                              | <READ CASSETTE DESIGNATOR>
                              | <ACCESS-FPB DESIGNATOR>
                              | <REFER_ADDRESS DESIGNATOR>
                              | <REFER_LENGTH DESIGNATOR>
                              | <REFER_TYPE DESIGNATOR>
                              | <REINSTATE DESIGNATOR>
                              | <RESTORE DESIGNATOR>
                              | <REVERSE DESIGNATOR>
                              | <SAVE DESIGNATOR>
                              | <SAVE_STATE DESIGNATOR>
                              | <SORT DESIGNATOR>
                              | <SORT_MERGE DESIGNATOR>
                              | <SORT_SWAP DESIGNATOR>
                              | <THAW_PROGRAM DESIGNATOR>
                              | <THREAD_VECTOR DESIGNATOR>
                              | <TRACE DESIGNATOR>
                              | <TRANSLATE DESIGNATOR>
```

ACCESS_FILE_INFORMATION

```
<ACCESS FILE INFORMATION
DESIGNATOR> ::=          ACCESS_FILE_INFORMATION (<FILE DESIGNATOR>,
                         <RETURN TYPE>, <DESTINATION>
<FILE DESIGNATOR> ::=    <FILE IDENTIFIER>
                       | <SWITCH FILE IDENTIFIER> (<EXPRESSION>)

<RETURN TYPE> ::=        BIT / CHARACTER

<DESTINATION> ::=        <ADDRESS GENERATOR>
```

The <ACCESS FILE INFORMATION DESIGNATOR> returns the end-of-file pointer and the device type from the FIB of the specified file to the specified destination.

The information may be returned as either bit or character. The format is as follows:

```
01  DESTINATION_FIELD,
    02 EOF_POINTER      BIT(24),   %  CHARACTER(8)
    02 DEVICE_TYPE      BIT(6);    %  CHARACTER(2)
```

To insure that the FIB exists, this communicate should only be used on open files.

CHANGE_STACK_SIZES

```
<CHANGE STACK
SIZES DESIGNATOR> ::=    CHANGE_STACK_SIZES (<VSSIZE>,
                         <NSSIZE>, <CSSIZE>, <ESSIZE>,
                         <PPSSIZE>, <DYNAMIC SIZE>)

<VSSIZE> ::=             <NUMBER>

<NSSIZE> ::=             <NUMBER>

<CSSIZE> ::=             <NUMBER>

<ESSIZE> ::=             <NUMBER>

<PPSSIZE> ::=            <NUMBER>

<DYNAMIC SIZE> ::=       <NUMBER>
```

This statement is restricted to Lexic Level Zero of programs with no global data. Also, due to technical incompatibilities, it may not be used in a program that invokes profiling, timing, or monitoring facilities. Note that the parameters are in an order corresponding to the order of the stacks in memory.

The result of the execution of the statement is to change the program's stack sizes to the values given.

CHARACTER_FILL

<CHARACTER FILL DESIGNATOR> ::= CHARACTER_FILL (<OF DESTINATION>, <OF SOURCE>)

<OF DESTINATION> ::= <ADDRESS GENERATOR>

<OF SOURCE> ::= <EXPRESSION>

The high-order 8 bits of the <CF SOURCE> will be spread throughout the <CF DESTINATION>.

COMMUNICATE

<COMMUNICATE DESIGNATOR>::= COMMUNICATE (<EXPRESSION>)

The <EXPRESSION> is expected to be a valid communicate message. This is intended only for experimental testing of communicates.

COMPILE_CARD_INFO

<COMPILE-CARD-INFO DESIGNATOR>::= COMPILE_CARD_INFO (<CCI DESTINATION FIELD>)

<CCI DESTINATION FIELD>::= <ADDRESS GENERATOR>

This function is intended for use by the compilers only. The information on the compile card is returned in the following format:

```
OBJECT NAME                                                  CHARACTER (30)
EXECUTE TYPE (DECIMAL)                                       CHARACTER (2)

     01  EXECUTE
     02  COMPILE AND GO
     03  COMPILE FOR SYNTAX
     04  COMPILE TO LIBRARY
     05  COMPILE AND SAVE
     06  GO PART OF COMPILE AND GO
     07  GO PART OF COMPILE AND SAVE
COMPILER PACK IDENTIFIER                                     CHARACTER (10)
COMPILER INTERPRETER NAME                                    CHARACTER (30)
COMPILER INTRINSIC NAME                                      CHARACTER (10)
COMPILER PRIORITY (DECIMAL)                                  CHARACTER (2)
COMPILER SESSION NUMBER                                      CHARACTER (6)
COMPILER JOB NUMBER (DECIMAL)                                CHARACTER (6)
COMPILER 1ST AND 2ND NAMES OF RUNNING PROGRAM                CHARACTER (20)
COMPILER CHARGE NUMBER                                       CHARACTER (7)
FILLER                                                       CHARACTER (1)
COMPILATION DATE AND TIME COMPILED                           BIT (36)
FILLER                                                       BIT(4)
COMPILER USERCODE                                            CHARACTER (10)
COMPILER PASSWORD                                            CHARACTER (10)
COMPILER PARENT JOB NUMBER                                   CHARACTER (04)
COMPILER PARENT QUEUE IDENTIFIER                             CHARACTER (20)
COMPILER LOG SPO                                             CHARACTER (1)
```

DC_INITIATE_IO

```
<DC_INITIATE_IO
DESIGNATOR> ::=          DC_INITITATE_IO (<PORT>, <CHANNEL>,
                         <IO DESC ADDRESS>

<PORT> ::=               <EXPRESSION>

<CHANNEL> ::=            <EXPRESSION>

<IO DESC ADDRESS> ::=    <EXPRESSION>
```

See MCP documentation for DC_INITIATE_IO (communicate verb 40).

DEBLANK

<DEBLANK DESIGNATOR>::= DEBLANK (<FIRST CHARACTER>)

<FIRST CHARACTER>::= <IDENTIFIER>

The <FIRST CHARACTER> is a simple identifier which describes the first character to be examined. Deblank repeatedly increments the address field of the descriptor for <FIRST CHARACTER> until <FIRST CHARACTER> describes a non-blank character.

DISABLE_INTERRUPTS

<DISABLE_INTERRUPTS DESIGNATOR> ::= DISABLE_INTERRUPTS

For MCP use only.

The <DISABLE INTERRUPTS DESIGNATOR> suppresses all interrupts until an <ENABLE INTERRUPTS DESIGNATOR> is encountered.

Note that this construct cannot be executed by normal state programs.

DUMP

<DUMP DESIGNATOR> ::= DUMP

The MCP will create a dumpfile, and program execution will continue after the dump.

DUMP_FOR_ANALYSIS

<DUMP-FOR-
ANALYSIS DESIGNATOR>::= DUMP_FOR_ANALYSIS

Execution of this function will cause a dumpfile to be created and execution to continue.

ENABLE_INTERRUPTS

<ENABLE_INTERRUPTS
DESIGNATOR> ::= ENABLE_INTERRUPTS

For MCP use only.

The <ENABLE INTERRUPTS DESIGNATOR> causes the MCP to return to the normal interrupt-processing mode after the <DISABLE INTERRUPTS DESIGNATOR> has changed that mode. See above.

Note that this construct cannot be executed by a normal state program.

ERROR_COMMUNICATE

<ERROR COMMUNICATE
DESIGNATOR> ::= ERROR_COMMUNICATE (<EXPRESSION>)

The value of the expression should be in the following form:

2 BITS	6 BITS	16 BITS	24 BITS
0	N	0	0

where N is the error number.

The value of the expression will be put on the Evaluation Stack as a descriptor, and an MCP communicate will be performed.

If N = 29 then the MCP will use the 16-bit field as a bit length and the 24-bit field as a base relative bit address of the error message to be printed on the SPO. Otherwise, N is the MCP-defined error message number.

EXECUTE

See <EXECUTE OPERATOR DESIGNATOR> in Section 8.

FETCH

<FETCH DESIGNATOR> ::= <FETCH SPECIFIER> (<I/O REFERENCE ADDRESS>, <PORT, CHANNEL ADDRESS>, <RESULT DESCRIPTOR ADDRESS>)

<FETCH SPECIFIER>::= FETCH | FETCH_AND_SAVE

<I/O REFERENCE ADDRESS> ::= <EXPRESSION>

<PORT,CHANNEL ADDRESS> ::= <ADDRESS GENERATOR>

<ADDRESS GENERATOR> ::= See ADDRESS GENERATORS

<RESULT DESCRIPTOR ADDRESS> ::= <ADDRESS GENERATOR>

The <FETCH DESIGNATOR> fetches the result of an I/O operation. If there is a high priority interrupt, then that interrupt will be reported. Otherwise, if the <I/O REFERENCE ADDRESS> is non-zero, then only an interrupt on an I/O descriptor with the reference address the same as the <I/O REFERENCE ADDRESS> will be reported. The PORT (3 BITS) and CHANNEL (4 BITS) of the interrupt are stored from left to right in the low-order 7 bits of <PORT, CHANNEL ADDRESS>. The I/O RESULT DESCRIPTOR REFERENCE ADDRESS is stored in the low-order 24 bits of the <RESULT DESCRIPTOR ADDRESS>. If there were no interrupts, then these two fields will be zero. FETCH_AND_SAVE is obsolete as of the 5_1 release.

<FIND DUPLICATE CHARACTERS DESIGNATOR> ::= FIND_DUPLICATE_CHARACTERS (<FDC TEXT> , <DUPLICATE COUNT>, <DUPLICATE CHARACTER> , <NON-DUPLICATE TEXT>)

<FDC TEXT> ::= <SIMPLE IDENTIFIER>

<DUPLICATE COUNT> ::= <ADDRESS GENERATOR>

<DUPLICATE CHARACTER> ::= <ADDRESS GENERATOR>

<NON-DUPLICATE TEXT> ::= <SIMPLE IDENTIFIER>

The text to be scanned for contiguous duplicate characters is described initially by <FDC TEXT>. The text will be scanned until three or more contiguous duplicates are found. Upon return, <FDC TEXT>'s descriptor will be reduced to describe the text beyond the duplicate; <NON-DUPLICATE TEXT>'s descriptor will be modified to describe the non-duplicate text that was scanned; <DUPLICATE COUNT> will contain the number of duplicate characters; and <DUPLICATE CHARACTER> will describe the duplicate character.

FREEZE_PROGRAM

<FREEZE-PROGRAM DESIGNATOR>::= FREEZE_PROGRAM

Execution of this function will prevent the program from being moved in memory or from being rolled out of memory.

GROW

<GROW DESIGNATOR>::= GROW (<PAGED ARRAY IDENTIFIER>, <EXPRESSION>)

This statement dynamically increases the array bound of the specified paged array by the value of the expression. The expression may not be negative (the bound may not be decreased) and the resulting array bound must not be larger than 16277215.

HALT

<HALT DESIGNATOR> ::= HALT (<EXPRESSION>)

The <HALT DESIGNATOR> causes the value of the <EXPRESSION> to be moved to the M-Machine T-Register. If the value is longer than 24 bits, only the low-order 24 bits are moved. If the value is less than 24 bits, the value is right-justified and leading zeroes are added.

After the value is moved, an M-Machine halt is executed.

EXAMPLES:

