B 1 0 0 0

S D L 2 C O M P I L E R

BURROUGHS CORPORATION

COMPANY CONFIDENTIAL

JUNE 1984

BURROUGHS CORPORATION
COMPUTER SYSTEMS GROUP
SANTA BARBARA PLANT

TABLE OF CONTENTS

INTRODUCTION

This document describes the syntax and operating instructions for the second generation B1000 system development language, SDL2. Appendix A documents the corresponding user programming language, UPL2. Appendix B documents the differences between SDL and SDL2.

RELATED DOCUMENTATION

NAME	NUMBER
SDL2 SMACHINE	2228 3568
ISSA	2233 2845
MCP COMMUNICATES AND STRUCTURES	2228 3659
MCP CONTROL SYNTAX	2228 3501
B1000 MCPII	2212 5462

SDL2 LANGUAGE

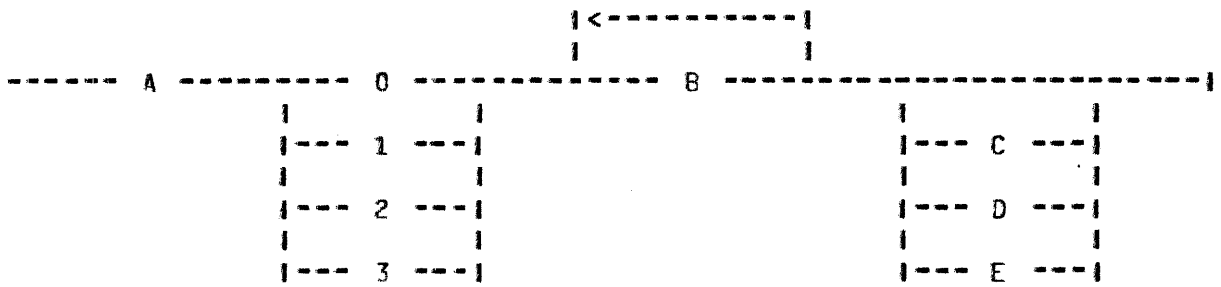
The SDL2 language is described with syntax diagrams.

A syntax diagram consists of a main line which contains all required elements of the syntax. The main line may have other lines above and below it. Lines above the main line indicate loops back in the syntax, and lines below the main line indicate alternate productions. Continuation from one line of a diagram to another is represented by a right arrow, ">", at the end of the current line and the beginning of the next line. A complete syntax diagram is terminated by a vertical bar, "|".

If an element appears on the main line of the syntax diagram with no lines below it, then that element is required. If an element appears on the main line of the diagram but other elements appear in lines directly below it, then one of those elements is required. If an element appears in a line below the main line and the main line above it is empty, then the element is optional.

Elements which are in uppercase, or are special characters, are terminal symbols and must appear literally in the final production. Lowercase elements reference a syntax diagram of that name.

Example:



where

A0B	is valid
A1BB	is valid
A2BBBC	is valid
A0	is invalid

TOKENS

SDL2 recognizes the following token types: identifiers, bit strings, fixed numbers, character strings, and special characters.

Comments are allowed anywhere in the input stream, except within quoted strings. There are two forms of comments. The first form begins with a % and terminates with the end of that line. The second form begins with /* and terminates with */. A comment enclosed in /* */ may span multiple lines.

Blanks, special characters, comments, and end-of-lines are all treated as separators, and so cannot be embedded within a token.

Identifiers

An SDL2 identifier may contain a maximum of 72 characters. Valid characters include upper-case letters (A-Z), lower-case letters (a-z), numerals (0-9), and the underscore (_). Identifiers must begin with a letter, and are not normally case-sensitive.

Example:

```
THIS_TOKEN
A100
XYZ
```

Fixed Numbers

A string consisting entirely of numerals, without any bracketing characters, is treated as a fixed number.

Example:

```
12366
400
```

Character Strings

Character strings are bracketed by the double quote character ("). If a quote is desired within a quoted string, then two quotes ("") must be used.

Example:

```
"ABCDEF"
"KEYWORD ""THEN"" IS OUT OF CONTEXT"
```

Bit Strings

Bit strings are bracketed by the at-sign character (@). The string may be either binary, quartal, octal, or hexadecimal, where the number of bits per digit is specified within parentheses. The default is hexadecimal (4 bits per digit).

Example:

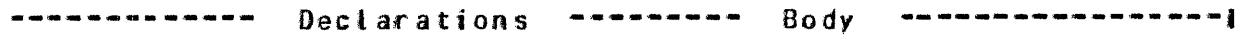
```
@0AFF@
@(1)0100@
@(2)0122@
```

Special Characters

SDL2 recognizes the following special characters.

&	.	,	<	@	#
$	:	+	>	"	%
(	;	-	*	[	
)	!	=	/	]	

STRUCTURE OF AN SDL2 PROGRAM

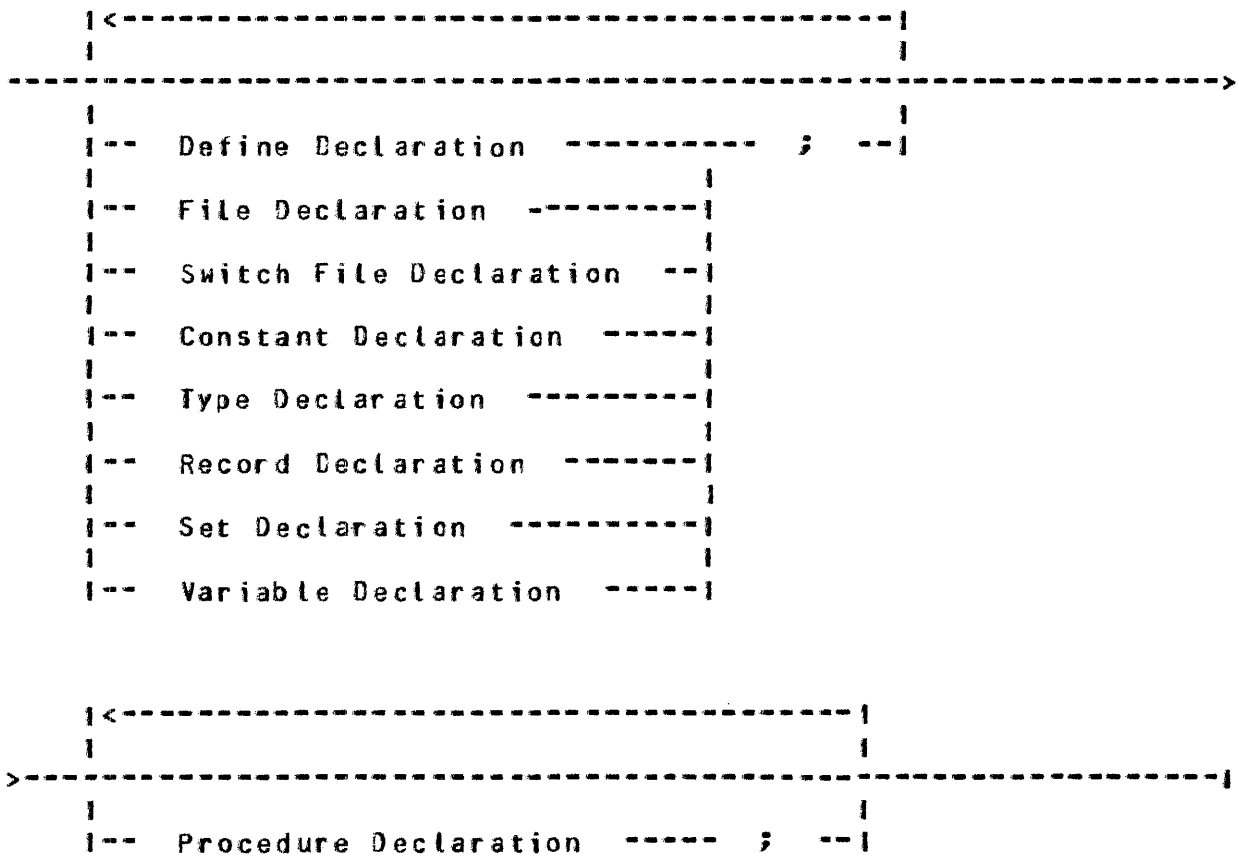


The structure of an SDL2 program must follow the sequence described in the syntax graphs. All variables, files, defines and data types must be declared before the procedures, and all procedures must be declared before the main executable part of the program (Body).

Program execution begins with the first statement in the program Body.

DECLARATIONS

DECLARATIONS



All identifiers must be declared before they can be referenced, except within the define text of a define declaration where they must be declared before the expansion of the define.

Scope of an Identifier

For global declarations, the scope of an identifier is all the subsequent declarations, nested procedures, and the main body of the program, except for any nested procedures, that declare another version of the identifier.

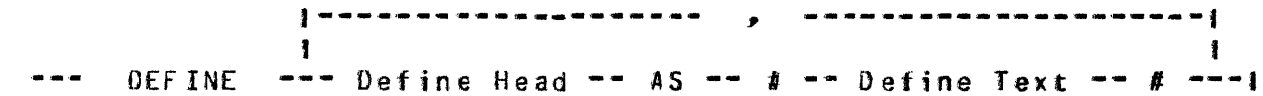
For a declaration within a procedure, the scope of an identifier is that procedure and any nested procedures, except for any that declare another version of the identifier.

Restrictions

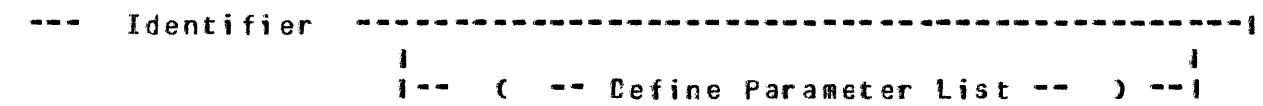
File Declarations and Switch File Declarations are allowed only at the global level (lexic level 0).

Intrinsics, Separate Units, and Independent Units may not declare any Files, Switch Files, or Variables at the global level. They may, however, declare global Defines, Constants, Types, Records, and Sets.

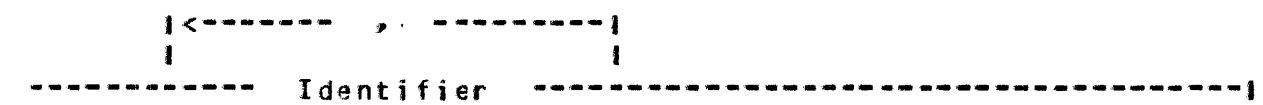
DEFINE DECLARATION



Define Head



Define Parameter List



The Define Declaration allows a string of text to be allocated an identifier. Whenever this identifier is encountered in the source program it is replaced by the text (except in a define declaration). Define expansion is also turned off when the compiler is expecting an identifier in any of the following: Declaration Identifier, Define Identifier, Procedure Identifier, Formal Parameter Identifier, Do Group Identifier.

A define declaration may have parameters. When the define text is invoked there must be a text string for every parameter. This string will replace all occurrences of the define parameter identifier in the define text. Parameters containing commas which are not enclosed in parentheses, or parameters containing mismatched parentheses, must be enclosed in pound signs (#).

Defines are included in SDL2 to ease the conversion from SDL. In most cases defines can be replaced by Type and Constant statements, resulting in faster compilation speeds.

Example:

```
DEFINE
  ARRAY_SIZE AS #1024#,

  MAX (X,Y)  AS # IF X > Y THEN X ELSE Y #;
```

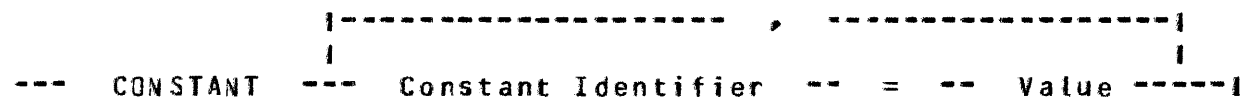
FILE DECLARATION

See INPUT/OUTPUT Section.

SWITCH FILE DECLARATION

See INPUT/OUTPUT Section.

CONSTANT DECLARATION



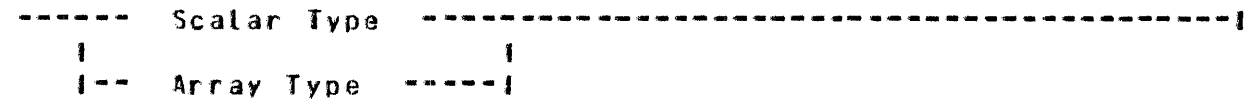
The Constant Declaration introduces an identifier as a synonym for a constant. Value may be either a character or bit literal, an expression which evaluates to a fixed constant, or a previously defined constant.