```
DECLARE X BIT(24);
HALT (X:=HEX_SEQUENCE_NUMBER);

HALT (SUBBIT (HEX_SEQUENCE_NUMBER, 0, 24));
```

HARDWARE_MONITOR

<HARDWARE MONITOR DESIGNATOR> ::= HARDWARE_MONITOR (<EXPRESSION>)

The monitor micro-opcode will be executed using the low-order 8 bits of the <EXPRESSION> as its operand.

INITIALIZE_VECTOR

<INITIALIZE_VECTOR DESIGNATOR> ::= INITIALIZE_VECTOR (<TABLE ADDRESS>)

<TABLE ADDRESS> ::= <ADDRESS GENERATOR>

For use by SORT only.

The <TABLE ADDRESS> points to the table containing the vector address, the vector level-1 address, the key table address, and the vector limit address.

MESSAGE_COUNT

<MESSAGE_COUNT DESIGNATOR> ::= MESSAGE_COUNT (FILE DESIGNATOR>, <ADDRESS GENERATOR>

<FILE DESIGNATOR> ::= <FILE IDENTIFIER>
| <SWITCH FILE IO> (<EXPRESSION>)

The <FILE SPECIFIER> is assumed to be a queue file and the number of messages in the queue will be returned as a fixed number into <ADDRESS GENERATOR>. If <FILE SPECIFIER> is a queue file family, an array of values, one for each family member, will be returned into <ADDRESS GENERATOR>.

MONITOR

See Appendix VIII: SDL MONITORING FACILITY

OVERLAY

<OVERLAY DESIGNATOR> ::= OVERLAY (<EXPRESSION>)

The <EXPRESSION> will be used as an index into the interpreter dictionary by the interpreter swapper. The interpreter dictionary entry will specify the action to be taken. See the B1700 MCP Reference Manual.

READ_CASSETTE

<READ CASSETTE DESIGNATOR>::= READ_CASSETTE (<DESTINATION SPECIFIER>, <HASH_TOTAL SPECIFIER>, <RESULT SPECIFIER>)

<DESTINATION SPECIFIER>::= <ADDRESS GENERATOR>

<HASH_TOTAL SPECIFIER>::= HASH_TOTAL
| NO_HASH_TOTAL

<RESULT SPECIFIER>::= <ADDRESS GENERATOR>

The <READ CASSETTE DESIGNATOR> causes the number of bits specified by the <DESTINATION SPECIFIER> to be read from the console cassette to the address specified by that <DESTINATION SPECIFIER>. This number of bits must be equal to the record size minus the hash-total size (if it is present) of 16 bits. The <HASH_TOTAL SPECIFIER> indicates whether or not a hash-total is expected at the end of the record.

A value of 0 or 1 will be left in the <RESULT SPECIFIER> indicating that the HASH-TOTAL was incorrect or correct, respectively.

READ_FPB, WRITE_FPB

<ACCESS-FPB DESIGNATOR> ::= <ACCESS-FPB IDENTIFIER> (<FILE SPECIFIER>, <SOURCE OR DESTINATION FIELD>)

<ACCESS-FPB IDENTIFIER> ::= READ_FPB | WRITE_FPB

<FILE SPECIFIER> ::= <FILE DESIGNATOR>
| <FILE NUMBER>

<FILE DESIGNATOR> ::= <FILE IDENTIFIER>
| <SWITCH FILE IDENTIFIER> (<EXPRESSION>)

<FILE NUMBER> ::= <EXPRESSION>

<SOURCE OR DESTINATION FIELD> ::= <ADDRESS GENERATOR>

<ADDRESS GENERATOR> ::= See ADDRESS GENERATORS

The File Parameter Block of the file indicated by the <FILE SPECIFIER> is read into, or written from the <SOURCE OR DESTINATION FIELD>.

Note that the <SOURCE OR DESTINATION FIELD> should be 1440 bits in length.

READ_OVERLAY, WRITE_OVERLAY

<ACCESS OVERLAY DESIGNATOR> ::= <ACCESS OVERLAY IDENTIFIER>(<EXPRESSION>)

<ACCESS OVERLAY IDENTIFIER> ::= READ_OVERLAY / WRITE_OVERLAY

The value of the <EXPRESSION> is assumed to be a 76-bit field with the following format from high-order to low-order:

BITS	CONTENTS
0-3	EU = 0 (Not used)
4-27	Base relative beginning address
28-51	Base relative ending address
52-75	Disk address (Relative to user area)

The area described by the beginning and ending addresses is read to, or written from the user disk at the (relative) DISK ADDRESS given.

REFER ADDRESS

<REFER_ADDRESS DESIGNATOR>::= REFER_ADDRESS (<REF VAR>, <EXPRESSION>)

The value of <EXPRESSION> is stored in the address part of <REF VAR>.

REFER LENGTH

<REFER_LENGTH_DESIGNATOR>::= REFER_LENGTH (<REF VAR>, <EXPRESSION>)

The value of <EXPRESSION> is stored in the length part of <REF VAR>.

REFER TYPE

<REFER_TYPE_DESIGNATOR>::= REFER_TYPE (<REF VAR>, <EXPRESSION>)

The value of <EXPRESSION> is stored in the type part of <REF VAR>.

REINSTATE

<REINSTATE DESIGNATOR> ::= REINSTATE (<REINSTATED PROGRAM>)

<REINSTATED PROGRAM> ::= <ADDRESS GENERATOR>

The <REINSTATED PROGRAM> is assumed to describe the field RS_COMMUNICATE_MSG_PTR of RS_NUCLEUS of the program to be reinstated (See description of the RUN STRUCTURE in B1700 MCP Reference Manual).

The reinstating program's M-Machine state is stored in the appropriate parts of its RS_NUCLEUS. The address of the reinstating program's RS_NUCLEUS is stored in the reinstated program's RS_COMMUNICATE_LR.

The program whose RS_COMMUNICATE_MSG_PTR is described by <REINSTATED PROGRAM> is then reinstated.

RESTORE

<RESTORE DESIGNATOR> ::= RESTORE (<ADDRESS GENERATOR LIST>)

<ADDRESS GENERATOR LIST> ::= See ADDRESS GENERATORS

The <RESTORE DESIGNATOR> assigns the current value on the top of the Evaluation Stack to each <ADDRESS GENERATOR>, from right to left, in the list. This operator is used in conjunction with the <SAVE DESIGNATOR>. See above.

EXAMPLE:

```
SAVE (A,B,C);
      .
      .
      .
RESTORE (A,B,C);
```

NOTE THAT RESTORE (A,B,C) IS THE SAME AS:

```
RESTORE (C);
RESTORE (B);
RESTORE (A);
```

REVERSE_STORE

<REVERSE STORE DESIGNATOR> ::= REVERSE_STORE (<ADDRESS GENERATOR LIST>,<EXPRESSION>)

<ADDRESS GENERATOR LIST> ::= See ADDRESS GENERATORS

The REVERSE_STORE OPERATION has the effect of evaluating multiple store operations from left to right instead of from right to left. See THE REPLACE OPERATORS.

For example:

```
REVERSE_STORE (L,M,N,P,X+1);
```

has the same effect as:

```
L := M;
M := N;
N := P;
P := X+1;
```

With the REVERSE_STORE, however, the descriptor for each <ADDRESS GENERATOR> in the list is determined only once.

Note:

```
REVERSE_STORE (L,M,N,P,X+1);
```

is not the same as

```
L:=M:=N:=P:=X+1;
```

SAVE

<SAVE DESIGNATOR> ::= SAVE (<EXPRESSION LIST>)

Each of the <EXPRESSION>s, from left to right, will be evaluated, and the value of each left on the Evaluation Stack (and Value Stack, if necessary). See <RESTORE DESIGNATOR>.

SAVE_STATE

<SAVE STATE DESIGNATOR> ::= SAVE_STATE

The state of the interpreter will be stored in RS.M.MACHINE (See B1700 MCP Reference Manual). Execution will then continue.

SORT

<SORT DESIGNATOR> ::= SORT (<SORT INFORMATION TABLE SPECIFIER>, <SORT KEY TABLE SPECIFIER>, <INPUT FILE DESIGNATOR>, <OUTPUT FILE DESIGNATOR> <TRANSLATE FILE DESIGNATOR>)

<SORT INFORMATION TABLE SPECIFIER> ::= <ADDRESS GENERATOR>

<SORT KEY TABLE SPECIFIER> ::= <ADDRESS GENERATOR>

<INPUT FILE DESIGNATOR ::= <FILE DESIGNATOR>

<TRANSLATE FILE DESIGNATOR> ::= <EMPTY> | . <FILE DESIGNATOR>

<OUTPUT FILE DESIGNATOR> ::= <FILE DESIGNATOR>

<FILE DESIGNATOR>::= <FILE IDENTIFIER>
| <SWITCH FILE IDENTIFIER> (<EXPRESSION>)

The <SORT DESIGNATOR> is a communicate which requests the transfer of records from the input file to the output file according to the SORT key table. The SORT information table includes codes for SORT type, hardware available, and other options.

For formatting specifications of the SORT information table, refer to SORT documentation.

SORT_MERGE

<SORT_MERGE DESIGNATOR> ::= SORT_MERGE
(<SORT INFORMATION TABLE SPECIFIER>,
<SORT KEY TABLE SPECIFIER>,
<INPUT TABLE SPECIFIER>,
<OUTPUT FILE DESIGNATOR>
<TRANSLATE FILE DESIGNATOR>)

<INPUT TABLE SPECIFIER> ::= <ADDRESS GENERATOR>

See SORT STATEMENT for other parameters, and SORT documentation for table formats and semantics.

SORT_SWAP

<SORT_SWAP DESIGNATOR> ::= SORT_SWAP (<RECORD 1>,<RECORD 2>)

<RECORD 1> ::= <ADDRESS GENERATOR>

<RECORD 2> ::= <ADDRESS GENERATOR>

While the <SORT SWAP DESIGNATOR> is intended to be used by the SORT, its application is such that it may be generally useful.

This designator allows the user to swap or exchange two records in memory without allocating a third area for storing one of the records.

Specifically, the record pointed to by <RECORD 1> is exchanged with the record pointed to by <RECORD 2>.

Note: The interpreter being used must contain the SORT_SWAP operator.

THAW_PROGRAM

<THAW-PROGRAM DESIGNATOR>::= THAW_PROGRAM

Execution of this function will allow the program to be rolled out of memory. It will not force it to be rolled out.

THREAD_VECTOR

<THREAD_VECTOR DESIGNATOR> ::= THREAD_VECTOR (<TABLE ADDRESS>,<INDEX>)

<TABLE ADDRESS> ::= <ADDRESS GENERATOR>

<INDEX> ::= <EXPRESSION>

For use by sort only.

The <TABLE ADDRESS> points to the table containing the information described under INITIALIZE_VECTOR. The <INDEX> provides the offset from the beginning of the vector to the next record to be used for comparison.

TRACE

<TRACE DESIGNATOR> ::= TRACE | NOTRACE | TRACE (<EXPRESSION>)

The TRACE will cause the SDL instructions of the normal state program to be traced on the line printer. NOTRACE will turn off the trace. The trace will only be effective when the program is run with an SDL trace interpreter.

TRACE (<EXPRESSION>) provides greater control of the tracing to be done. The low-order 10 bits are used in the following way (numbering of the 10 is from left to right):

Bit	Use
0	Trace all commands except those which modify data or change the program pointer stack. Normal state only.
1	Trace commands which modify data items (e.g., CLR, SNDL, etc.). Normal state only.
2	Trace commands which change the program pointer stack (e.g., IFTH, CASE, EXIT, etc.). Normal state only.
3	Not used.
4-6	Same as 0-2, but for MCP. Several MCP routines (GETSPACE, FORGETSPACE, and others) will not be traced.
7-9	Same as 0-2, but will trace those MCP routines not traced by 4-6.

Note that TRACE(#380#) is the same as TRACE, while TRACE(0) is the same as NOTRACE.

TRANSLATE

<TRANSLATE DESIGNATOR> ::= TRANSLATE (<TRANSLATE

<TRANSLATE TABLE> , <TRANSLATE TABLE ITEM SIZE> , <TRANSLATE RESULT>)

<TRANSLATE SOURCE> ::= <ADDRESS GENERATOR>

<TRANSLATE SOURCE ITEM SIZE> ::= <EXPRESSION>

<TRANSLATE TABLE> ::= <EXPRESSION>

<TRANSLATE TABLE ITEM SIZE> ::= <EXPRESSION>

<TRANSLATE RESULT> ::= <ADDRESS GENERATOR>

<TRANSLATE SOURCE> is assumed to consist of items of size <TRANSLATE SOURCE ITEM SIZE>. Each of the items in <TRANSLATE TABLE> and <TRANSLATE RESULT> are assumed to be of size <TRANSLATE TABLE ITEM SIZE>. Each of the source items is used to subscript into the table to obtain an item which is placed into the result field in the position corresponding to the position of the original item obtained from source. This process continues until the source is exhausted, the result is full, or an error occurs.

If either source or result is not a multiple of its respective item size, then the translation of the last item is undefined.

Both source and table item sizes must be less than or equal to 24. The table must be large enough to accomodate all items in source. If either of these is violated, a run-time error will occur.

APPENDIX I: RESERVED AND SPECIAL WORDS

The following is a list of reserved words in SDL, complete as of May, 1978. These words may only be used as reserved words.

ACCEPT AND AS

BASE BIT BUMP BY

CASE CAT CHANGE CHARACTER CLEAR CLOSE

DECLARE DECREMENT DEFINE DISPLAY DO DUMMY DYNAMIC

ELSE END EQL ENTER_COROUTINE EXIT_COROUTINE EXOR

FILE FILLER FINI FIXED FORMAL FORMAL_VALUE FORWARD FROM

GEQ GTR

IF INTRINSIC

LEQ LOCK LSS

MOD

NEQ NOT

OF ON OR OPEN

PAGED PROCEDURE

READ READ_FILE_HEADER RECORD REDUCE REFER REFERENCE REMAPS

RETURN RETURN_AND_ENABLE_INTERRUPTS

SEARCH_DIRECTORY SEEK SEGMENT SEGMENT_PAGE SKIP SPACE STOP

SUBBIT SUBSTR SWITCH_FILE

THEN TO

UNDO USE

VARYING

WRITE WRITE_FILE_HEADER

ZIP

The following is a list of special words in SDL, complete as of December, 1976. Each special word has a particular meaning, however it may be used as an identifier. In that case, it loses its special significance in SDL.

ACCESS_FILE_INFORMATION

BASE_REGISTER BINARY

CHANGE_STACK_SIZES CHARACTER_FILL CHAR_TABLE COMMUNICATE

COMPILE_CARD_INFO COMMUNICATE_WITH_GISMO CONTROL_STACK_BITS
CONTROL_STACK_TOP CONSOLE_SWITCHES CONV CONVERT

DATA_ADDRESS DATE DC_INITIATE_IO DEBLANK DECIMAL

DELIMITED_TOKEN DESCRIPTOR DISABLE_INTERRUPTS DISPATCH
DISPLAY_BASE DMS_CALL DUMP DUMP_FOR_ANALYSIS DYNAMIC_MEMORY_BASE

ENABLE_INTERRUPTS ERROR_COMMUNICATE EVALUATION_STACK_TOP

EXECUTE

FETCH FETCH_COMMUNICATE_MSG_PTR FETCH_AND_SAVE

FIND_DUPLICATE_CHARACTERS FREEZE_PROGRAM

GROW

HALT HARDWARE_MONITOR HASH_CODE HASH_UNPACK

INITIALIZE_VECTOR INTERROGATE_INTERRUPT_STATUS

LENGTH LIMIT_REGISTER LOCATION

MAKE_DESCRIPTOR MAKE_READ_ONLY MAKE_READ_WRITE MESSAGE_COUNT

M_MEM_SIZE MONITOR_SET MONITOR_RESET MONITOR_CHANGE MONITOR_SET

NAME_OF_DAY NAME_STACK_TOP NDL_OP NEXT_ITEM NEXT_TOKEN NOTRACE

NULL

OVERLAY

PARITY_ADDRESS PREVIOUS_ITEM PROGRAM_SWITCHES

READ_CASSETTE READ_FPB READ_OVERLAY REINSTATE RESTORE

REVERSE_STORE

SAVE SAVE_STATE SEARCH_LINKED_LIST SEARCH_SERIAL_LIST S_MEM_SIZE

SEARCH_SDL_STACKS SORT SORT_DELETE SORT_FILE_FIXUP SORT_MERGE
SORT_RETURN SORT_SEARCH SORT_STEP_DOWN SORT_SWAP SORT_UNBLOCK
SWAP SPO_INPUT_PRESENT

THAW_PROGRAM THREAD_VECTOR TIME TRACE TRANSLATE

VALUE_DESCRIPTOR

WAIT WRITE_FPB WRITE_OVERLAY

X_ADD X_SUB X_MUL X_DIV X_MOD

APPENDIX II: SDL CONTROL CARD OPTIONS

Every SDL control card must have a $ in column one. Columns 73-80 may be used as a sequence field. Note that once an option has been turned on (off), it will remain on (off) until explicitly turned off (on).

```
<CONTROL CARD> ::=              $ <CONTROL STATEMENT>

<CONTROL STATEMENT> ::=         <CONTROL OPTION LIST>
                              | <VOID OPTION>

<CONTROL OPTION LIST> ::=       <CONTROL OPTION>
                              | <CONTROL OPTION>
                                <CONTROL OPTION LIST>

<CONTROL OPTION> ::=            <CONTROL OPTION WORD>
                              | NO <CONTROL OPTION WORD>
                              | <DEBUG OPTION>
                              | <SEQUENCE OPTION>
                              | <PAGE OPTION>
                              | <MERGE OPTION>
                              | <STACK SIZE LIST>
                              | <INTERPRETER OPTION>
                              | <INTRINSIC OPTION>
                              | <RECOMPILE OPTION>
                              | <LIBRARY PACK OPTION>

<CONTROL OPTION WORD> ::=       LIST | LISTALL | SINGLE
                              | SGL | DOUBLE | CODE
                              | CONTROL | NEW | SUPPRESS
                              | XMAP | CHECK | PROFILE | PPROFILE
                              | DETAIL | AMPERSAND | NO_DUPLICATES
                              |  NO_SOURCE | MONITOR
                              | XREF | XREF_ONLY | EXPAND_DEFINES
                              | SIZE | FORMAL_CHECK
                              | TIME_PROCEDURES | TIME_BLOCKS
                              | PASS_END | ERROR_FILE
                              | FREEZE | NEST_PROCEDURE TIMES
                              | ADVISORY | LOCKI
                              | USEDOTS | CONVERTDOTS
                              | TIME_MCP | UNDERSCORES_IN_FILE_NAMES

<DEBUG OPTION> ::=              DEBUG <NUMBER>

<NUMBER> ::=                    <UNSIGNED INTEGER, 8 OR LESS DIGITS>

<SEQUENCE OPTION> ::=           NO SEQ
                              | SEQ <SEQUENCE PARAMETERS>

<SEQUENCE PARAMETERS> ::=       <BASE>
                              | <INCREMENT>
```

```
                                | <BASE> <INCREMENT>

<BASE> ::=                        <NUMBER>

<INCREMENT> ::=                   + <NUMBER>

<PAGE OPTION> ::=                 PAGE

<MERGE OPTION> ::=                MERGE

<STACK SIZE LIST> ::=             <STACK SIZE DESIGNATOR>
                                | <STACK SIZE DESIGNATOR>
                                  <STACK SIZE LIST>

<STACK SIZE
DESIGNATOR> ::=                   <STACK DESIGNATOR> <STACK SIZE>

<STACK DESIGNATOR> ::=            VSSIZE | NSSIZE | ESSIZE
                                | CSSIZE | PPSSIZE | DYNAMICSIZE