Example:

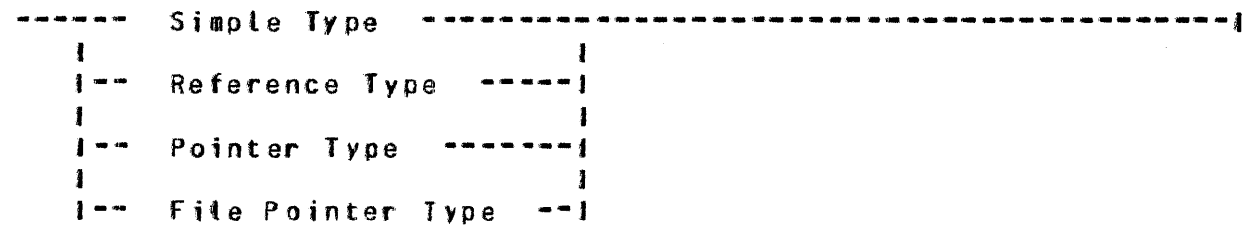
```
CONSTANT    TEN   = 10,
            FIFTY = 5 * TEN,
            LEVEL = "11.0";
```

TYPES

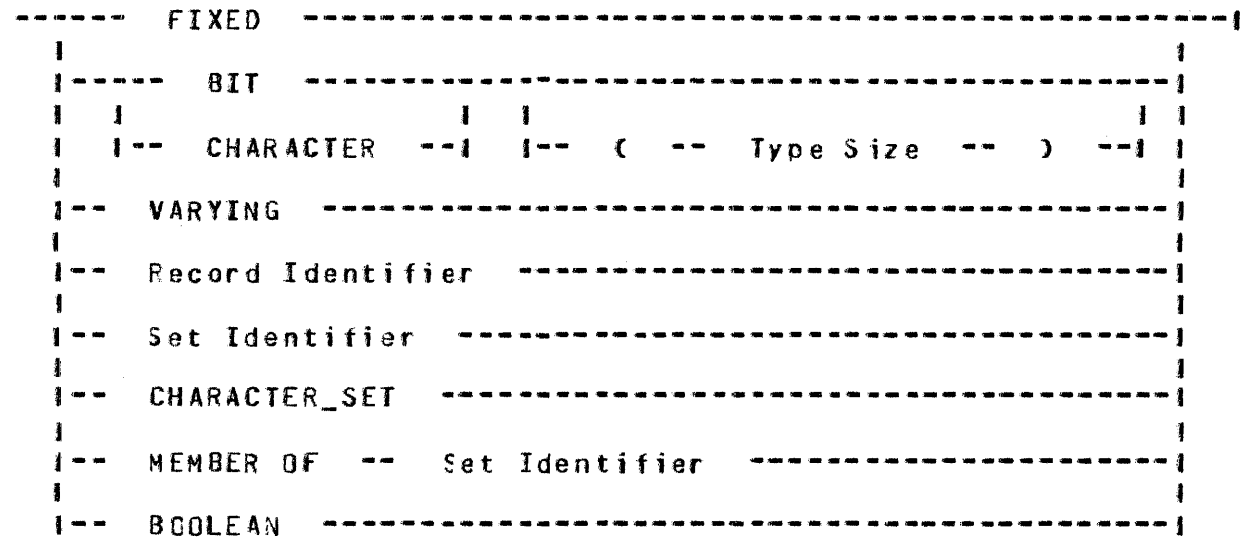
Type



Scalar Type



Simple Type



The type FIXED will allocate a 24 bit field, where the high order bit is interpreted as the sign bit (0 = positive, 1 = negative). Negative values are represented in their two's complement form. Fixed numbers can range from -(2**23)+1 to (2**23)-1.

The type BIT will allocate a field of <Type Size> bits. It can have a maximum size of 65535 bits. <Type Size> can only be omitted in the <Type> part of a formal parameter declaration, the

<Type> part of a procedure declaration, the <Type> part of a global dynamic variable (see ALLOCATE), or a Reference Type declaration, indicating that the size is dynamic.

The type CHARACTER will allocate a field of <Type Size> characters. Each character occupies 8 bits. It can have a maximum size of 8191 characters. <Type Size> can only be omitted in the <Type> part of a formal parameter declaration, the <Type> part of a procedure declaration, the <Type> part of a global dynamic variable (see ALLOCATE), or a Reference Type declaration, indicating that the size is dynamic.

The type VARYING can only be used in a procedure declaration or a formal parameter declaration. It indicates that both the type and length are dynamic.

The type <Record Identifier> will allocate a field with the length of the sum of the fields in the Record Declaration, and make available the field names for field selection. The field will be of type Character if all subfields in the Record Declaration are Character; otherwise it will be of type Bit.

The type <Set Identifier> will allocate a field with a length equal to the number of members declared in the Set Declaration (in bits).

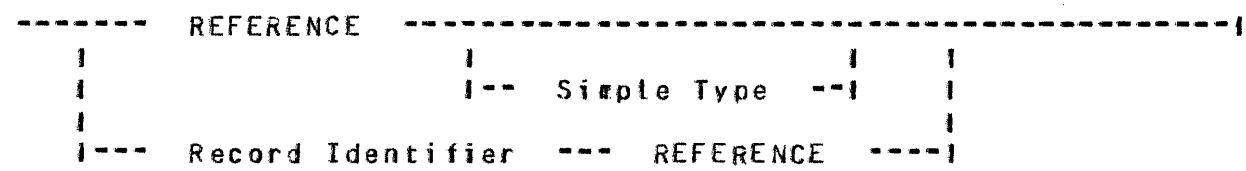
The type CHARACTER_SET is a predefined Set Declaration with 256 members, the EBCDIC character set.

The type MEMBER OF will allocate a field with a length equal to the number of bits required to represent the set member with the largest ordinal value.

The type BOOLEAN is a predefined Set Member Declaration. It is defined as a member of a set containing two members, FALSE and TRUE, with ordinal values 0 and 1, respectively. In the explanation of types BIT and CHARACTER, include "global dynamic variables" as a case where <Type Size> may be omitted, and refer to the ALLOCATE standard procedure for more information.

In the explanation of <Array Type>, include "global dynamic variables" as a case where an asterisk is allowed in place of <Array Bound>, and refer to the ALLOCATE standard procedure for more information.

Reference Type



If a variable is declared to have a type of reference in a Variable Declaration, a descriptor is allocated for it on the Descriptor Stack. It originally has no data associated with it, but can be REFERed to some data space. If <Simple Type> has a constant (non-zero) length, then the reference variable is coerced to have the specified type and length, and any REFERs can alter only its data address, but not its type or length. If <Simple Type> is just BIT or CHARACTER without a length, then the type serves only as a comment, unless the INITIALIZE_REFERENCES compiler control option is set. (See INITIALIZE_REFERENCES in the COMPILER CONTROL Section).

It is the programmer's responsibility to guarantee that a reference variable has been REFERed to valid data space before it is used. The use of an uninitialized reference variable will produce unpredictable results.

If a field is declared to have a type of reference in a Record Declaratation, 96 bits are allocated. When it is REFERed, a symbolic form of the descriptor is stored in the field. The selection of a field that is a reference type will cause an automatic dereference; the descriptor can only be accessed with a Refer statement.

The second form of Reference Type declarations, with REFERENCE specified last instead of first, is included only as a means of conversion from SDL to SDL2.

Example:

```
DECLARE
   TEXT REFERENCE CHARACTER,
   SOURCE CHARACTER (80);

REFER TEXT TO SOURCE;
```

Pointer Type

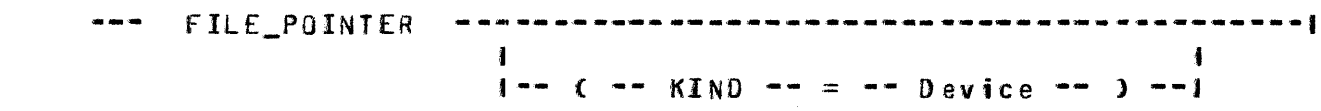
```
-----  POINTER  --  Simple Type  ------------------------------|
```

The <Simple Type> must be a constant length type. A field of 24 bits will be allocated to contain the address of the REFERed data space. A pointer variable has no data associated with it until it is REFERed to a data item. Whenever a pointer is used it will automatically be dereferenced to the field pointed to by the pointer. There is no protection if a pointer is REFERed into dynamic memory and that page gets rolled out of memory. The main use of pointers is for the MCP. For most other applications, reference variables should be used instead of pointers.

Example:

```
RECORD RS_NUCLEUS
    .
    .
    .
    MCP_BIT       BIT(1),

RECORD HINTS_FORM
    .
    .
    .
    FIRST_Q       POINTER RS_NUCLEUS,
    .
    .
    .
    MCP_RSN       POINTER RS_NUCLEUS,
    .
    .
    .

DECLARE
    HINTS         HINTS_FORM,
    ACTUAL_RSN    RS_NUCLEUS;

REFER HINTS.MCP_RSN TO ACTUAL_RSN;
HINTS.MCP_RSN.MCP_BIT:=TRUE;
```

File Pointer Type



A file pointer is used as a programmatic reference of a file. A Refer statement is used to assign a logical file to a file pointer variable. The file pointer may then be used in any context where a file is allowed. For example, file pointers are allowed in any I/O statement. File pointers may also be passed as parameters, whereas conventional files may not. Note, however, that file pointer parameters may not be passed by value.

It is the programmer's responsibility to guarantee that a file pointer has been REFERed to a valid file before it is used, since I/O statements with an unassigned file pointer will produce unpredictable results.

Device can be any legal device kind. (See the KIND file attribute in the INPUT/OUPUT Section). The default device kind is DISK. The Kind specification is required to distinguish port and remote files from other kinds of files. Port and remote file pointers may only be REFERed to port or remote files. Similarly, file pointers which are not declared as port or remote may not be Referred to port or remote files. These restrictions are required because the compiler must generate different I/O communicates for port and remote files.

File pointers are not allowed as array elements, record fields, or function return values.

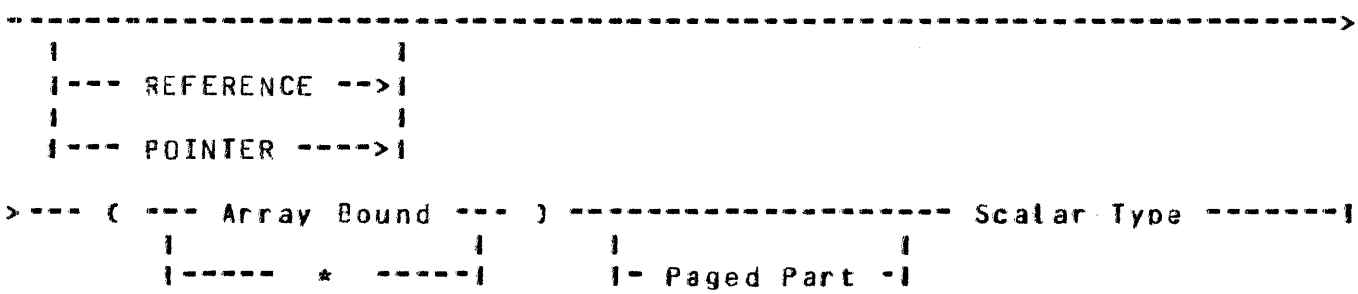
Example :

```
FILE      F1          (KIND = DISK),
          F2          (KIND = PRINTER),
          F3          (KIND = REMOTE);

DECLARE   (X, Y)      FILE_POINTER,
          Z           FILE_POINTER (KIND = REMOTE),
          BUFFER      CHARACTER (80);

          PROCEDURE   TALK (F);
          FORMAL      F  FILE_POINTER;
          F.RECORDSIZE := 80;
          OPEN F;
          WRITE F (BUFFER);
          CLOSE F LOCK;
          END TALK;
```

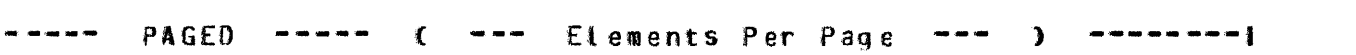
```
        PROCEDURE GOSSIP (F);
        FORMAL    F FILE_POINTER (KIND = REMOTE);
        WRITE F (BUFFER);
        READ  F (BUFFER);
        END GOSSIP;

    BUFFER := "";
    REFER X TO F1;
    REFER Y TO F2;
    REFER Z TO F3;
    BUFFER := "HI THERE";
    TALK (X);
    TALK (Y);
    GOSSIP (Z);
```

Note that TALK (Z), GOSSIP (Y), or REFER Z TO F1 are not allowed, and will be syntaxed by the compiler.

Array Type



Paged Part



An array is a collection of individual elements which are all of the same Scalar Type. Each element of the array can be referenced by using the name of the array variable, followed by the element number enclosed in parentheses. Elements are numbered from 0 to Array Bound - 1.

Array Bound and Elements Per Page can range from 1 to 65535. Elements Per Page will be rounded down to the next lower power of 2. An asterisk may be used in place of Array Bound only within a formal parameter declaration, in a REFERENCE or POINTER array, or a global dynamic array (see ALLOCATE).

If Paged Part is specified, the array will be allocated to dynamic memory. Pages will be allocated only as necessary, and the bounds of the array are allowed to increase past Array Bound. However Array Bound will be used to determine how much memory will be initially assigned to the non-overlayable page table for the array. Paged arrays are not allowed within formal parameter declarations, REFERENCE or POINTER array types.

Example:

```
DECLARE
  A (10)  BOOLEAN,
  P (256) PAGED (32) CHARACTER(30);
```

Programs using paged arrays may find it necessary to increase Dynamic Memory and/or Virtual Disk above the default values supplied by the compiler.

Reference Arrays

System programs sometimes need to treat an area of memory as a large array. A reference array allows the programmer to set up a reference to an area of memory at an arbitrary address, with any entry size up to 8KB and any number of entries up to 16777215.

None, some, or all of the following attributes may be included in the declaration of the array: Bound (number of entries), element type, and element size. For POINTER arrays, all of these attributes must be specified at declaration time--none may vary dynamically. For REFERENCE arrays, any attributes missing from the declaration may be changed using the appropriate standard procedures described below.

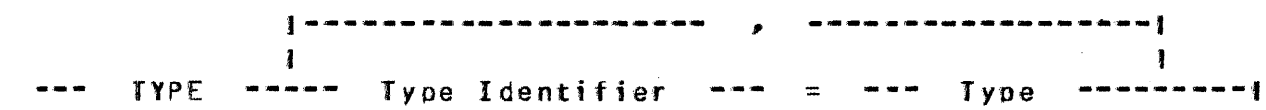
For each REFERENCE array declared, an array descriptor is allocated. Any attributes specified in the declaration are automatically used to initialize this descriptor (regardless of the setting of the INITIALIZE_REFERENCES dollar option), and may not be changed. Any attributes not specified are initialized to the same default values used by the INITIALIZE_REFERENCES dollar option and if the bound is varying then it is initialized to zero. Later these varying attributes may be changed by the programmer, using the REFER_ADDRESS, REFER_TYPE, REFER_LENGTH or REFER_BOUND standard procedures. For each POINTER array declared, a 24-bit address field is allocated. Only the REFER_ADDRESS procedure is allowed with pointers. REFERENCE or POINTER arrays are not allowed in the REFER statement.

REFERENCE and POINTER arrays may not be PACED, nor appear as function return values or formal parameters. Further, REFERENCE arrays are not allowed as record fields or array elements.

As an example of a reference array, suppose a programmer wants to set up a reference to an area of memory to be treated as an array of some arbitrary number of elements, where each element is to be 180 characters in length. If the address of the area is BUF_ADDR and the number of elements the area will hold is BUF_SECTORS, then say:

```
DECLARE RA REFERENCE (*) CHARACTER (180);
...
REFER_ADDRESS (RA, BUF_ADDR);
REFER_BOUND   (RA, BUF_SECTORS);
...
REFER THIS_SECTOR TO RA(I);
etc.
```

TYPE DECLARATION

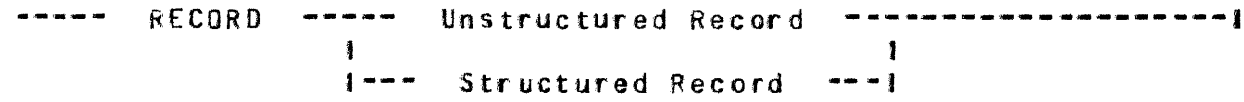


The Type Declaration allows an identifier to appear as a Type inside a Variable Declaration, Procedure Declaration, Record Declaration, or Type Declaration. Dynamic types are not allowed.

Example:

```
TYPE      STATE_INDICATOR = MEMBER OF STATE_SPACE,
          ADDRESS         = BIT (24),
          STATE_VECTOR    = (8) STATE_INDICATOR;

DECLARE   STATE     STATE_INDICATOR,
          VECTOR    STATE_VECTOR,

IF VECTOR (I) IN TERMINAL_STATES THEN TERMINATE (I);
```

RECORD DECLARATION

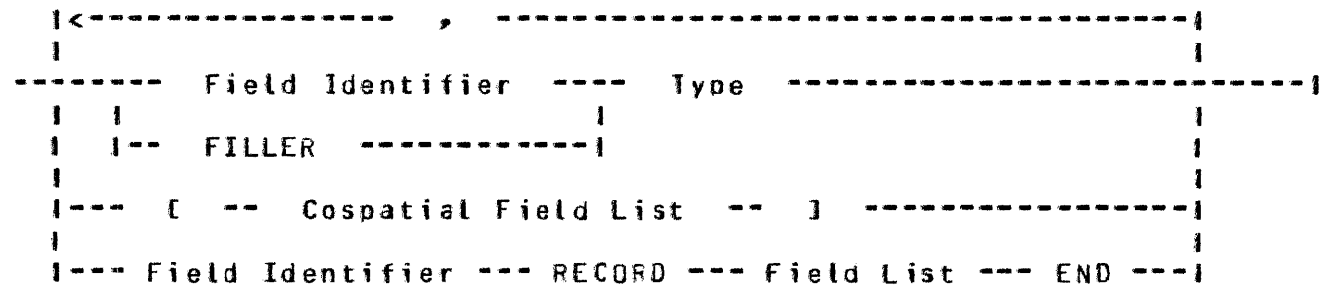


The Record Declaration is a convenient way of mapping the layout of a structure. It allows variables and fields within other records to be given structure. (See Selector in EXPRESSIONS Section and WITH STATEMENT in STATEMENTS Section for referencing of subfields).

The structured record declaration is included in SDL2 only as a means of easier conversion of the SDL PL/I style structures. The unstructured record declaration is the preferred form.

Unstructured Record

Field List



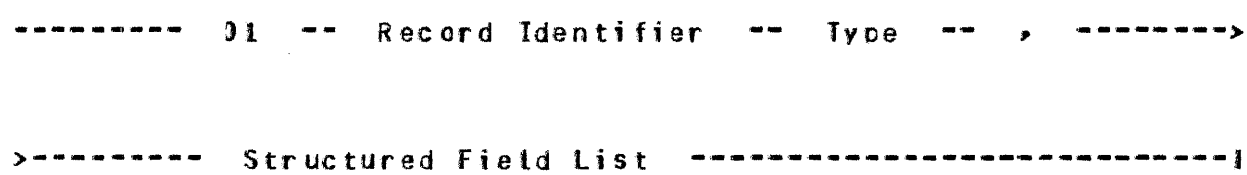
Cospatial Field List

Fields may be declared cospatial. That is, each field list remaps the other cospatial field lists, and the length will always be the length of the longest field list.

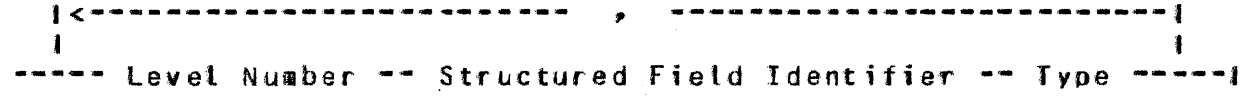
Example:

```
RECORD SYSTEM_DESCRIPTOR_FORM
  IN_USE                    BIT(1),
  MEDIA                     BIT(1),
  LOCK                      BIT(1),
  IN_PROCESS                BIT(1),
  INITIAL                   BIT(1),
  FILE_OBJECT               BIT(1),
  DK_FACTOR                 BIT(3),
  SEG_PG                    BIT(7),
  TYPE                      BIT(4),
  [ ADDR                    BIT(36)
  | FILLER                  BIT(12),
    CORE                    BIT(24)
  ],
  LEN                       BIT(24);

  DECLARE
     SYSTEM_DESCRIPTOR  SYSTEM_DESCRIPTOR_FORM;

IF SYSTEM_DESCRIPTOR.MEDIA
     THEN READ_TO_MEMORY;
```

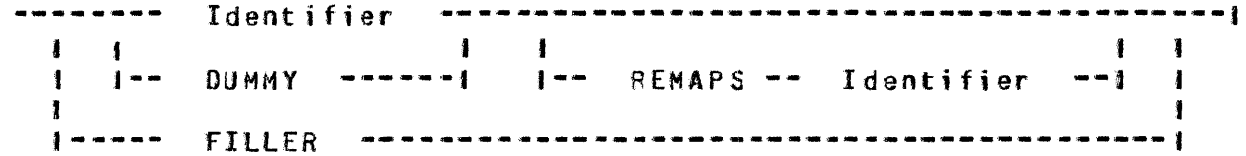
Structured Record



Structured Field List



Structured Field Identifier



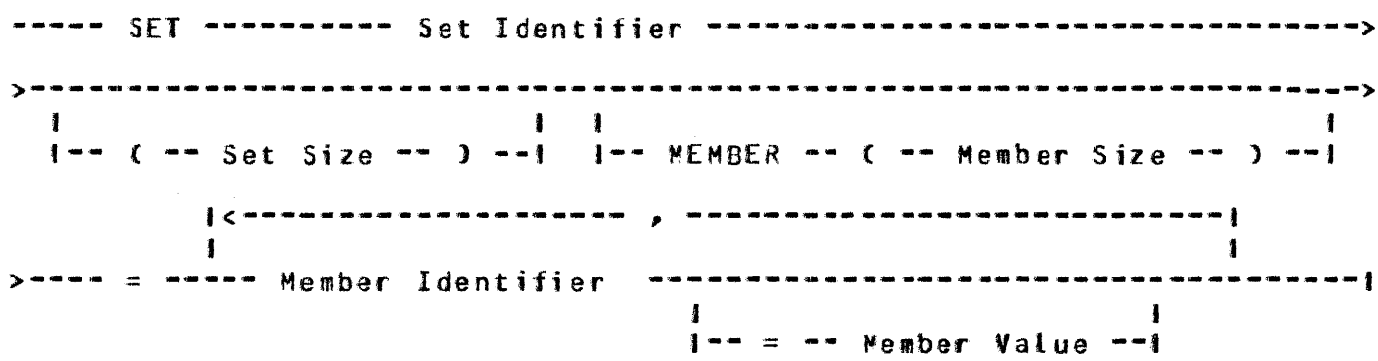
Example:

```
RECORD 01 MEMORY_LINK BIT(MEMORY_LINK_SIZE),
            02 ML_DISK                          DSK_ADDR,
            02 ML_GROUP                         BIT(47),
              03 ML_POINTER                     ADDRESS,
              03 ML_JOB_NUMBER                  BIT(16),
              03 ML_TYPE                        BIT(6),
              03 ML_SAVE                        BIT(1),
            02 ML_SIZE                          BIT(24),
            02 ML_PRIORITY_FIELD                BIT(30),
              03 ML_DK_INTERVAL                 BIT(10),
              03 ML_CURRENT_DK_INT              BIT(10),
              03 ML_INCOMING_PRIORITY           BIT(5),
              03 ML_RESIDENCE_PRIORITY          BIT(5),
                04 ML_RP_WHOLE                 BIT(4),
                04 ML_RP_FRACTION              BIT(1),
            02 ML_FRONT                         BIT(24),
            02 ML_BACK                          BIT(24),
            02 ML_USAGE_BITS                    BIT(2),
              03 ML_PREVIOUS_SCAN_TOUCH         BIT(1),
              03 ML_CURRENT_SCAN_TOUCH          BIT(1);
```

The above structured record in the preferrec form appears as:

```
RECORD   MEMORY_LINK
               ML_DISK                          DSK_ADDR,
               ML_GROUP                         RECORD
                 ML_POINTER                     ADDRESS,
                 ML_JOB_NUMBER                  BIT(16),
                 ML_TYPE                        BIT(6),
                 ML_SAVE                        BIT(1)
                 END,
               ML_SIZE                          BIT(24),
               ML_PRIORITY_FIELD                RECORD
                 ML_DK_INTERVAL                 BIT(10),
                 ML_CURRENT_DK_INT              BIT(10),
                 ML_INCOMING_PRIORITY           BIT(5),
                 ML_RESIDENCE_PRIORITY          RECORD
                   ML_RP_WHOLE                  BIT(4),
                   ML_RP_FRACTION               BIT(1)
                   END
                 END,
               ML_FRONT                         BIT(24),
               ML_BACK                          BIT(24),
               ML_USAGE_BITS                    RECORD
                 ML_PREVIOUS_SCAN_TOUCH         BIT(1),
                 ML_CURRENT_SCAN_TOUCH          BIT(1)
                 END;

CONSTANT MEMORY_LINK_SIZE = TYPE_LENGTH (MEMORY_LINK);
```

SET DECLARATION



The Set Declaration is used to define the members of a set. Each member is given an ordinal value, in ascending order, beginning with zero. (Set members are very similar to Pascal's Scalar Type constants).

CHARACTER_SET is a predefined Set type whose 256 members are the EBCDIC character set. The predefined type BOOLEAN is a member of the predefined Set = FALSE, TRUE. Hence FALSE has the value 0 and TRUE the value 1.

See also MEMBER OF in TYPES Section.

Example:

```
SET SYMBOL_KIND_SET =
    INVALID,
    NUMBER,
    STRING,
    IDENTIFIER,
    SPECIAL;

DECLARE
  NUMERICS  CHARACTER_SET,
  ALPHAS    CHARACTER_SET,
  SPECIALS  CHARACTER_SET,
  SYMBOL    MEMBER OF SYMBOL_KIND_SET,
  OPERANDS  SYMBOL_KIND_SET,
  CH        CHARACTER(1);

OPERANDS := [NUMBER, STRING, IDENTIFIER];
NUMERICS := ["0", "1", "2", "3", "4", "5", "6," "7", "8", "9"];
ALPHAS := ["A" TO "I", "J" TO "R", "S" TO "Z"];

IF CH IN NUMERICS
  THEN SYMBOL := NUMBER;
ELSE IF CH IN ALPHAS
```

```
      THEN SYMBOL := IDENTIFIER;
    ELSE IF CH = """"
      THEN SYMBOL := STRING;
    ELSE IF CH IN SPECIALS
      THEN SYMBOL := SPECIAL;
    ELSE SYMBOL := INVALID;

    IF SYMBOL IN OPERANDS
      THEN GET_OPERAND;
      ELSE CASE SYMBOL OF
             SPECIAL : GET_SPECIAL;
             INVALID : ERROR ("INVALID SYMBOL");
           END CASE;
```

Note that the compiler performs some checking to see that sets and members are not accidentally misused. A member declared within a Set Declaration belongs only to its parent set. Any single character value can belong to any character set. Fixed or bit values can belong to any bit string, or any constant set consisting only of fixed and bit members.

See also CHR and ORD in STANDARD FUNCTIONS Section.

Optional Container Sizes and Member Values

System programs which manage persistent disk structures, or which interface with hardware using predefined bit formats, may sometimes need to specify exact field widths or explicit member values for sets. In these cases, the compiler's default values or container-size may be overridden by the programmer, by including Set size, Member Size, or Member Value in the SET declaration.

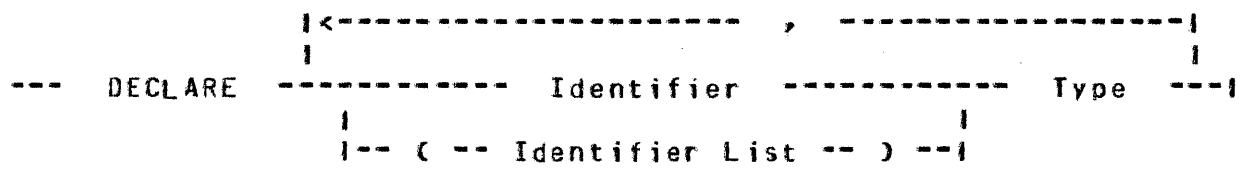
Note that if one Member Value is specified, then all Member Values must be specified. Also, the Member Values must be given in ascending order.

For example:

```
SET review_kinds (128) MEMBER (8) =
  poor   = 0,
  or     = 4,
  good   = 6,
  great  = 9,
  rave   = 10,

RECORD review-record
  responses   review_kinds,
  my_opinion  MEMBER OF review_kinds;
```

In this record, the "responses" field will be 128 bits wide, and the "my-opinion" field will be 8 bits wide.

VARIABLE DECLARATION



Identifier List

The Declare Statement allocates memory storage for variables and defines their type.

If the type size and/or the array bound of <Type> is a non-constant expression, then the declaration will be considered a dynamic declaration. Dynamic variables are not allowed at lexic level 0. Dynamic variables will be allocated a descriptor, and space will be allocated for the variables on procedure entry.

Paged arrays are allowed at any lexic level. They will be allocated a descriptor, and their memory space will be allocated from dynamic memory as needed.