<STACK SIZE> ::=                  <NUMBER>

<VOID OPTION> ::=                 VOID <TERMINATING SEQUENCE FIELD>

<TERMINATING SEQUENCE
FIELD> ::=                        <EMPTY>
                                | <EXACTLY 8 CHARACTERS>

<INTERPRETER OPTION> ::=          INTERPRETER <INTERPRETER NAME>

<INTERPRETER NAME> ::=            <EXTERNAL FILE NAME>

<INTRINSIC OPTION> ::=            INTRINSIC
                                  <INTRINSIC FAMILY NAME>

<INTRINSIC FAMILY
NAME> ::=                         <IDENTIFIER> | <CHARACTER STRING>
                                | <FILE FAMILY NAME>

<FILE FAMILY NAME> ::=            <MULTIFILE ID>
                                | <PACK_ID/MULTIFILE ID>

<PACK_ID> ::=                     <CHAR STRING>

<MFID> ::=                        <CHAR STRING>

<LIBRARY PACK OPTION> ::=         LIBRARY_PACK <PACK_ID>

<RECOMPILE OPTION> ::=            CREATE_MASTER
                                | RECOMPILE
```

SEMANTICS IN ALPHABETICAL ORDER:

Note: Default is OFF except where specified as ON.

ADVISORY	Prints advisory messages on the listing. Default is ON.
AMPERSAND	Prints those ampersand cards which are examined. Default is ON.
CHECK	The merged source will be checked for sequence errors. Default is ON. Sequence checking is done after any resequencing due to a $SEQ is complete.
CODE	Prints generated code.
CONTROL	Prints control cards.
CONVERTDOTS	Converts dots "." to underscores "_" when used as separators in identifiers. The conversion will be reflected in all compiler output including the listing and NEWSOURCE files. RECORD constructs may not be used with dot separators in identifiers.
CREATE_MASTER	See Appendix VII_
CSSIZE	Control Stack size.
DEBUG	Compiler debug use only.
DETAIL	Prints expansion of define invocations.
DOUBLE	Double spaces listing when printing.
DYNAMICSIZE	Amount of memory used for paged array pages.
ERROR_FILE	A separate error file will be produced containing only errors and warnings and the source images to which they apply.
ESSIZE	Evaluation Stack size.
EXPAND_DEFINES	Causes define expansions to be cross-referenced (used in conjunction with XREF or XREF_ONLY).
FREEZE	The FREEZE bit will be set in the program's FBP, preventing the program from being rolled out during execution.

FORMAL.CHECK	Procedure actual parameters and values returned from typed procedures will be checked respectively against their corresponding formal parameters and procedure formal types.
INTERPRETER	Changes the interpreter name.
INTRINSIC	Changes the family names of intrinsics to be used.
LIBRARY_PACK	Assumes all library files are on the pack specified.
LIST	Lists the source input which was compiled. NO LIST will also turn off LISTALL. Default is ON.
LISTALL	Lists all SDL source input (whether or not conditionally excluded). LISTALL turns on list, but NO LISTALL will not turn off list.
LOCKI	Intermediate work files will be locked into the disk directory as they are created. (See Appendix IV: RUNNING THE COMPILER).
MERGE	The primary source file is on tape or disk which will have the cards, from the card reader, merged with it.
MONITOR	See Appendix VIII: SDL MONITOR FACILITY
NEST_PROCE-DURE_TIMES	See Appendix III.
NEW	Creates a new source file.
NO	NO preceding an option (which allows it) will turn that option off.
NO_DUPLICATES	Newly declared identifier will not be checked for uniqueness. The programmer must guarantee that there are no duplicates before using this option. It will reduce compile time for large programs only.
NO_SOURCE	Program source images will not be saved, thereby shortening the compiler work file. No source listing will be possible when this option is specified. This should be used with long programs only.

NSSIZE — Name Stack size.

PAGE — Page eject if listing.

PASS_END — The total elapsed time and the number of errors will be printed at the end of each pass.

PPSSIZE — Program Pointer Stack size.

RECOMPILE — See Appendix VII.

RECOMPILE_TIMES — The start and stop times of each of the phases of the "bind" pass of a CREATE_MASTER or RECOMPILE will be printed on the listing.

SEQ — Resequences new source file usirg base and increment specified. Default increment is 1000, default base is the sequence number of the $SEQ card. If the $SEQ card has no seq number the default base is 1000.

SINGLE (SGL) — Single spaces listing when printing. Default is ON.

SIZE — Prints segment sizes by name at end of compile.

SUPPRESS — Suppresses warning messages. To suppress sequence error messages, turn off CHECK.

TIME_BLOCKS

TIME_PROCEDURES

TIME_MCP — See Appendix III.

UNDERSCORES_IN_FILE_NAMES

Provides capability to convert internal file names with dots (.) as separators. This option should be used with the CONVERTDOTS option.

USEDOTS — Allows the use of dots, ".", as separators in identifiers. Otherwise, underscores, "_" will be required (See CONVERTDOTS).

VOID — The VOID option will void records in the primary file which have sequence fields less than or equal to the <TERMINATING SEQUENCE FIELD>. If the field is omitted, only the record with the sequence number corresponding to the VOID card sequence number will be deleted. The VOID option will not delete images in a secondary (card) source file.

VSSIZE — Value Stack size.

WORKING_SET_BYTES

Specifies the working set size of the object program as used by MCPI. This option has no effect on programs to be run under MCPII.

XMAP — Creates an extended code map file for post compilation analysis. The name of the file passed to SDL/XMAP is "XMAPMMDDYY/<TIME>", where MM is the month, DD is the day of the month, YY is the year, and <TIME> is the time of day of the compile.

XREF — Produces a cross-reference listing of the program. The name of the file passed to SDL/XREF is "XREFMMDDYY/<TIME>", where MM is the month, DD is the day of the month, YY is the year, and <TIME> is the time of day of the compile.

XREF_ONLY — Produces a cross-reference listing and then terminates the compilation. The name of the file passed to SDL/XREF_ONLY is "XREFMMDDYY/<TIME>", where MM is the month, DD is the day of the month, YY is the year, and <TIME> is the time of day of the compile.

Note: All control cards may use & in column 1 in place of $. Those control cards with & in column 1 will be permanently placed in a new source file whenever one is made. They may also be conditionally included or excluded during compilation.

EX SDL/XREF FILE PFILENAME XREF.....
FILE PRINT NAME outputfilename

EX SDL/XMAP FILE MAPFILE NAME XMAP .....
FILE PRINTER NAME outputfilename

APPENDIX III: PROGRAMMING OPTIMIZATION

The following control card options can be useful to the programmer who wishes to determine the most time consuming part(s) of his program. The purpose of these control options is to point out the parts of the program which are the most time consuming and/or heavily used.

PROFILE

PPROFILE — Establishes a dynamic array, each element of which is a counter for one procedure. The index number for each procedure appears in the listing following the <PROCEDURE IDENTIFIER>. The value of the counter will reflect the number of entrances to the procedure in question. Those with the highest counters should be investigated with the PROFILE option.

PROFILE — Establishes a dynamic array, each element of which is a counter for one branching operation (<DO GROUP>, <IF STATEMENT>, or <CASE STATEMENT>). The index into the array will appear in the listing following the statement in question. Those branches with the highest counter values are the branches most heavily used.

HARDWARE MONITOR

<HARDWARE MONITOR DESIGNATOR> ::= HARDWARE_MONITOR (<EXPRESSION>)

The B1700 is equipped with a hardware monitor which may be manually wired to suit the needs of the programmer. The device can be useful as a timer or a counter to monitor program efficiency.

The low-order 8 bits of the <EXPRESSION> is used as the low-order 8 bits of the M-instruction monitor. For wiring instructions of the hardware device see Computer Performance Monitor II: System Summary Manual.

PROGRAM TIMING

A high-resolution timer and the means to access it are available on select B1720-series systems. This timer is accessed directly by the interpreter, bypassing the MCP and its inherent effects on timing accuracy.

Timing of procedures and/or blocks is initiated by the use of control options: $TIME_PROCEDURES and $TIME_BLOCKS. The appearance of either of these options turns it on; the appearance of the option preceded by NO turns it off. The setting of the option at the time of parsing of the procedure head or of the block head (DO and DO FOREVER, in the case of DO groups) determines whether or not the attendant body of code is to be timed.

For each item to be timed, a timer cell number is assigned. Upon entrance to the body of code, the timer value is subtracted from the proper cell and upon exit, the timer value is added to the cell. Procedures are not timed around calls of other procedures, so that procedure times reflect only the elapsed time spent within that procedure. Block timing works the same way, i.e., times of nested blocks are added to those of enclosing blocks, but times of procedures which are called are not included in the times of the calling procedure or blocks. The times of called procedures WILL be added to those of the caller by specifying the option NEST_PROCEDURE_TIMES.

At the time of execution, an intrinsic will be invoked which will print the timing cells ordered by value. The contents of these cells are the number of microseconds spent in the timed bodies of code. If the job terminates abnormally, then DUMP/ANALYZER will print the contents of the timing cells.

It is intended that the timing functions will be used in the following manner: First, all the procedures in a program will be timed. Upon isolation of the "hot" procedures, block timings will be requested for those blocks contained in these procedures. If both block and procedure timings are requested for large programs, an inordinate amount of memory will be allocated for the timing cells, which are 48 bits in length.

This scheme is usable by the MCP. The $-option $TIME_MCP must be included at compile time. The timing cells are printed with a SPO message.

APPENDIX IV: RUNNING THE COMPILER

SYSTEM CONTROL CARDS FOR B1700

There are two basic deck setup formats. They are:

4. The primary source file is on cards.

```
      <SYSTEM COMPILE CARD>
*     <FILE EQUATE CARD FOR FILE NEWSOURCE>
      DATA CARDS
* $ NEW
      <SDL PROGRAM>
      FINI
      END
```

* If the primary source file is to be saved on tape or disk, these cards must be included.

8. The primary source file is on disk.