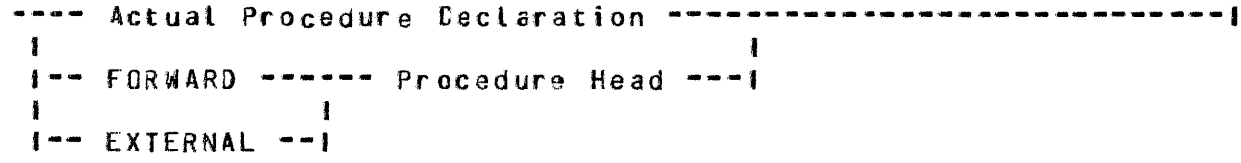
Static variables will be allocated space in memory and will not have a descriptor.

Reference variables do not have space allocated, but get a descriptor allocated that may be REFERed to some data space.

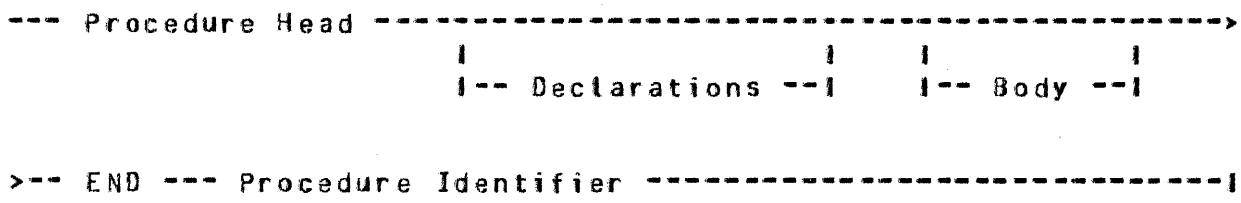
Example:

```
DECLARE
  I                      FIXED,
  (J, K)                 CHARACTER(10),
  A (10)                 FIXED,
  R REFERENCE            FIXED,
  P (1024) PAGED (16)    FIXED;
```

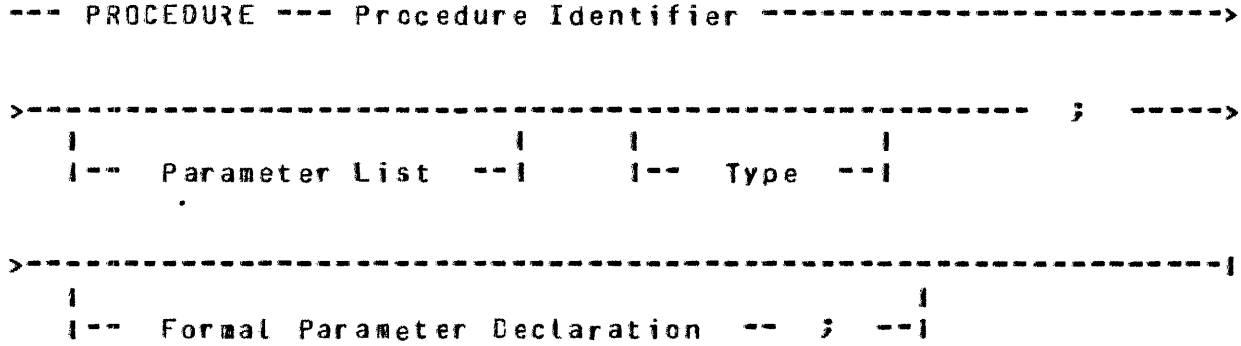
PROCEDURE DECLARATION



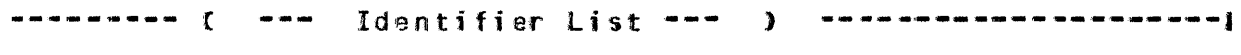
Actual Procedure Declaration



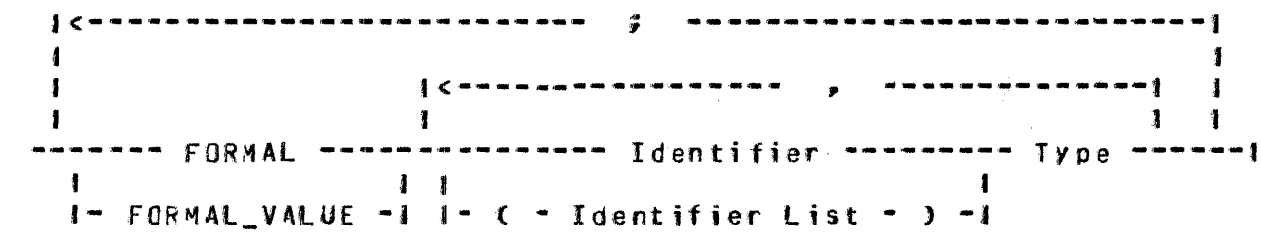
Procedure Head



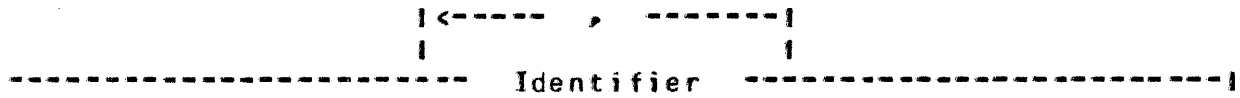
Parameter List



Formal Parameter Declaration



Identifier List



The Procedure Declaration is used to define a program fragment and to associate it with an identifier, so that it can be activated by a Call Statement or Typed Procedure Call.

Declarations within a procedure only have the scope of that procedure (i.e. when the <END Procedure Identifier> is encountered they are no longer accessible to the programmer). If a procedure needs to be called before it is declared, a FORWARD declaration must be made to define it and its formal parameters. If the procedure is declared in another program unit (see SEPARATE COMPILATION), then an EXTERNAL declaration is needed. EXTERNAL declarations are only allowed at lexic level 0.

Procedure declarations may be nested to a maximum of 15 levels.

The order of declaration of formal parameters must match the order in <Parameter List>.

If a parameter is declared FORMAL_VALUE, a copy of the data will be made, and the original data will be preserved. Arrays and file pointers may not be declared FORMAL_VALUE.

If a formal parameter is a Reference or Pointer variable, its address or length may be adjusted inside the procedure via a REDUCE or REFER statement. However these adjustments will only affect the parameter within the procedure body, and vanish on procedure exit.

If a parameter is declared as an Array, then any actual argument passed to the procedure must also be an Array, though it need not be of the same type or size.

Note that no type coercion is done for either FORMAL or FORMAL_VALUE parameters, unless they are declared as Records. For non-Record type parameters, actual arguments passed in a procedure call will maintain their originally declared type, and the type from the parameter declaration will be ignored. No type checking is done unless the parameters are Sets or Members, in which case a syntax error will be given if the actual argument is not of the same type as the corresponding parameter declaration. If a parameter is declared as a Record, then any actual argument passed to the procedure must be of the same length, though it need not be of the same type.

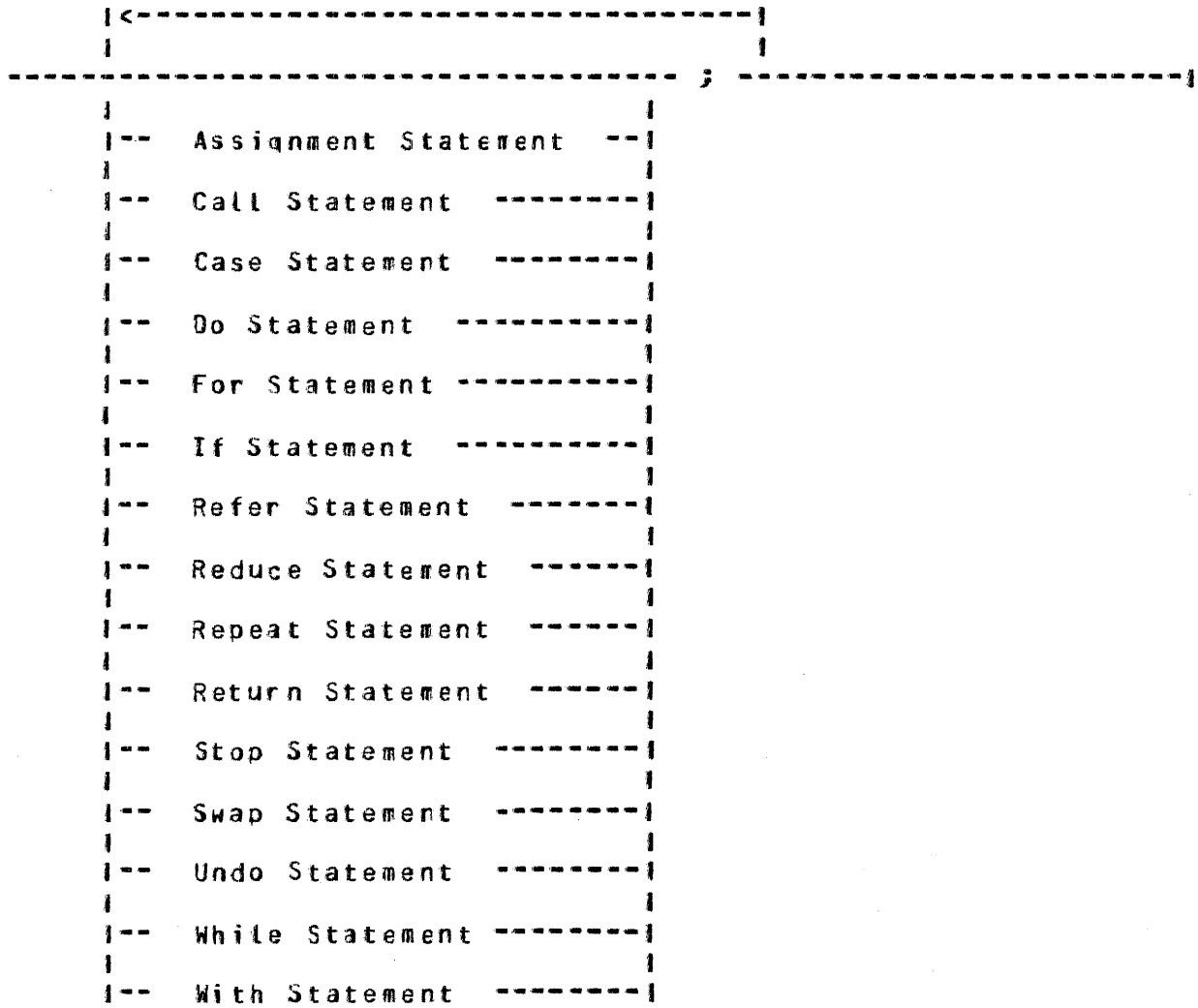
Example:

```
FORWARD PROCEDURE A(P);
  FORMAL P FIXED;

PROCEDURE B(P);
  FORMAL P FIXED;
  .
  .
  A(P);
  .
  .
END B;

PROCEDURE A(P);
  FORMAL P FIXED;
  .
  .
  B(P);
  .
  .
END A;

PROCEDURE MAX(X,Y) FIXED;
  FORMAL (X,Y) FIXED;
  IF X>Y
    THEN RETURN X;
    ELSE RETURN Y;
END MAX;
```

STATEMENTS

BODY

STATEMENT LIST



ASSIGNMENT STATEMENT

```
----- Selector -----  :=  -----  Expression  ----------------|
```

The Assignment Statement is used to store a value into a variable. If both fields are of type Character, then the move will be done from left to right with blank-filling on the right if the destination field is longer than the source field. If either of the fields is not of type Character, then the move will be done from right to left, with zero-filling on the left if the destination field is longer than the source field.

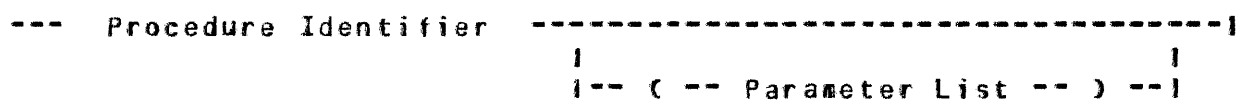
Care should be exercised when the source and destination fields overlap, since data is moved only three bytes at a time in the direction indicated by the operand types. To avoid unexpected results, the destination field of an overlapping assignment should be parenthesized, eg. SUBSTR (A, 1) := (A).

See also Put File Attribute in INPUT/OUTPUT Section.

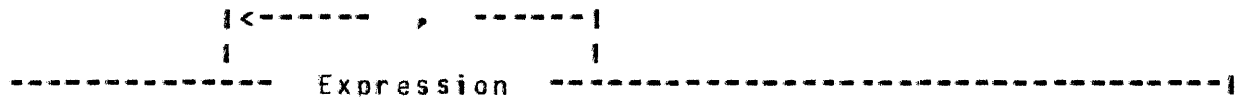
Example:

```
DECLARE
  (A, B, C)    FIXED;

A:= B + C;


DECLARE
  BIGNUM         BIT (24),
  LITTLENUM      BIT (16),
  BIGSTRING      CHARACTER (5),
  LITTLESTRING   CHARACTER (2);

BIGSTRING := "ABCDE";
LITTLESTRING := BIGSTRING;    % Sets LITTLESTRING to "AB"
BIGSTRING := LITTLESTRING;    % Sets BIGSTRING to "AB   "
BIGNUM := @7632512@;
LITTLENUM := BIGNUM;          % Sets LITTLENUM to @3251@
BIGNUM := LITTLENUM;          % Sets BIGNUM to @0032512@

BIGSTRING := " ";             % Sets BIGSTRING to "     "
BIGNUM := " ";                % Sets BIGNUM to @0000402@
BIGNUM := 0;                  % Sets BIGNUM to @0000000@
LITTLESTRING := 0;            % Sets LITTLESTRING to @0000@
```

CALL STATEMENT



Parameter List

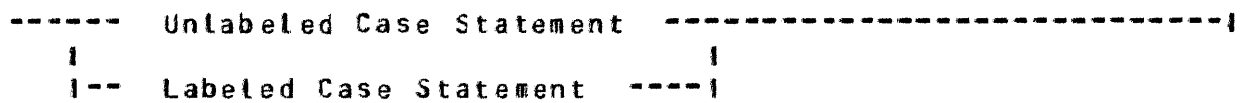


The Call Statement is used to activate an untyped procedure that has been declared with a Procedure Declaration. There must be an Expression in the Parameter List for every Formal Parameter declared.