```
      <SYSTEM COMPILE CARD>
      <FILE EQUATE CARD FOR FILE SOURCE>
*     <FILE EQUATE CARD FOR FILE NEWSOURCE>
      DATA CARDS
  $ MERGE
* $ NEW
      <PATCHES TO SDL PROGRAM>
      END
```

* If the merged file is to be saved, these cards must be included.

Note: Refer to the B1700 MCP Software Operational Guide for the exact format of the compile and file equate cards.

SDL FILE NAMES

CARDS	Card input file (80 or 90 byte records)
SOURCE	Primary source file if $ MERGE is used (80 or 90 byte records)
NEWSOURCE	Updated source file if $ NEW is used (90 byte records)

LINE	Line printer file
ERROR.LINE	Separate error file (produced when $ERROR.FILE is used)
XREF.LINE	Lists file for XREF. Allows file equation in the compiler.
XMAP.LINE	Lists file for XMAP. Allows file equation in the compiler

SDL WORKFILE NAMES

PFILE	Intermediate file produced by the pre-pass.
IFILE	Intermediate file produced by the first pass.
IMAGE.FILE	Source image file produced by the pre-pass.

SPO INPUT TO COMPILER

The compiler will notice if the operator gives it SPO input during any of the first three passes (SDLP, SDL1, SDL2). SPO input will be ignored during SDL3, the partial recompilation binder. The operator may give any of the following commands in the AX message:

STATUS	The compiler will display the current pass executing, sequence number being compiled, and errors detected so far.
LIST	The compiler will begin listing in whatever pass is currently executing.
NO LIST	Stops listing in whatever pass is currently executing.
PASS_END	Sets option to display a message as each pass completes.
NO PASS_END	resets PASS_END option.
LOCKI	The compiler will lock intermediate files as they are created and will lock any that have already been created but not released. The

intermediate files may then be used to restart the compiler if necessary (see below) or be analyzed with SDL/IA (not released outside the company).

NO LOCKI — Intermediate files not already locked will not be locked.

SDL RESTART

If intermediate files have been saved (see LOCKI above) and a compile is terminated in SDL1, SDL2, or SDL3 due to machine failures, it may be restarted in SDL1 or SDL2 to avoid repeating the entire compile. Program switch zero is normally set to zero indicating a full compile. It may be set on the compile card, however, to one (indicating an SDL1 restart) or two (indicating an SDL2 restart). SDL3 cannot be restarted; instead the operator must restart SDL2.

The compiler will expect the following files when restarted:

SDL1	PFILE
	IMAGE.FILE
	MASTER/INF (if CREATE_MASTER compile)
SDL2	IFILE
	IMAGE.FILE
	MASTER/INF (if CREATE_MASTER compile)

Files will have been saved under these names if (a) the operator entered a LOCKI message or (b) $LOCKI appeared on a compiler control card.

APPENDIX V: CONDITIONAL COMPILATION

The conditional compilation facility allows the user to selectively compile blocks of code without the necessity of physically adding or removing records.

<CONDITIONAL INCLUSION> records are always written to a new file (if one is created), whether or not they are compiled. If conditional compilation records are to be printed with the source listing, then LISTALL must appear on the $-card. If not specified, only those conditional compilation records which were compiled are printed.

The BNF for the conditional compilation is as follows:

```
<CONDITIONAL INCLUSION> ::=  <SET STATEMENT>
                           | <RESET STATEMENT>
                           | <PAGE STATEMENT>
                           | <LIBRARY STATEMENT>
                           | <IF BLOCK>

<SET STATEMENT> ::=          SET <SET SYMBOL LIST>

<SET SYMBOL LIST> ::=        <SET SYMBOL>
                           | <SET SYMBOL LIST>
                             <SET SYMBOL>

<SET SYMBOL> ::=             <BOOLEAN SYMBOL>

<BOOLEAN SYMBOL> ::=         <LETTER>
                           | <BOOLEAN SYMBOL> <LETTER>
                           | <BOOLEAN SYMBOL> <DIGIT>

<RESET STATEMENT> ::=        RESET <RESET SYMBOL LIST>

<RESET SYMBOL LIST> ::=      <RESET SYMBOL>
                           | <RESET SYMBOL LIST>
                             <RESET SYMBOL>

<RESET SYMBOL> ::=           <BOOLEAN SYMBOL>

<PAGE STATEMENT> ::=         PAGE

<LIBRARY STATEMENT> ::=      LIBRARY <FILE NAME>

<FILE NAME>::=               <MULTI-FILE IDENTIFIER>
                           | <MULTI-FILE IDENTIFIER> /
                             <FILE IDENTIFIER>
                           | <PACK IDENTIFIER> /
                             <MULTI-FILE IDENTIFIER> /
```

```
                              | <PACK IDENTIFIER> /
                                <MULTI-FILE IDENTIFIER> /
                                <FILE IDENTIFIER>

<PACK IDENTIFIER>::=            <IDENTIFIER>

<MULTI-FILE IDENTIFIER>::=      <IDENTIFIER>

<FILE IDENTIFIER>::=            <IDENTIFIER>

<IF BLOCK> ::=                  <IF STATEMENT>
                                <INCLUSION BLOCK>
                                <END STATEMENT>
                              | <IF STATEMENT>
                                <TRUE PART>
                                <INCLUSION BLOCK>
                                <END STATEMENT>

<IF STATEMENT> ::=              IF <BOOLEAN EXPRESSION>

<BOOLEAN EXPRESSION> ::=        <BOOLEAN FACTOR>
                              | <BOOLEAN EXPRESSION> OR
                                <BOOLEAN FACTOR>

<BOOLEAN FACTOR> ::=            <BOOLEAN SECONDARY>
                              | <BOOLEAN FACTOR> AND
                                <BOOLEAN SECONDARY>

<BOOLEAN SECONDARY> ::=         <BOOLEAN PRIMARY>
                              | NOT <BOOLEAN PRIMARY>

<BOOLEAN PRIMARY> ::=           <SET SYMBOL>
                              | <RESET SYMBOL>

<INCLUSION BLOCK> ::=           <SDL SOURCE IMAGE BLOCK>
                              | <IF BLOCK>

<SDL SOURCE
IMAGE BLOCK> ::=                <EMPTY>
                              | <1 OR MORE SDL SOURCE IMAGES>

<END STATEMENT> ::=             END

<TRUE PART> ::=                 <INCLUSION BLOCK> <ELSE STATEMENT>

<ELSE STATEMENT> ::=            ELSE
```

All records containing conditional compilation statements must have an ampersand (&) in column 1 (except the <SDL SOURCE IMAGE BLOCK>). In addition, a complete conditional inclusion statement must be contained on one &-CARD. Columns 2-72 are free-field, and columns 73-80 may contain sequence numbers.

Note that <BOOLEAN EXPRESSION>s may contain the logical operators (from lowest precedence to highest): OR, AND, and NOT.

The <PAGE STATEMENT> will cause a page eject if the source file is being listed. The <LIBRARY STATEMENT> will cause the images from the file specified by <FILE NAME> to be included in the source program.

As an example, consider the following SDL source statements illustrating nested conditional compilation statements and <SDL SOURCE IMAGE BLOCK>s.

```
COL 1                  FREE-FIELD: COLS 2-72          SEQ: 73-80

& SET A B C                                                0100
& RESET D E                                                0200
DECLARE (A,B) FIXED;                                       0300
& IF A AND E                                               0400
A := B;                                                    0500
& ELSE                                                     0600
A := X CAT Y+Z; % WHOLE SOURCE IMAGE IS INCLUDED           0700
& IF C                                                     0800
B := A;                                                    0900
& END                                                      1000
& END                                                      1100
& IF B OR D                                                1200
BUMP B;                                                    1300
& ELSE                                                     1400
BUMP A;                                                    1500
& END                                                      1600
```

The compilation of the following statements would result.

```
DECLARE (A,B) FIXED;                                       0300
A := X CAT Y+Z; % WHOLE SOURCE IMAGE IS INCLUDED           0700
B := A;                                                    0900
BUMP B;                                                    1300
```

Note that every IF must be paired with either an ELSE or an END. Every ELSE must have an END associated with it.

APPENDIX VI: SDL PROGRAMMING TECHNIQUES

This section contains coding suggestions and examples which result in decreased source code and/or object code.

DECLARATIONS:

1. As many non-structured declarations as possible (up to a maximum of 32) should be declared in one <declare STATEMENT>. Example:

```
DECLARE A FIXED, (B,C) BIT(24);
```

generates more efficient code than:

```
DECLARE A FIXED;
DECLARE (B,C) BIT(24);
```

2. A <DEFINE ACTUAL PARAMETER> (See DEFINE INVOCATION) may be a series of SDL statements. For example:

```
DEFINE COMPARE(TS,S) AS#
     IF TOKEN_SYMBOL=TS
          THEN DO;
                 S;
                 UNDO THIS_ONE;
               END#;
```

may be invoked as:

```
DO THIS_ONE FOREVER;
     COMPARE ("SINGLE", SINGLE_SPACE := TRUE);
     COMPARE ("MERGE", IF LASTUSED + 0
          THEN UNDO THIS_ONE;
          LASTUSED := 2;
          OPEN SOURCE INPUT;
          READ SOURCE (TAPEWORK));
     COMPARE (...,...);
     .
     .
     .
END THIS_ONE;
```

PROCEDURES:

1. Procedures from highest efficiency to lowest are:

PARAMETERS	LOCAL DATA
NO	NO
NO	YES
YES	NO
YES	YES

STATEMENTS:

1. When the value returned by a typed procedure is to be ignored:

```
IF P(X-Y) THEN;
```

is more efficient than:

```
TEMP := P(X-Y);
```

2. Use "%" at the beginning of a comment rather than "/*...*/" as delimiters. The "%" stops the scanning of that record. If the "/*...*/" form is used, scanning must continue to detect the ending termirator. Thus compile time is increased.

3. The expression:

```
SUBSTR("0123456789ABCDEF",N,1)
```

generates much less code than

```
CASE N OF ("0","1","2",..."E","F")
```

4. The fact that a boolean expression evaluates to a one or zero can often be used to advantage. For example, the statement:

```
X := A>0;
```

is more efficient than

```
X := IF A>0 THEN 1 ELSE 0;
```

and the results are the same.

5. BUMP A := B; stores B into A and bumps B, and BUMP A ::= B; stores B into A and bumps A.

6. REVERSE_STORE (IF <CONDITION> THEN A ELSE B, C;) selectively stores C into A or B.

7. Consider the following:

In a compiler, for example, assume that all calls on the error routine follow a THEN/ELSE or are in a <CASE STATEMENT>. Example:

```
1.  IF <CONDITION> THEN ERROR(E005);
2.  CASE N;
                .;
                .;
                .;
                ERROR(E137);
                .;
            END CASE;
```

It is sometimes desirable to put these calls into a separate segment, especially when E005 and E137 represent character strings (i.e., in-line ERROR MESSAGEs).

For example:

```
DEFINE ERROR(N) AS #SEGMENT (ERROR_CALLS);
ERROR_ROUTINE (N)#;
```

Because of the temporary nature of segmenting subordinate executable statements, only the calls will be in separate segments.

8. When two or more elements of a <CASE STATEMENT> or an <IF STATEMENT> have identical code, more efficient code is generated if the code is put into a separate procedure (with no parameters or data). In both cases, execution time will be identical, but object code savings could be substantial.

9. Use conditional compilation statements to remove debugging code, rather than physically removing the code. See Appendix VII.

APPENDIX VII: SDL PARTIAL RECOMPILATION FACILITY

The SDL compiler includes a facility whereby it is possible to save information from one compilation which will enable the compiler to recompile only one (or more) Lexic Level Zero procedures in subsequent runs, thus reducing computer time for the recompilations.

A. SAVING THE MASTER COMPILER INFORMATION

The master compile information is saved by the compiler in the following five files:

Internal Name	Default External Name
NEWSOURCE	"NEW"/"SOURCE"
NEW_INFO_FILE	"NEW"/"INF"
NEW_SECONDARY_FILE	"NEW"/"SEC"
NEW_BLOCK_ADDRESS_FILE	"NEW"/"BAF"
NEW_FPB_FILE	"NEW"/"FPB"

Note that the file NEWSOURCE is identical to, and created in the same way as, the file created with the $NEW card. All five files will be created with the compiler $-option (Ncte: Brackets here indicate optional specifications):

$CREATE_MASTER [[<PACK_ID>/]<MULTIFILE_ID>]

If specified, <MULTIFILE_ID> will be used instead of the default multifile id, "NEW", for all the files. If also specified, <PACK_ID> will direct all the files to the named user disk pack or cartridge instead of system disk. <PACK_ID> and <MULTIFILE_ID> must be quoted character literals.

Notes:

1. The CREATE_MASTER option must be on the first card in the compile deck (file "CARDS"), and that card may contain no other dollar options (except RECOMPILE--See the following section).

2. The new source file must be completely sequenced, so $SEQ should be used to assure this if necessary. This includes all $-CARDS, as they will be included in the new source file.

3. $NEW option has no effect in conjunction with CREATE_MASTER.

8. PARTIAL RECOMPILATION

By supplying the information saved during a CREATE_MASTER compile, one may have only those Lexic Level Zero procedures recompiled which have actually been patched. The patch deck is perfectly ordinary except that no patch cards may change Lexic Level Zero code, declarations or procedure heads.

Partial recompilation will be invoked with the $-option (Note: Brackets here indicate optional specifications):

$RECOMPILE [[<PACK_ID>/]<MULTIFILE_ID]

The compiler will then expect the following six files as input:

Internal Name	Default External Name
SOURCE	"MASTER"/"SOURCE"
MASTER_INFO_FILE	"MASTER"/"INF"
MASTER_SECONDARY_FILE	"MASTER"/"SEC"
MASTER_BLOCK_ADDRESS_FILE	"MASTER"/"BAF"
MASTER_FPB_FILE	"MASTER"/"FPB"
MASTER_MPT_FILE	"MASTER"/"MPT"

If specified in the RECOMPILE option, <MULTIFILE_ID> will be used instead of the default id "MASTER". If also specified, the files will be expected to be found on user pack or cartridge <PACK_ID>. <PACK_ID> and <MULTIFILE_ID> must be quoted character literals.

Notes:

1. The RECOMPILE option must be on the first card in the compile deck (file "CARDS") and that card may contain no other dollar options (except CREATE_MASTER, see previous section).

2. The patch deck may contain $-CARDs and &SET and &RESET cards followed by patch cards. If &-CARDs are used, however, they will only apply to procedures being recompiled and may, therefore, cause unwanted effects.

3. Neither $SEQ nor $MERGE may be used with $RECOMPILE.

C. SIMULTANEOUS RECOMPILE AND CREATE_MASTER

New master information may be saved from a recompilation run with very little overhead. Both RECOMPILE and CREATE_MASTER options (See above.) must be on the first card of the compile deck. All restrictions noted in A and B should be observed.

D. GENERAL CONSIDERATIONS

1. All input and output files must be on disk. (This does not apply to the SOURCE file for a straight CREATE_MASTER which is read in the normal way as the result of a $MERGE card. It does apply to SOURCE when doing RECOMPILE.)

2. File equation cards for recompilation files will be ignored unless no <PACK_ID> or <MULTIFILE_ID> has been specified on the $-CARD.

3. During recompilation the only source which can be listed is that which is actually being recompiled.

4. $-CARDs for timing, monitoring, and PROFILE may be added during recompilation. They will only affect those procedures being recompiled, however, even if they are at the beginning of the patch deck.

5. A CREATE_MASTER compilation reporting syntax errors which are strictly local to lexic level zero procedures will produce usable master files. These may then be used to recompile the offending procedures. Since the CREATE_MASTER produced no object file, however, some of the $-Card information will be missing for the recompilation--specifically stack size cards. These must be included in the recompile deck.

6. $XMAP is incompatible with partial recompilation and may not be specified if CREATE_MASTER or RECOMPILE have been invoked.

E. EXAMPLES

1. CREATE_MASTER compilation

```
?COMPILE MYPROG WITH SDL TO LIBRARY
?FILE SOURCE NAME MYPROG/OLDSOURCE TAPE;
?DATA CARDS
$CREATE_MASTER "MYPROG"
$MERGE SEQ LIST
[Patch Cards]
?END

?DUMP TO MYTAPE/RECOMP  MYPROG/=;
```

2. Partial recompilation (from user pack)

```
?LOAD TO MYPACK FROM MYTAPE/RECOMP MYPROG/=;

?COMPILE MYNEWPROG WITH SDL TO LIBRARY
?DATA CARDS
$RECOMPILE "MYPACK"/"MYPROG"
$LIST
[Patch Cards]
?END
```

3. Simultaneous operations

```
?LOAD FROM MYTAPE/RECOMP MYPROG/=;

?COMPILE MYNEWPROG WITH SDL TO LIBRARY
?DATA CARDS
$RECOMPILE "MYPROG" CREATE_MASTER "MYPROG"
[Patch Cards]
?END

?DUMP TO MYNEWTAPE/RECOMP MYNEWPROG/=;
```

APPENDIX VIII: SDL MONITORING FACILITY

Procedure entry and exit can be dynamically monitored via features that are available through the SDL compiler. Use of the monitoring feature proceeds in two steps. First, at compilation time, the user specifies via control cards that various procedures are to be "candidates for monitoring" in subsequent executions of the program. Then at execution time the user specifies via a RUN-TIME MONITOR STATEMENT that some subset of the candidate procedures are to be monitored for this run. The RUN-TIME MONITOR STATEMENT can be input through the SPO, or from some user file, at program BOJ or during the execution of the program via execution of built-in functions.

OUTPUT FORMATS

Assume a procedure named PROC is being monitored and that it has two parameters X and Y. An invocation of PROC would produce the following monitor information:

```
------k blanks ------[k]PROC cccccccc-->>ddddddddd
------k+1 blanks------Y= the value of Y at the point of invo-
cation as an SDL literal
------k+1 blanks------X= the value of X at the point of invo-
cation as an SDL literal
```

Here k describes the nesting level of the call, cccccccc is the sequence number of the invocation point, and ddddddddd is the sequence number of the procedure head of PROC.

When PROC is exited, the following line is emitted:

```
------k blanks------[k] exit PROC at eeeeeeee
```

If PROC is a function, the following line will also be emitted:

```
------k+1 blanks------PROC= the value of PROC specified as an
SDL literal
```

The output data may be directed to any file. This is done by associating the file attribute MONITOR_OUTPUT_FILE with some file in the program. The following restrictions hold.

MONITOR_OUTPUT_FILE RESTRICTIONS

1. The feature is not dynamic. (It cannot be changed with a CHANGE statement).

2. The length of a record in the output file should be more than 71 characters.

3. If several files are given the MONITOR_OUTPUT_FILE attribute, the last file so declared becomes the monitor output file.

4. If any procedures are declared to be candidates for monitor then a monitor output file should be declared. If it is not, the compiler will append a file to the program for this purpose.

5. The file must be sequential with fixed length records.

6. The user should never issue an explicit open on the file.

If the value of a parameter or a procedure is being written and current output record is insufficient in length, the literal will be continued to the next record for as many records as is necessary. Indentation is not peformed on subsequent lines. Indentation of the first line ceases within 60 spaces of the end of the monitor output record. Values of length zero are noted appropriately regardless of type. If a character value contains unprintable data, the value will be printed as three asterisks followed by a hex representation of the data. Only the first 30 characters of any procedure name and the first 10 characters of any formal name are used.

MONITORING: SPECIFYING PROCEDURES

The user specifies that procedures are candidates for monitoring with the dollar card options MONITOR and MONITOR_OFF. The qualifier NO is meaningful in front of both words. The discussion of MONITOR_OFF will be deferred to a later section. However, for the purposes of qualification, the two options are semantically equivalent. Specifically, if MONITOR is ON when the procedure name first appears (either in its forward or its head), then the procedure becomes a candidate for monitoring. Note that the MONITOR option relates to procedures and not to procedure invocations. There is no way to specify the concept that a procedure is a candidate for monitoring but that some particular invocation of that procedure is not to be monitored. Also note that it is the state of the option when the FORWARD (if present)

is encountered that is important.

The concept of a RUN-TIME MONITORING statement was previously introduced. This statement will be read into the program at BOJ from any file that the user specifies. This is done by giving the attribute MONITOR_INPUT_FILE to some file declared in the program. The following restrictions hold:

Restrictions:

1. 1,2,3,5, and 6 under MONITOR_OUTPUT_FILE RESTRICTIONS.

2. If no file is declared with the attributes MONITOR_INPUT_ FILE and procedures are declared to be candidates for monitoring then the program issues accepts at the beginning of job to obtain the necessary information from the SPO.

3. If a file is declared to be the MONITOR_INPUT_FILE then the monitoring information must be the first record(s) of the file.

RUN-TIME MONITOR STATEMENT

The RUN-TIME MONITOR statement consists of a run-time monitor expression that is terminated by a semicolon. Formal specification of the RUN-TIME MONITOR expression syntax is deferred to a later section. The following examples will (hopefully) illustrate the salient features of the statement. Here please read "all procedures" as "all procedures which are candidates for monitoring".

EXAMPLE	MEANING
1. $ALL;	Monitor all procedures
2. $NONE;	Monitor no procedures
3. X1;	Monitor all procedures whose name is X1.

4.
```
***          ***
* X1 X2;      *
*              *
* X1,X2;      *
*              *
* X1 OR X2;   *
*              *
* X1 + X2;    *
***          ***
```
(All four statements are equivalent).

Monitor all procedures named X1 or or X2.

5.
```
***          ***
* NOT X1;    *
*              *
*  -X1;      *
***          ***
```
(Both statements are equivalent).
Monitor all procedures whose name is not X1.

6.
```
00000000-01999999;
```
Monitor all procedures whose forwards or procedure heads occurred on or between the two sequence numbers.

7.
```
***                    ***
* 00000000-01999999    *
*                       *
* AND NOT SCAN;         *
*                       *
* 00000000-01999999    *
*                       *
* * - SCAN              *
*                       *
***                    ***
```
Same as (6.) above except that procedures name SCAN are not to be monitored.

8.
```
***                    ***
* 00000000-01999999    *
*                       *
*  * SCAN;              *
***                    ***
```
Monitor all procedures named SCAN in the range described.

9.
```
***                    ***
* 01426000-01579000    *
*        or             *
* 02748300-99999999    *
*        or             *
* SCAN;                 *
***                    ***
```
Monitor all procedures in the two ranges specified plus any procedure named SCAN which is out of these ranges.

MONITORING: PROGRAMMATIC CONTROL

The $MONITOR_OFF option and the three specials MONITOR_SET, MONITOR_RESET, and MONITOR_CHANGE are added to SDL to allow program control of monitoring. If the $MONITOR_OFF option was ever on, the program will not require a RUN-TIME MONITOR statement at BOJ and will behave as if the RUN-TIME MONITOR statement "$NONE;" had been read.

Each of the three specials is an unvalued procedure with one argument, a RUN-TIME MONITOR statement expressed as an expression which generates a character string, e.g., MONITOR_SET ("X1,X2;");. MONITOR_RESET causes monitoring to be discontinued for all procedures satisfying its argument. If a procedure is not currently being monitored but still satisfies MONITOR_RESET's argument, it will continue not to be monitored.

MONITOR_SET causes monitoring to be commenced on all procedures satisfying its argument. If a procedure is satisfied by MONITOR_SET's argument and is currently being monitored, it continues to be monitored. If a procedure is currently being monitored and does not satisfy MONITOR_SET's argument, it continues to be monitored.

After the execution of a MONITOR_CHANGE only those procedures referenced by its argument will be monitored.

There are no problems of symmetry on calls and returns; i.e., one can begin monitoring a procedure that has already been entered or discontinue the monitoring of some procedure that has currently been entered. The only loss is that the monitor output information is "thrown out of sync" in terms of the nesting level for a while.

SYNTAX OF A RUN-TIME MONITORING STATEMENT