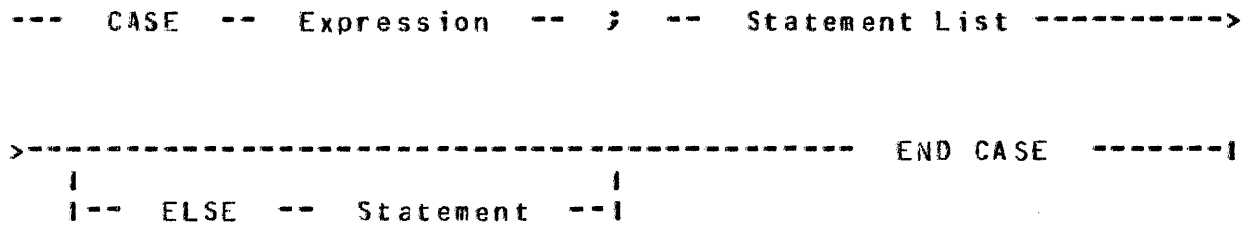
Example:

```
A:= MAX(I,J);
```

CASE STATEMENT



Unlabeled Case Statement

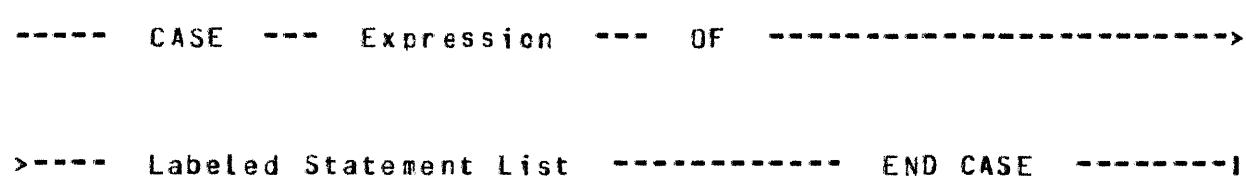


The value of Expression is used as an index into Statement List. Suppose there are N Statements in Statement List. If Expression has a value between 0 and N-1, then the corresponding Statement will be executed. If the value of Expression is out of range, then the Statement following the ELSE will be executed, or if there is no <ELSE Statement>, then a run-time error will be generated. Once the selected Statement has been executed, control is passed to the statement following <END CASE>, unless the selected Statement was an Undo or a Return.

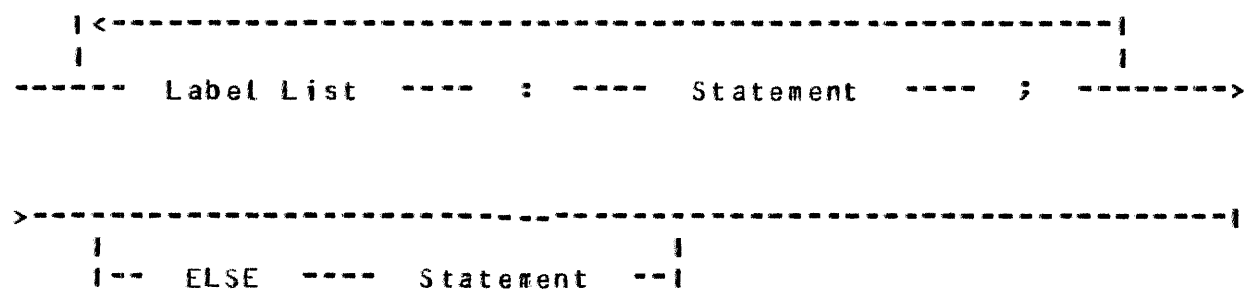
Example:

```
CASE I;
  HEX := "0";
  HEX := "1";
  HEX := "2";
  HEX := "3";
  HEX := "4";
  HEX := "5";
  HEX := "6";
  HEX := "7";
  HEX := "8";
  HEX := "9";
  HEX := "A";
  HEX := "B";
  HEX := "C";
  HEX := "D";
  HEX := "E";
  HEX := "F";
END CASE;
```

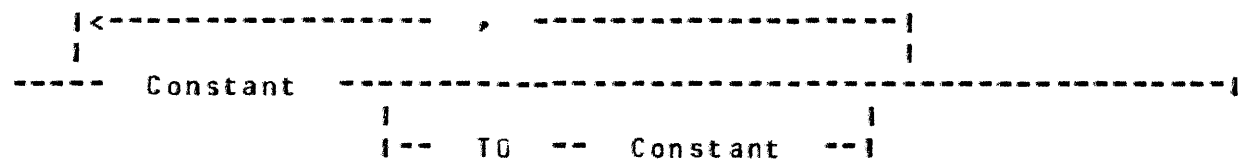
Labeled Case Statement



Labeled Statement List



Label List



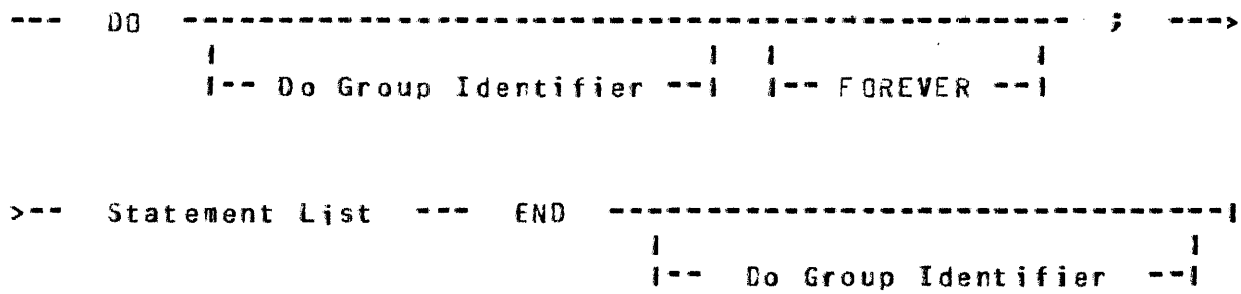
The Labeled Case Statement selects for execution that statement whose Label is equal to the value of Expression. If no statement has a Label that is equal to the value of Expression, then the Statement following the ELSE is executed, or a run-time error is generated if there is no <ELSE Statement>. After execution of the selected Statement control passes to the statement following <END CASE>, unless the selected Statement was an Undo or a Return.

Each Label must be within the range 0 to 255, and may be of any type. If TO is used in a case label, the first constant must be less than or equal to the second constant. Duplicate labels are not allowed.

Example:

```
DECLARE
  CH         CHARACTER(1),
  CH_TYPE    MEMBER OF TYPE_SET;

CASE CH OF
  "0" TO "9" :
     CH_TYPE := NUMERIC;
  ";" :
     CH_TYPE := SEMICOLON;
  "," :
     CH_TYPE := COMMA;
  "(", ")" :
     CH_TYPE := PAREN;
  ELSE
     CH_TYPE := UNKNOWN;
END CASE;
```

DO STATEMENT

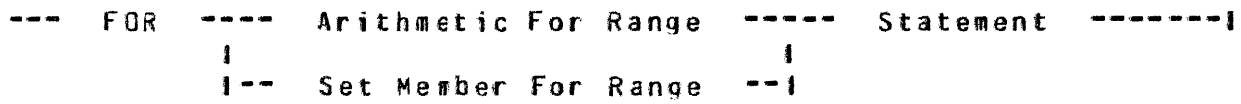


The Do Statement allows a list of statements to be grouped together. If FOREVER is present control will be passed back to the first statement from the end of the statement list. If FOREVER is not present, then control will be passed to the statement following the <END Do Group Identifier>. The Do Group can be exited with an Undo or Return statement.

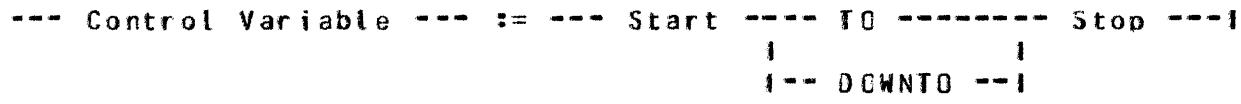
Example:

```
DECLARE
  I                FIXED,
  A (ARRAY_SIZE) FIXED;

I:= 0;
DO INIT_A FOREVER;
  IF I = ARRAY_SIZE THEN
    DO;
      I:= 0;
      UNDO INIT_A;
    END;
  A(I):= I;
  I:= I + 1;
END INIT_A;
```

FOR STATEMENT



Arithmetic For Range



The Control Variable must be a simple variable of type Fixed. Start and Stop must be Expressions, and will be coerced to be of type Fixed. Start and Stop will be evaluated and then Control Variable will be initialized to the value of Start. Statement will be executed for each value of Control Variable between (and including) Start and Stop. The Control Variable will be incremented (TO) or decremented (DOWNTO) by one after each iteration of the loop.

Example:

```
DECLARE
  I                FIXED,
  A (100)          FIXED;

FOR I:= 0 TO 4
   A(I):= I;            % will be executed 5 times

FOR I:= 99 DOWNTO 0
   A(I):= 0;            % will be executed 100 times

FOR I:= 2 TO 1
   A(I):= 1000;         % will never be executed
```

Set Member For Range

```
--- Control Variable --- IN --- Set Expression ----------------|
```

The Control Variable must be declared as a member of a set. The Set Expression can be ary legal set expression of which Control Variable is a member. For each member of Set Expression, Statement will be executed with Control Variable set to the value of the member.

Example:

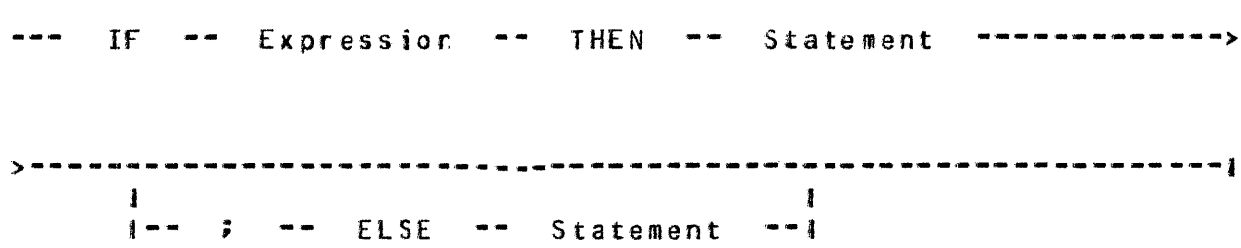
```
SET          S = APPLE, BANANA, ORANGE, PINEAPPLE;
DECLARE
  FRUIT           MEMBER OF S,
  MY_FAVORITES              S,
  YOUR_FAVORITES            S;

MY_FAVORITES := [ORANGE, BANANA];
YOUR_FAVORITES := [ORANGE, APPLE];

FOR FRUIT IN MY_FAVORITES UNION YOUR_FAVORITES
  EAT (FRUIT);                  % will eat apple, banana
                                % and orange once for each
```

Note :

Any modification of the Set Member Control Variable or the Set Expression within the loop will not alter the looping. Hence:

```
FOR FRUIT IN MY_FAVORITES DO;
  EAT (FRUIT);                  % will still eat banana
  MY_FAVORITES := [];           % and orange once for
  END;                          % each

FOR FRUIT IN MY_FAVORITES DO;   % will still eat banana
  EAT (FRUIT);                  % and orange once for
  FRUIT := APPLE;               % each
  END;
```

IF STATEMENT



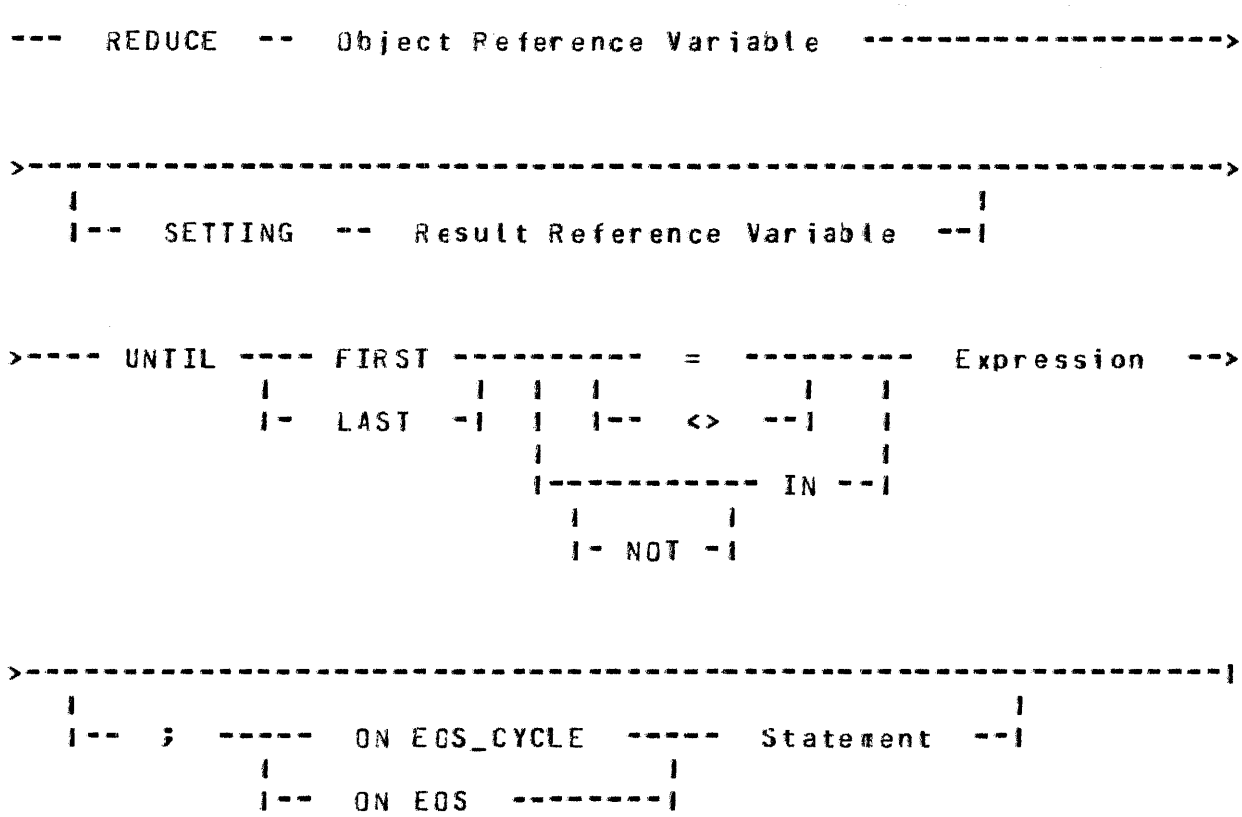
The Statement following the THEN is executed if Expression yields a true result. If Expression yields a false result then the Statement following the ELSE is executed, if the ELSE is present.

When using nested IF statements, an ELSE will be matched to the closest previously unmatched THEN.

Example:

```
DECLARE
  MAX       FIXED,
  VALUE_1   FIXED,
  VALUE_2   FIXED;

IF VALUE_1 >= VALUE_2 THEN
  MAX:= VALUE_1;
ELSE
  MAX:= VALUE_2;
```

REDUCE STATEMENT



The Reduce Statement is an efficient method of scanning character strings. The execution of a Reduce statement does not change any data; the reference variables are adjusted to point at a substring of the original string. The Reduce statement scans from left to right if FIRST is specified and from right to left if LAST is specified. After execution of the Reduce statement the Object Reference Variable is left describing the substring of the orignal Object Reference Variable that meets the condition of the reduction.

If the SETTING option is specified the Result Reference Variable will describe the substring of the original string that did not meet the condition of the reduction.

If the condition of the reduction is = or <>, the Expression must be a character string. It should be noted that only the first three characters of the Expression are compared to the string, but the length will be used to test for end of string conditions. If the condition of the reduction is IN, the Expression must be a SET of characters (See SET DECLARATION Section).

If ON EOS_CYCLE or ON EOS is specified, Statement will be executed when the length of Object Reference Variable has been reduced to zero. After completion of Statement, control will be passed to the next statement if the ON concition was EOS, or if the ON condition was EOS_CYCLE and the length of Object Reference Variable is still zero. If the ON condition was EOS_CYCLE and the Object Reference Variable is not of length zero after executing Statement, then control will be passed back to the Reduce Statement.

Example:

```
DECLARE
  NUMERICS            CHARACTER_SET
  ALPHAS              CHARACTER_SET
  SOURCE_TEXT         CHARACTER(72),
  TEXT      REFERENCE CHARACTER,
  TOKEN     REFERENCE CHARACTER;

PROCEDURE NEXT_RECORD;
  READ SOURCEFILE (SOURCE_TEXT);
  REFER TEXT TO SOURCE_TEXT;
END NEXT_RECORD;

ALPHAS := CHAR_TABLE ("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
NUMERICS := ["0" TO "9"];

REDUCE TEXT UNTIL FIRST <> " ";                                  % A
  ON EOS_CYCLE NEXT_RECORD;
IF SUBSTR (TEXT, 0, 1) IN NUMERICS THEN
  REDUCE TEXT SETTING TOKEN UNTIL FIRST NOT IN NUMERICS;         % B
ELSE IF SUBSTR (TEXT, 0, 1) IN ALPHAS THEN
  REDUCE TEXT SETTING TOKEN UNTIL FIRST NOT IN ALPHAS;           % C
ELSE DO SPECIAL_CHARACTER;
     REFER TOKEN TO SUBSTR (TEXT, 0, 1);
     REFER TEXT TO SUBSTR (TEXT, 1);
     END SPECIAL_CHARACTER;
```

Assume SOURCE_TEXT contains " IF X = 999 THEN   " and TEXT has been refered to SOURCE_TEXT. After execution of the REDUCE at line %A, TEXT will describe "IF X = 999 THEN   ". As the first character of TEXT is in the set ALPHA, the REDUCE at line %C, will be selected. After execution of this statement, TEXT will describe " X = 999 THEN   " and TOKEN will describe "IF".

REFER STATEMENT

---- REFER -- Referent -- TO -- Referee ----------

The Refer Statement is used to change the address (and possibly the length and type) of a reference variable, reference field, pointer variable, or pointer field to map the data of another variable. It is also used to assign a file to a file pointer.

If the Referent is a Reference or Pointer variable, then Referee must be an Address Generator. Only untyped or zero-length reference variables can have their length and type modified by a Refer statement. Pointers and typed reference variables will always retain the type and length from their original declaration, and may only be REFERed to objects with the same length.

If Referent is a File Pointer, then Referee must be a File.

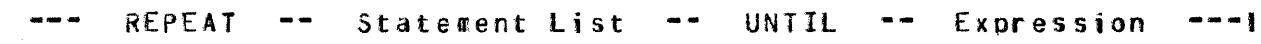
Note: REFERENCE or POINTER arrays are not allowed in the REFER statement.

Example:

```
FILE  INFILE (KIND=DISK);

DECLARE
  F       FILE_POINTER,
  TEXT    REFERENCE,
  BUFFER  CHARACTER (80);

REFER TEXT TO BUFFER;
REFER F TO INFILE;
```

Note:
REFERENCE or POINTER arrays are not allowed in the REFER statement.

REPEAT STATEMENT

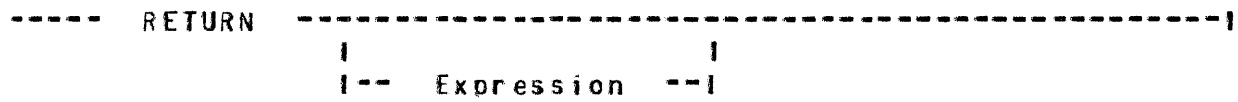


Statement List will be executed repetitively until Expression returns a true result. The exit condition is tested at the end of each loop, so Statement List will be executed at least once.

Example:

```
REPEAT

  DO_SOMETHING;
  IF END_CONDITION
    THEN DONE := TRUE;

UNTIL DONE;
```

RETURN STATEMENT

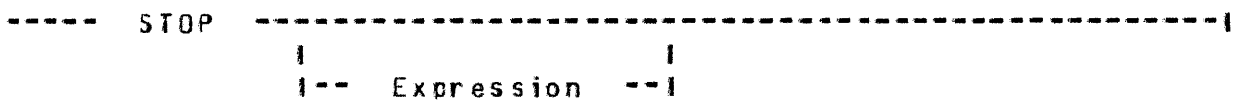


The Return Statement allows an explicit return from a procedure. If the procedure is typed then Expression must be present and will be the value returned from the procedure. If the procedure is not typed then the presence of Expression is an error. An implicit return is generated by the compiler at the end of every procedure. If the procedure is typed a value matching the procedure's type is returned (0 if Fixed, Bit, or Record, and a null string if Character).

Example:

```
PROCEDURE P;
  .
  RETURN;
  .
END P;

PROCEDURE Q FIXED;
  .
  RETURN 0;
  .
END Q;
```

STOP STATEMENT

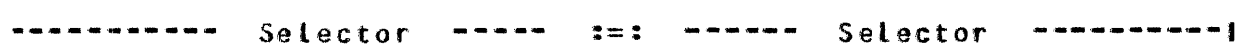


The Stop Statement will generate a terminate communicate to the MCP to end the execution of the program. Expression is intended for use by compilers to communicate the number of syntax errors to the MCP.

Example:

```
STOP;
STOP SYNTAX_ERROR_COUNT;
```

SWAP STATEMENT



The Swap Statement will exchange the values of the two fields without using an intermediate variable for temporary storage. If the fields are of unequal lengths then the shorter length will be used for both fields, and the trailing part of the longer field will remain unchanged.

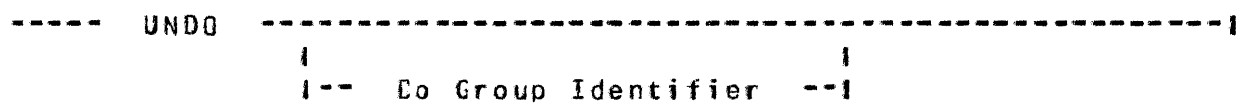
Example:

```
BUFFER_1 :=: BUFFER_2;
```

is equivalent to:

```
TEMP := BUFFER_1;
BUFFER_1 := BUFFER_2;
BUFFER_2 := TEMP;
```

UNDO STATEMENT

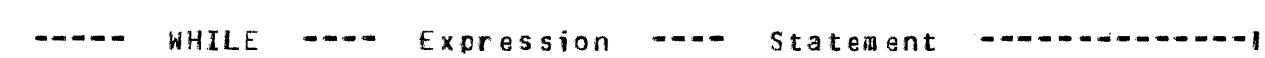


If a Do Group Identifier is specified, then the Undo Statement transfers control to the end of the Do Group by that name. If a Do Group Identifier is not specified, then control will be transferred to the end of the current Do Group.

Example:

```
DECLARE
  I                FIXED,
  A (ARRAY_SIZE) FIXED;

I:= 0;
DO INIT_A FOREVER;
  IF I = ARRAY_SIZE THEN UNDO INIT_A;
  A(I):= I;
  I:= I + 1;
END INIT_A;
```

WHILE STATEMENT

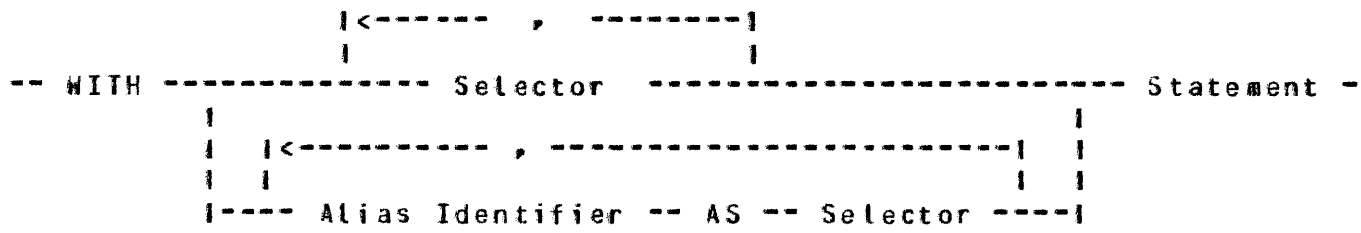


Statement will be executed repetitively while the Expression yields a true result. Expression is evaluated before each iteration, so if Expression is initially false then Statement will never be executed.

Example:

```
DECLARE
  I                FIXED,
  A (ARRAY_SIZE)   FIXED;

I:= 0;
WHILE I < ARRAY_SIZE
  DO;
  A(I):= I;
  I:= I + 1;
  END;
```

WITH STATEMENT



Selector must be a record variable. The With Statement opens a new scope containing the field indentifiers of the Record type of Selector.

Inside Statement a field of Selector is selected by specifying only its field name, without preceeding it with the entire Selector expression.

If a named field is found in more than one of the Selectors, then it is considered to belong to the last Selector specified in the With statement.

If an Alias identifier is used, it may stand for Selector within the body of the WITH block. The alias may not redefine any currently defined identifier. The alias becomes again undefined on exit of the WITH statement.

Example:

```
P(I).FIELD_1 :=X;
P(I).FIELD_2 :=Y;
P(I).FIELD_3 :=Z;
```

is equivalent to:

```
WITH P(I) DO;
  FIELD_1 :=X;
  FIELD_2 :=Y;
  FIELD_3 :=Z;
  END;
```

Example:

```
WITH A, B, C
```

is equivalent to:

```
WITH A WITH B WITH C
```

If an Alias identifier is used, it may stand for Selector within the body of the WITH block. The alias may not redefine any currently defined identifier. The alias becomes again undefined on exit of the WITH statement.

Example:

```
Job_array (This_job).My_entry_record.My_name := "Fred";
Job_array (This_job).My_entry_record.My_id := This_job_id;
Job_array (This_job).My_entry_record.My_time := Current_time;
Job_array (This_job).My_entry_record.My_id := This_job_id;
Process_entry_record (Job_array (This_job).My_entry_record);
```

is equivalent to:

```
WITH This_entry AS Job_array (This_job).My_entry_record DO;
   This_entry.My_name := "Fred";
   This_entry.My_id := This_job_id;
   This_entry.My_time := Current_time;
   Process_entry_record (This_entry);
   END;
```

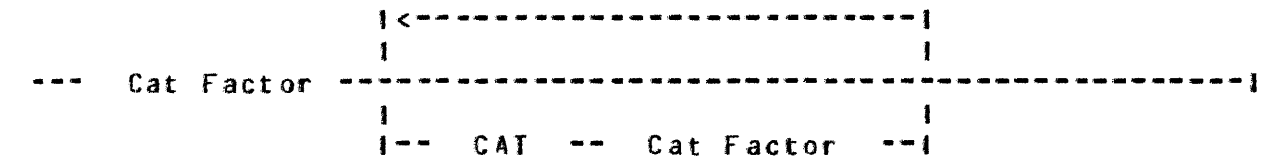
Warning: In both forms of the WITH statement, each Selector Location is evaluated once, at the point where it is named. Any subsequent change in the selection path (like changing an array index or re-referring a reference variable) will not affect the with object. Thus, in the above example, changing the value of This_job inside the body of the WITH block will have no affect on the meaning of This_entry, nor on the corresponding anonymous with object in the non-alias version.

EXPRESSIONS

Expressions are the rules for calculating a value using constants, variables, functions, and operations. The conventional rules of left to right evaluation and operator precedence are observed, as indicated by the syntax graphs. The following list summarizes the SDL2 precedence rules by listing the operators in order of decreasing precedence.

```
+  -  (unary)
*  /  MOD
+  -  (additive)
<  <=  >  >=  =  <>  IN  CONTAINS
NOT
AND  CAND  INTERSECT
OR  COR  EXOR  UNION
CAT
```

Expression

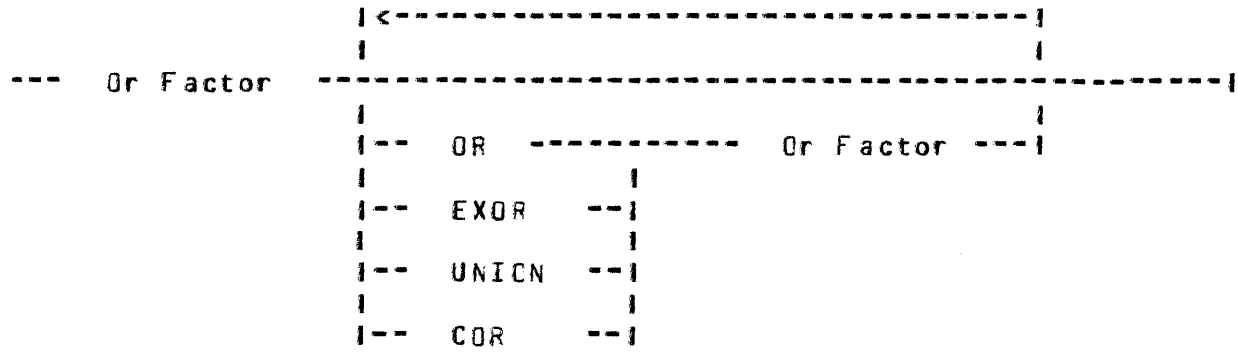


The CAT operator will concatenate two fields together as one field. The resulting type will be Character if both fields are Character; for all other combinations it will be Bit.