```
<STATEMENT> ::=  <EXPRESSION>;                          1
                 |$ALL;                                 2
                 |$NONE;                                3
<EXPRESSION> ::= <TERM>                                 4
                 |<TERM> <OR> <EXPRESSION>              5
<TERM> ::= <FACTOR>                                     6
```

```
            |<FACTOR> <AND> <TERM>                                    7

<FACTOR> ::= <PRIME>                                                  8

              |<NOT> <PRIME>                                          9

<PRIME> ::= (<EXPRESSION>)                                            10

             |<RANGE>                                                 11

             |<LIST>                                                  12

<RANGE> ::= <8 DIGIT SEQ #>-<8 DIGIT SEQ #>                           13

<LIST ::= <SDL_IDENTIFIER>                                            14

           |<SDL_IDENTIFIER>, <LIST>                                   15

           |<SDL_IDENTIFIER><LIST>                                     16

<OR> ::=  OR                                                          17

           |+                                                         18

<AND> ::=  AND                                                        19

            |*                                                        20

<NOT> ::=  |NOT                                                       21

            |-                                                        22
```

NOTES

1. The <8 DIGIT SEQ #>s referred to in line 13 must be such that the first is less than or equal to the second.

2. The <SDL_IDENTIFIER>s referred to in (14-16) are names of procedures in the program. Only the first 30 characters are used.

THE MONITOR FILE

When monitoring a program, the monitoring intrinsics reference a random access file associated with the compilation of the program. The name of the code file and the name of its monitor file are given below:

CODE FILE	MONITOR FILE
A	A/$$A
A/B	A/$$B
A/B/C	A/B/$$C

INDEX

ACCEPT STATEMENT 9-12
ACCESS_FILE_INFORMATION 10-37
ADDRESS AND VALUE PARAMETERS 8-15
ADDRESS GENERATING FUNCTIONS 8-8
ADDRESS GENERATORS 8-13
ADDRESS MODIFIER 8-12
ADDRESS VARIABLES 8-5
ADVISORY 12-3
ALL_AREAS_AT_OPEN 5-28, 10-19
AMPERSAND OPTION 12-3
APPENDIX I: RESERVED AND SPECIAL WORDS 11-1
APPENDIX II: SDL CONTROL CARD OPTIONS 12-1
APPENDIX III: PROGRAMMING OPTIMIZATION 13-1
APPENDIX IV: RUNNING THE COMPILER 14-1
APPENDIX V: CONDITIONAL COMPILATION 15-1
APPENDIX VI: SDL PROGRAMMING TECHNIQUES 16-1
APPENDIX VII: SDL PARTIAL RECOMPILATION FACILITY 17-1
APPENDIX VIII: SDL MONITORING FACILITY 18-1
AREA_BY_CYLINDER 5-28, 10-19
ARITHMETIC OPERATORS 7-5
ARRAY 5-2
ARRAY STRUCTURE 5-13
ASSIGNMENT STATEMENT 7-8
ASSIGNMENT STATEMENTS AND EXPRESSIONS 7-1
ASSIGNOR 8-4

BACKUS NAUR FORM 1-1
BASE_REGISTER 8-17
BASIC COMPONENTS OF THE SDL LANGUAGE 2-1
BINARY CONVERSION 8-18
BINARY SEARCH 8-18
BIT STRINGS 2-3
BUFFERS 5-24, 10-22
BUMP 8-2, 10-12

CALLING ABILITY 3-5
CASE EXPRESSION 8-2
CASE STATEMENT 10-7
CHANGE STATEMENT (FILE ATTRIBUTE STATEMENT) 10-14
CHANGE_STACK_SIZES 10-37
CHAR_TABLE 2-5
CHARACTER STRINGS 2-4
CHARACTER_FILL 10-38
CHECK OPTION 12-3
CLEAR STATEMENT 10-12
CLOSE STATEMENT 9-4
CODE OPTION 12-3

COMMENTS 2-2
COMMUNICATE 10-38
COMMUNICATE_WITH_GISMO 8-19
COMPILE_CARD_INFO 10-38
CONCATENATION 7-11
CONDITIONAL COMPILATION 15-1
CONDITIONAL EXPRESSION 8-1
CONSOLE_SWITCHES 8-19
CONTROL OPTION 12-3
CONTROL_STACK_BITS 8-19
CONTROL_STACK_TOP 8-19
CONVERT 8-20
CONVERTOOTS OPTION 12-3
COROUTINE STATEMENT 10-33
CSSIZE OPTION 12-3

DATA STRUCTURING 5-3
DATA TYPES 5-1
DATA_ADDRESS 8-22
DATA_LENGTH 8-22
DATA_TYPE 8-22
DATE 8-22
DC_INITIATE_IO 10-39
DEBLANK 10-40
DEBUG OPTION 12-3
DECIMAL CONVERSION 8-23
DECLARATION STATEMENT 3-1
DECLARATIONS 5-1
DECLARE STATEMENT 5-2
DECREMENT 8-4, 10-12
DEFINE INVOCATION 5-38
DEFINE STATEMENT 5-36
DELIMITED_TOKEN 8-23
DESCRIPTORS 8-10
DETAIL OPTION 12-3
DEVICE 5-22, 10-17
DISABLE_INTERRUPTS 10-40
DISK ALLOCATION 5-28
DISK DRIVE ASSIGNMENT 5-28
DISK FILE 5-26, 10-22
DISPATCH 8-24
DISPLAY STATEMENT 9-13
DISPLAY_BASE 8-25
DO GROUPS 10-2
DOUBLE OPTION 12-3
DUMMY 5-13
DUMP 10-40
DUMP_FOR_ANALYSIS 10-41
DYNAMIC DECLARATIONS 5-16
DYNAMIC FILE CHANGE 10-14
DYNAMIC_MEMORY_BASE 8-25
DYNAMICSIZE OPTION 12-3

ENABLE_INTERRUPTS 10-41

END OF STRING 10-11
END_OF_PAGE_ACTION 5-29, 10-19
ENTER_COROUTINE 10-33
ERROR FILE OPTION 12-3
ERROR_COMMUNICATE 10-41
ESSIZE OPTION 12-3
EU_ASSIGNMENT 5-28
EVALUATION_STACK_TOP 8-25
EXCEPTION MASK PART 5-31
EXECUTABLE STATEMENT 3-1
EXECUTABLE STATEMENTS 10-1
EXECUTE 8-25, 10-42
EXECUTE-FUNCTION STATEMENT 10-36
EXECUTE-PROCEDURE STATEMENT 10-35
EXIT_COROUTINE 10-33
EXPAND_DEFINES 12-3
EXPRESSIONS 7-1
EXTENDED ARITHMETIC FUNCTIONS 8-27

FETCH 10-42
FETCH_AND_SAVE 10-42
FETCH_COMMUNICATE_MSG_PTR 8-10
FILE ATTRIBUTE STATEMENT (CHANGE STATEMENT) 10-14
FILE DECLARATIONS 5-20
FILLER 5-12
FIND_DUPLICATE_CHARACTERS 10-42
FINI 3-1
FORMAL.CHECK 5-41
FORMAL_CHECK 5-10, , , , 6-5
FORMAL_VALUE 6-4, 8-16
FORMALCHECK OPTION 12-4
FORWARD DECLARATION 5-40
FREEZE 12-3
FREEZE_PROGRAM 10-43

GROW 10-43

HALT 10-44
HARDWARE MONITOR 13-1
HARDWARE_MONITOR 10-44
HASH_CODE 8-27
HEX_SEQUENCE_NUMBER 2-5

I/O CONTROL STATEMENTS 9-1
IDENTIFIER 5-37
IDENTIFIERS 2-1
IF STATEMENT 10-5
INDEXED FIELD REFERENCES 5-6
INDEXING 8-5
INITIALIZE_VECTOR 10-44
INTERPRETER OPTION 12-4
INTERROGATE_INTERRUPT_STATUS 8-28
INTRA-STRUCTURE REMAP 5-11
INTRINSIC HEAD 6-5

INTRINSIC OPTION 12-4

LABEL 5-21, 10-16
LAST_LIO_STATUS 8-28
LENGTH 8-28
LEXICOGRAPHIC LEVEL 3-2
LIMIT_REGISTER 8-28
LIST OPTION 12-4
LISTALL OPTION 12-4
LOCATION 8-29
LOCK 5-25, 10-19
LOCKI 12-4
LOGICAL OPERATORS 7-7

M_MEM_SIZE 8-35
MAKE_DESCRIPTOR 8-11
MAKE_READ_ONLY, MAKE_READ_WRITE 10-32