Example:

```
BUFFER:= DECIMAL(LINE_NUMBER,4) CAT ":" CAT TEXT CAT ":";
```

Cat Factor

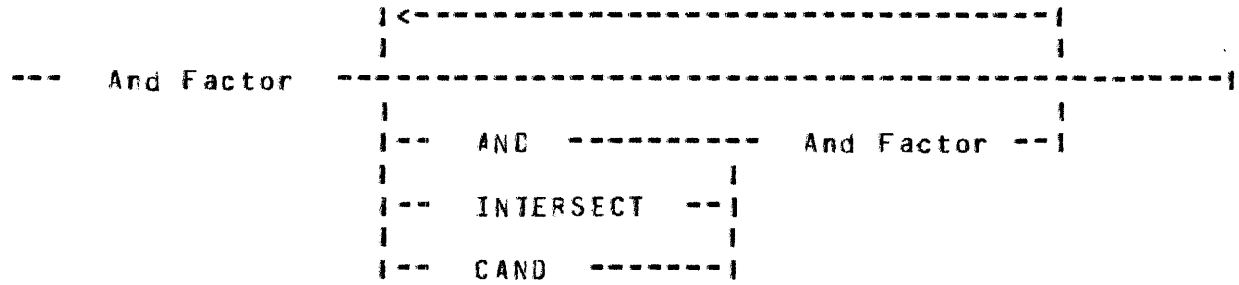


The OR/EXOR operation performs a logical bitwise OR/EXOR of two operands; the result will always be of type Bit. UNION performs a set union of two operands, and is functionally the same as an OR. COR perfomes a short-circuited OR of two operands. If the first operand is True, the result is True and the second operand is not evaluated. If the first operand is False, the result is the value of the second operand.

Example:

```
DECLARE
        (ALPHAS,
         NUMERICS,
         ALPHA_NUMERICS)  CHARACTERS_SET;

ALPHAS:= [ "0" TO "9" ];
NUMERICS:= [ "A" TO "I", "J" TO "R", "S" TO "Z" ];
ALPHA_NUMERICS:= ALPHAS UNION NUMERICS;
```

Or Factor

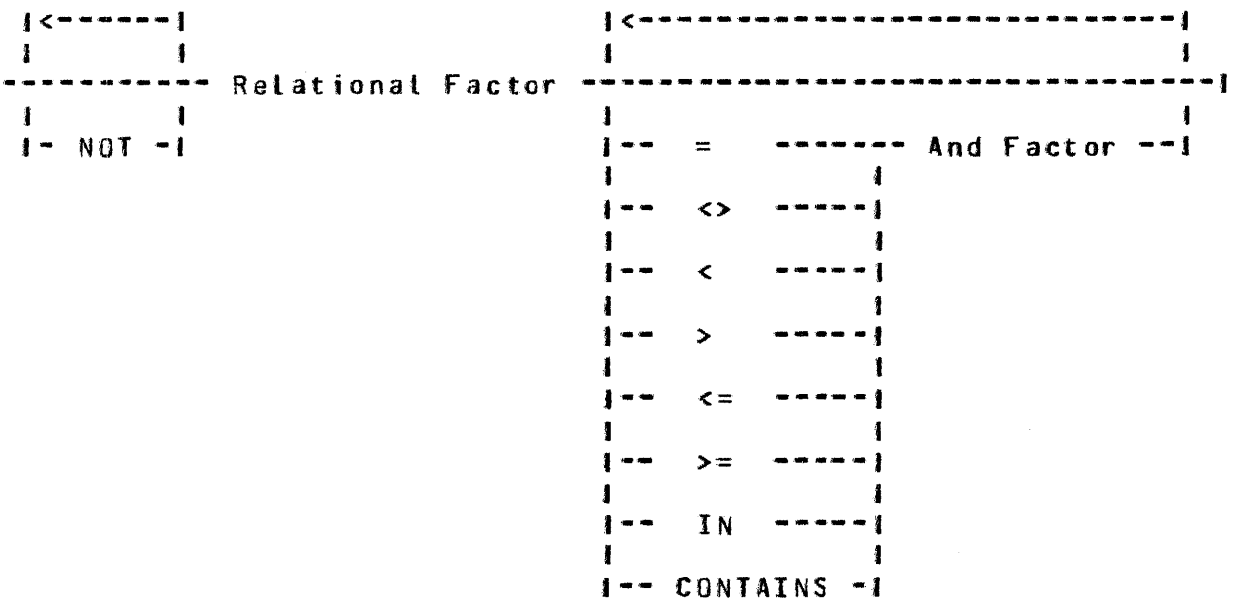


The AND operation performs a logical bitwise AND of two operands; the result will always be of type Bit. The INTERSECT operation performs a set intersection of two operands and is functionally the same as an AND. CAND performs a short-circuited AND of two operands. If the first operand is False the result is False and the second operand is not evaluated. If the first operand is True, the result is the value of the second operand.

Example:

```
SET SUITS = CLUBS, DIAMONDS, HEARTS, SPADES;

DECLARE
       (PLAYER_A, PLAYER_B, MATCHING_SUITS) SUITS
MATCHING_SUITS:= PLAYER_A INTERSECT PLAYER_B


IF I < ARRAY_BOUND (A) CAND A(I) = TOKEN
  THEN DISPLAY ("TOKEN FOUND");
```

And Factor



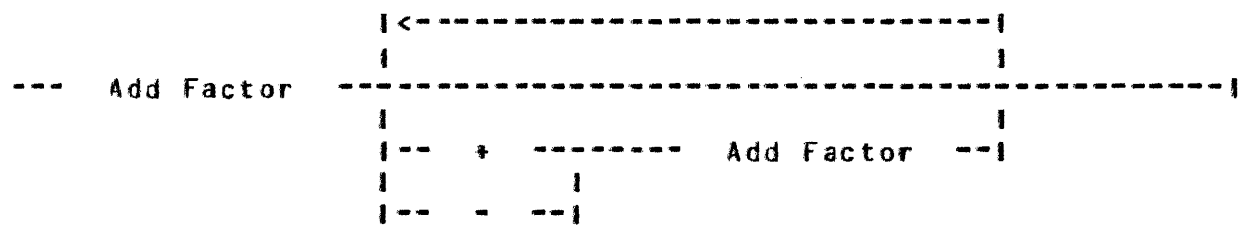
The relational operators leave a result that has a value of 1 if the cordition is true or 0 if the condition is false. The NOT operator will perform a logical NOT on one operand, and will have the same type as its single operand.

The IN operation tests whether the first factor (a set element) is a member of the second factor (a set). The CONTAINS operation tests whether the first factor (a set) contains the second factor (a set), that is, whether the second set is a subset of the first set.

Example:

```
A = B
X IN [1 TO 4]
[RED, YELLOW, GREEN, BLUE] CONTAINS [RED, BLUE]
```

Relational Factor

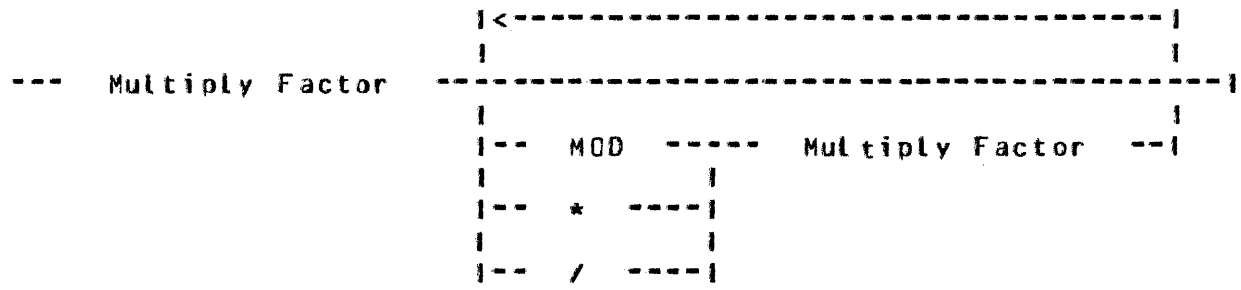


The + and - operations perform 24 bit, two's complement arithmetic on two operands. If either operand is longer than 24 bits, only the rightmost 24 bits will be used. The result will be Fixed if both operands are Fixed, otherwise it will be a Bit(24).

Example:

```
A + B
A - B
```

Add Factor

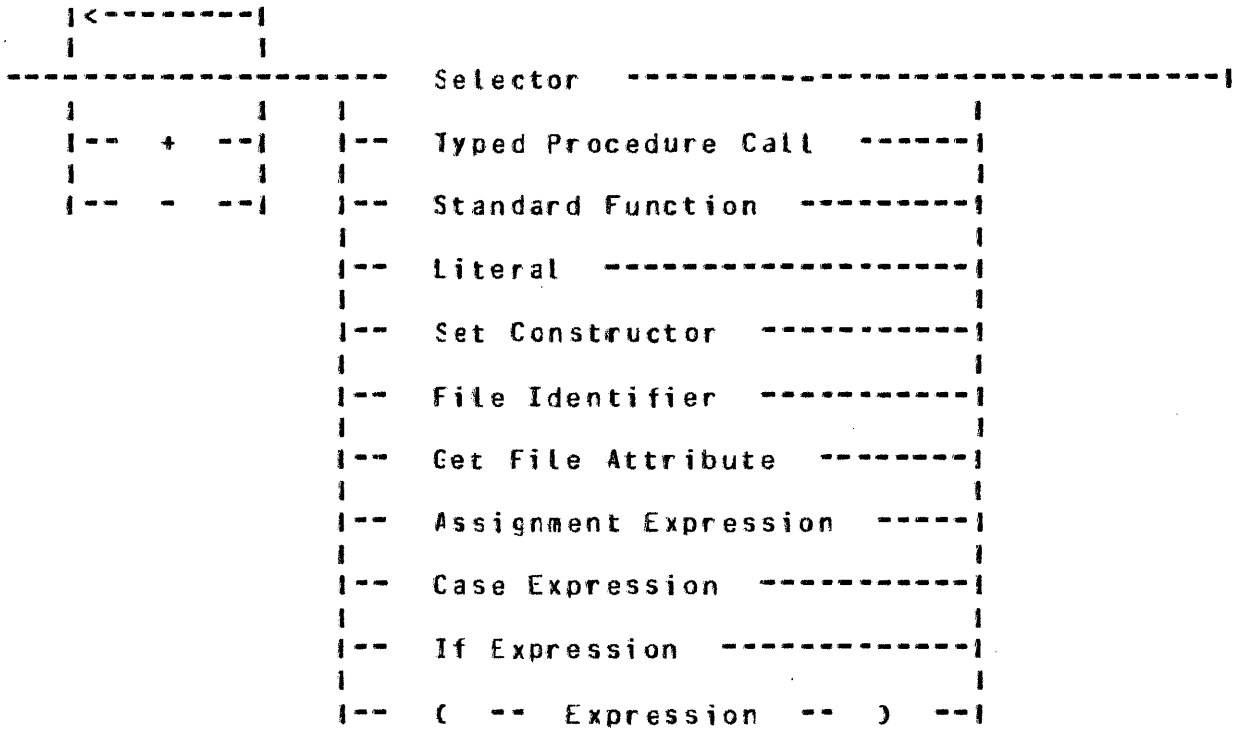


The MOD * and / operators perform 24 bit, two's complement arithmetic on two operands. The result will be Fixed if both operands are Fixed, otherwise it will be Bit(24). Division yields an integer result, ignoring the remainder (i.e. 9/4 = 2). The MOD operation is division resulting in the integer value of the remainder (i.e. 9 MOD 4 = 1).

Example:

```
A MOD B
A  *  B
A  /  B
```

Multiply Factor

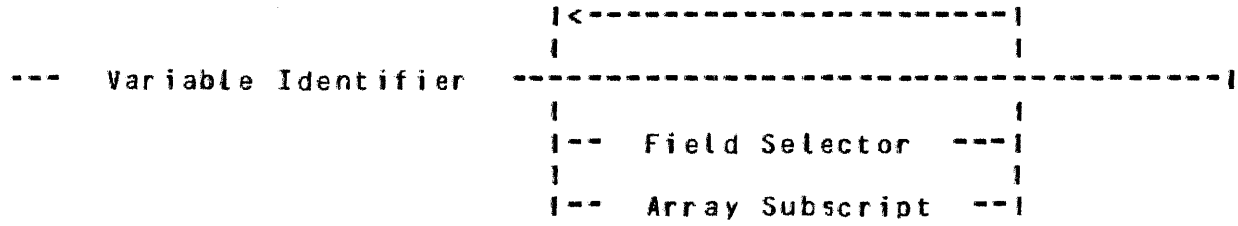


The unary - operation will negate the value of an operand. The unary + does not change the operand.

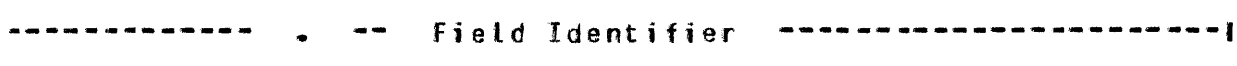
A Literal may be either a Fixed Number, a Character String, or a Bit String. See SDL2 LANGUAGE Section for details.

See INPUT/OUTPUT Section for File Identifier and Get File Attribute.