MERGE OPTION 12-4
MESSAGE_COUNT 10-45
MODE 5-24, 10-18
MODIFY STATEMENTS (CLEAR, BUMP, DECREMENT) 10-12
MONITOR 10-45, 12-4
MONITOR FILE 18-7
MONITOR SPEC PART 5-31
MONITOR_OUTPUT_FILE RESTRICTIONS 18-1, 18-6
MULTI PACK 5-28, 10-19

NAME_OF_DAY 8-29
NAME_STACK_TOP 8-30
NEST_PROCEDURE_TIMES 12-4
NESTING 3-2
NESTING LEVEL 6-7
NEW OPTION 12-4
NEXT_ITEM, PREVIOUS_ITEM 8-12
NEXT_TOKEN 8-30
NO OPTION 12-4
NO_DUPLICATES OPTION 12-4
NO_SOURCE_OPTION 12-4
NON-STRUCTURE DECLARATIONS 5-8
NSSIZE OPTION 12-5
NULL 8-12
NULL STATEMENT 10-13
NUMBER_OF_STATIONS 5-30
NUMBERS 2-3

ON SEQUENCE 9-16
OPEN OPTION 5-27
OPEN STATEMENT 9-2
OPERATOR PRECEDENCE TABLE 7-4
OPTIONAL FILE PART 5-31
OTHER CONSTANTS 2-5
OVERLAY 10-45

PACK_ID 5-27, 10-16

PAGE OPTION 12-5
PAGED ARRAY DECLARATIONS 5-15
PARITY SPECIFICATION 10-18
PARITY_ADDRESS 8-31
PASS END OPTION 12-5
POLISH NOTATION 7-2
PPROFILE 13-1
PPSSIZE OPTION 12-5
PREVIOUS_ITEM 8-12
PRIMARY ELEMENTS OF THE EXPRESSION 8-1
PROCEDURE BODY 6-6
PROCEDURE ENDING 6-8
PROCEDURE HEAD 6-2
PROCEDURE NESTING 3-4
PROCEDURE STATEMENT 3-1
PROCEDURES 6-1
PROCESSOR_TIME 8-31
PROFILE 13-1
PROFILE, PPROFILE OPTION 12-5
PROGRAM SEGMENTATION 4-1
PROGRAM SWITCHES 8-31
PROGRAM TIMING 13-2
PROGRAMMING OPTIMIZATION 13-1
PROGRAMMING TECHNIQUES 16-1

READ STATEMENT 9-6
READ_CASSETTE 10-46
READ_FILE_HEADER, WRITE_FILE_HEADER 10-30
READ_FPB, WRITE_FPB 10-46
READ_OVERLAY, WRITE_OVERLAY 10-47
RECOMPILATION FACILITY 17-1
RECOMPILE_TIMES OPTION 12-5
RECORD 5-4
RECORD REFERENCE DECLARATIONS 5-19
RECORD REFERENCE VARIABLES 5-19
RECORD SIZE 5-25, 10-22
RECORD STATEMENT 5-3
REDUCE STATEMENT 10-9
REEL NUMBER 5-26, 10-22
REFER ADDRESS 10-47
REFER LENGTH 10-48
REFER STATEMENT 10-8
REFER TYPE 10-48
REFERENCE DECLARATIONS 5-18
REINSTATE 10-48
RELATED PUBLICATIONS 1-2
RELATIONAL OPERATORS 7-6
REMAPPING 5-9, 5-13
REMOTE KEY 5-29
REPLACE OPERATORS 7-8
REPLACE, DESTRUCTIVE 7-8
RESERVED WORDS 11-1, 11-3
RESTORE 10-49
Restrictions: 5-17

RETURN STATEMENT 6-6
RETURN_AND_ENABLE_INTERRUPTS 6-6
REVERSE_STORE 10-49

S_MEM_SIZE, M_MEM_SIZE 8-35
SAVE 5-25, , 10-50
SAVE_STATE 10-50
SCOPE 3-5
SCOPE OF PROCEDURES 3-2
SEARCH_DIRECTORY 10-28
SEARCH_DIRECTORY STATEMENT 10-28
SEARCH_LINKED_LIST 8-32
SEARCH_SDL_STACKS 8-33
SEARCH_SERIAL_LIST 8-34
SEEK STATEMENT 9-11
SEGMENT, SEGMENT_PAGE 4-1
SEQ OPTION 12-5
SEQUENCE_NUMBER 2-5
SERIAL NUMBER PART 5-31
SINGLE SPACE OPTION 12-5
SIZE OPTION 12-5
SKIP STATEMENT 9-15
SORT 10-50
SORT_MERGE 10-51
SORT_SEARCH 8-35
SORT_STEP_DOWN 8-35
SORT_SWAP 10-52
SORT_UNBLOCK 8-36
SPACE STATEMENT 9-14
SPO_INPUT_PRESENT 8-36
STOP STATEMENT 10-26
STRUCTURE DECLARATIONS 5-11
STRUCTURE OF AN SDL PROGRAM 3-1
STRUCTURED RECORD STATEMENT 5-6
STRUCTURES 5-5
SUBBIT AND SUBSTR 8-8, 8-37
SUPPRESS OPTION 12-5
SWAP 8-38
SWITCH FILE DECLARATIONS 5-34

THAW_PROGRAM 10-52
THREAD_VECTOR 10-52
TIME 8-39
TIMER 8-39
TIMING OPTION 12-5
TODAYS_DATE 2-5
TRACE 10-53
TRANSLATE 10-54
TYPED PROCEDURES 8-15

UNARY OPERATORS 7-5
UNDERSCORES_IN_FILE_NAMES OPTION 12-5
UNDO 10-4
USE INPUT BLOCKING 5-29, 10-19

USE STATEMENT 5-43
USEDOTS OPTION 12-5

VALUE GENERATING FUNCTIONS 8-16
VALUE VARIABLES 8-14
VALUE_DESCRIPTOR 8-40
VARIABLE DATA FIELDS 6-4
VARIABLE RECORD 5-24, 10-19
VOID OPTION 12-5
VSSIZE OPTION 12-6

WAIT 8-40
WORK FILE 5-30
WORKING_SET_BYTES OPTION 12-6
WRITE STATEMENT 9-8
WRITE_FILE_HEADER 10-30
WRITE_OVERLAY 10-47

X_ADD 8-27
X_DIV 8-27
X_MOD 8-27
X_MUL 8-27
X_SUB 8-27
XMAP OPTION 12-6
XREF 12-6
XREF_ONLY 12-6

ZIP STATEMENT 10-27

8-22 TIMER

<TIMER DESIGNATOR>::= TIMER

A value of type BIT(24) is returned. The value is the current setting of the TIME register.

DATA LENGTH

<DATA_LENGTH DESIGNATOR>::= DATA_LENGTH (<EXPRESSION>)

Returns the length in bits of <EXPRESSION>, regardless of the data type.

DATA TYPE

<DATA_TYPE DESIGNATOR>::= DATA_TYPE (<EXPRESSION>)

Returns the type bits of <EXPRESSION>.

10-47 REFER ADDRESS

<REFER_ADDRESS DESIGNATOR>::= REFER_ADDRESS (<REF VAR>, <EXRESSION>)

The value of <EXPRESSION> is stored ir the address of <REF VAR>.

10-48 REFER LENGTH

<REFER_LENGTH_ DESIGNATOR>::= REFER LENGTH (<REFER VAR>, <EXPRESSION>)

The value of <EXPRESSION> is stored in the length of <REF VAR>.

REFER TYPE

<REFER_TYPE_ DESIGNATOR>::= REFER_TYPE (<REF VAR>, <EXPRESSION>

The value of <EXPRESSION> is stored in the type part of <REF VAR>.

SDL BNF (P.S. 2212 5405)

5-6 STRUCTURED RECORD STATEMENT

```
<STRUCTURED RECORD STATEMENT> ::=
            RECORD 01 <RECORD IDENTIFIER> <TYPE>
            <STRUCTURE ELEMENTS>

<RECORD IDENTIFIER> ::= <IDENTIFIER>

<STRUCTURE ELEMENTS> ::=
              ,<LEVEL NUMBER> <STRUCTURE ELEMENT>
            | ,<STRUCTURE ELEMENTS>
<STRUCTURE ELEMENT> ::=
              <FIELD NAME> <TYPE>
            | <FIELD NAME> <ARRAY BOUND> <TYPE>
            | FILLER <TYPE>
            | <FIELD NAME> REMAPS <REMAPS OBJECT> <TYP
```

Structured Records have been implemented to allow easier conversion of the current PL/I-style structures to records.

Structured Records have the same capabilities as RECORDS.

Fields declared as an array may not have nested structure.

BINARY SEARCH

```
<BINARY_SEARCH FUNCTION>::=  BINARY_SEARCH
                             (<START_RECORD>, <COMPARE_FIELD>,
                             <COMPARE_VALUE>, <NUMBER_OF_RECORDS>)

<START_RECORD>::=            <EXPRESSION>
<COMPARE_FIELD>::=           <TEMPLATE>
<COMPARE_VALUE>::=           <EXPRESSION>
<NUMBER_OF_RECORDS>::=       <ADDRESS GENERATOR>
```

BINARY_SEARCH searches an ordered list of items that start at <START_RECORD> for <NUMBER_OF_RECORDS> for a match.

The occurrence number of the entry that matches will be returned, or if there is no match, the occurrence number of the first entry that is greater will be returned.

8-18 NOTE: The comparison is always left justified and uses the length of <COMPARE VALUE>.