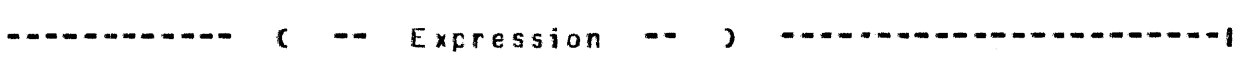
Selector



Field Selector



Array Subscript



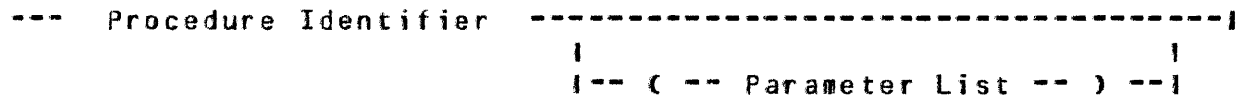
A Selector is an Address Generator which identifies a variable or a subfield of a variable.

If an Array is referenced with a subscript that is less than 0 or greater than or equal to the array bounc a run-time error will be generated.

Example:

```
RECORD R1
  LEN          FIXED,
  OFFSET       FIXED;

RECORD R2
  IDS (10)     R1,
  TYPE         FIXED;

DECLARE
  A (10)       R2;

A(I).IDS(J).LEN:= 0;
```

Typed Procedure Call

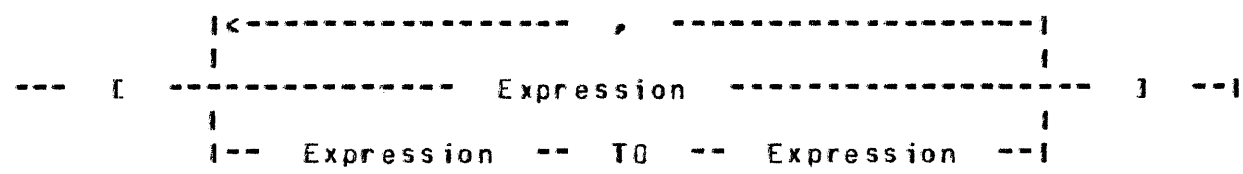


A Typed Procedure Call is used to call a procedure that returns a value, and to use that value in further Expression evaluation.

Example:

```
MAXIMUM := MAX(X,Y);
```

Set Constructor



The Set Constructor expression is used to build sets from their members. The form "Expression TO Expression" will include all members of the set bounded by the two expressions. If the first expression is greater than the second it denotes an empty set. Note that set members are declared in ascending order.

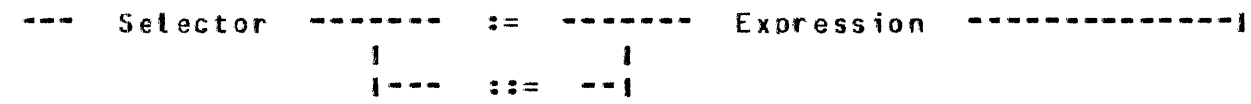
Example:

```
DECLARE
    NUMERICS            CHARACTER_SET,
    ALPHAS              CHARACTER_SET,

NUMERICS:= ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"];
ALPHAS:= ["A" TO "I", "J" TO "R", "S" TO "Z"];
```

Example:

```
SET STATES = INITIAL, MIDDLE, FINAL;

DECLARE
    STATE_SET           STATES,
    STATE               MEMBER OF STATES;

STATE_SET := [INITIAL TO FINAL];
WHILE STATE IN STATE_SET COMPUTE_STATE;
```

Assignment Expression

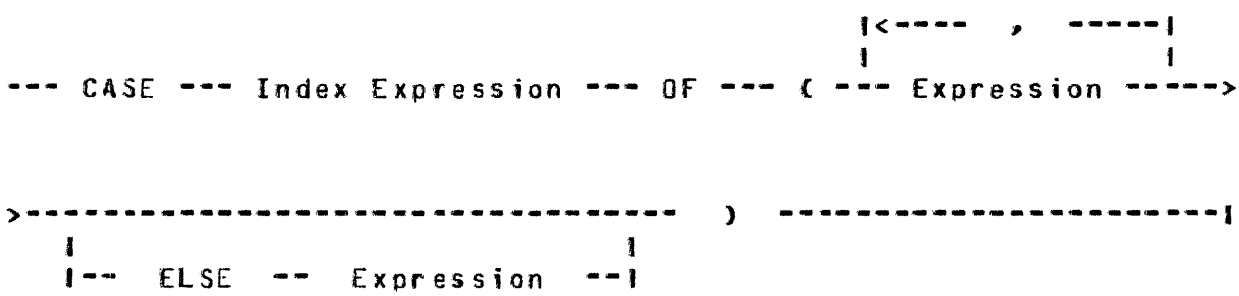


The Assignment Expression stores the value of Expression into Selector and leaves either the value (if := is used) or the Selector (if ::= is used) as an operand for further expression evaluation.

Example:

```
I:= -1;
DO FOREVER;
     A(I:= I+1) :=I;
     IF I = ARRAY_SIZE THEN UNDO;
END;
```

Case Expression

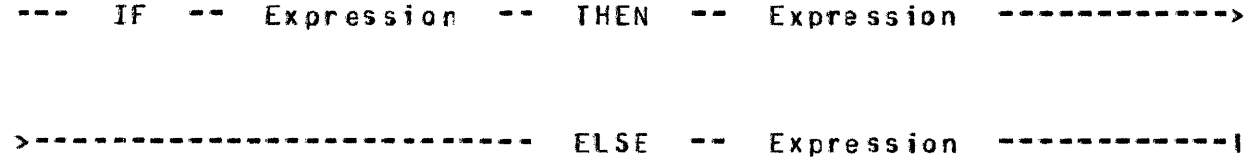


The Case Expression selects an Expression from a list of expressions to be evaluated. The value of Index Expression is used as an index into the list of Expressions. If there are N Expressions in the list the range is from 0 to N-1. If the value of Index Expression is greater than N-1, the Expression following ELSE will be used or a run-time error will be generated if <ELSE Expression> is not present.

Example:

```
I:= CASE X OF ( X+1, X+2, X+3, X+4 );

WRITE LINE (CASE I OF ("0", "1", "2", "3" ELSE "73"));
```

If Expression

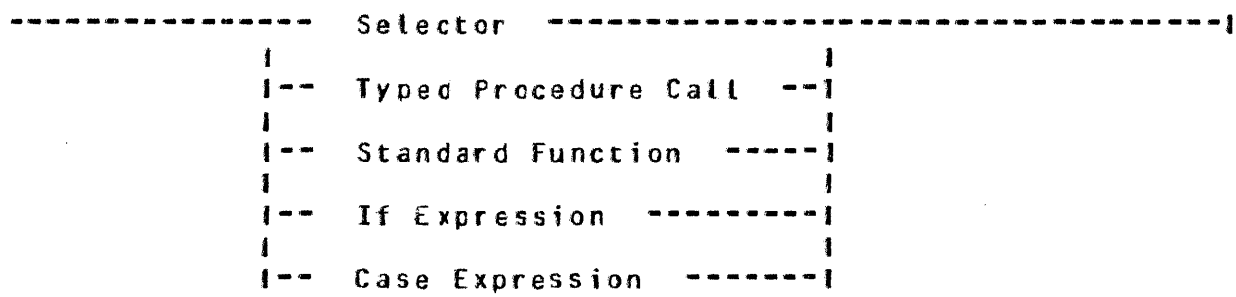


The If Expression is used to select an Expression for evaluation.

Example:

```
A := IF X<Y THEN X ELSE Y;
```

Address Generator



A Standard Function or Typed Procedure Call is considered an Address Generator if its result type is Reference.

An If Expression is considered an Address Generator if both the Expression following the THEN and the Expression following the ELSE are Address Generators. A Case Expression is considered to be an Address Generator if all elements of the Case Expression List are Address Generators. Note that the If and Case form of Address Generators are not allowed in all constructs.

STANDARD PROCEDURES AND FUNCTIONS

SDL2 provides a set of predefined procedures and functions to perform commonly required operations.

The Identifiers are not reserved and may be redeclared, in which case the user declared name takes precedence over the predefined one.

STANDARD FUNCTIONS

Allocate Memory

```
-----  ALLOCATE_MEMORY  ---  (  ---  Size  ---  )  ------------|
```

Allocates Size bits from dynamic memory, and returns the starting address as a 24 bit value. It is the programmer's responsibility to maintain an active reference to this memory address at all times. Once all active references have vanished, the data may be swapped out if the memory is needed for other purposes. Note, however, that there is no mechanism for swapping the data back in from virtual disk once it is swapped out, nor is there any mechanism for freeing the virtual disk used to store the rolled out data.

Example:

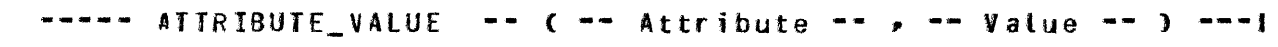
```
DECLARE  BUFFER  REFERENCE;

REFER_TYPE (BUFFER, BIT);
REFER_LENGTH (BUFFER, BUFFER_SIZE);
REFER_ADDRESS (BUFFER, ALLOCATE_MEMORY (BUFFER_SIZE));
```

Array Bound

```
-----  ARRAY_BOUND  --  (  --  Array Variable  --  )  -----------|
```

Returns the number of elements in Array Variable as a 24 bit value.

Attribute Value

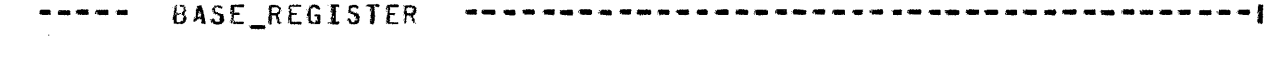


Attribute may be any file attribute allowed in a Get/Put Attribute statement. Value must be a predefined mnemonic value for that attribute. This function returns a fixed number which is the MCP-assigned number corresponding to the attribute value.

Example:

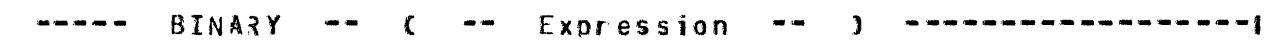
```
IF INFILE.KIND = ATTRIBUTE_VALUE(KIND,DISK)
  THEN DISPLAY ("INPUT IS FROM DISK");
```

Base Register



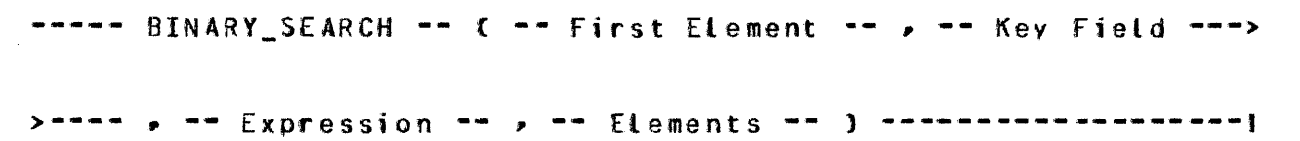
Returns the absolute memory location of the base of the program's data space as a 24 bit value.

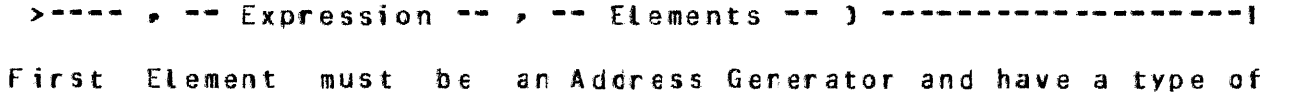
Binary



Returns a 24 bit value which is the binary representation of Expression. Expression is assumed to be a character string.

Binary Search



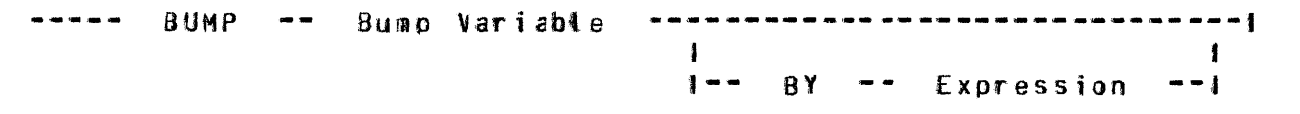


First Element must be an Address Generator and have a type of Record or Reference. It should be the first element of a structure whose elements are ordered by increasing key value. If First Element is a Record, then Key Field must be a field of the Record type of First Element that will be used to compare against Expression. If First Element is a Reference variable, then Key Field must be a Reference variable whose address is the offset of the Key Field into First Element, and whose length and type reflect the length and type of the key.

Expression is the value of the key to be searched for. The length of Expression must be equal to the length of Key Field or unpredictable results may occur. Elements is an expression whose value is used to give the bound of the search. The element number (the First Element is 0) of the first element whose Key Field is greater than or equal to Expression is returned.

This function should not be used if the elements to be searched are in more than one page of dynamic memory or are not in ascending order.

Bump



Bump Variable must be an Address Generator. If Expression is present Expression is added to Bump Variable, otherwise 1 is added to Bump Variable. The result is stored into Bump Variable and returned. (See also Bump in STANDARD PROCEDURES Section).

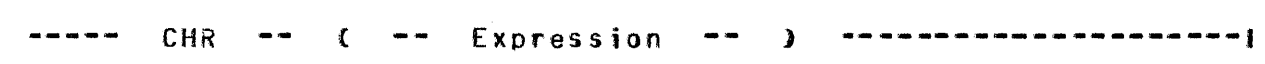
Character Table

String should be a quoted string of characters, or a bit string where every eight bits correspond to a non-graphic character. CHAR_TABLE will return a Character Set whose members are precisely those characters specified in String.

Example:

```
CHAR_TABLE ("ABCDE")        =  ["A", "B", "C", "D", "E"]
CHAR_TABLE ("XYZ", "xyz")   =  ["X", "Y", "Z", "x", "y", "z"]
```

Chr

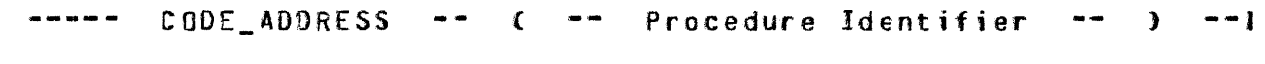


Expression should be an integer value (either bit or fixed). Returns a single character with the ordinal position of Expression in the EBCDIC character set.

Example:

```
CHR (%C1%)   = "A"
CHR (193)    = "A"
```

Code Address



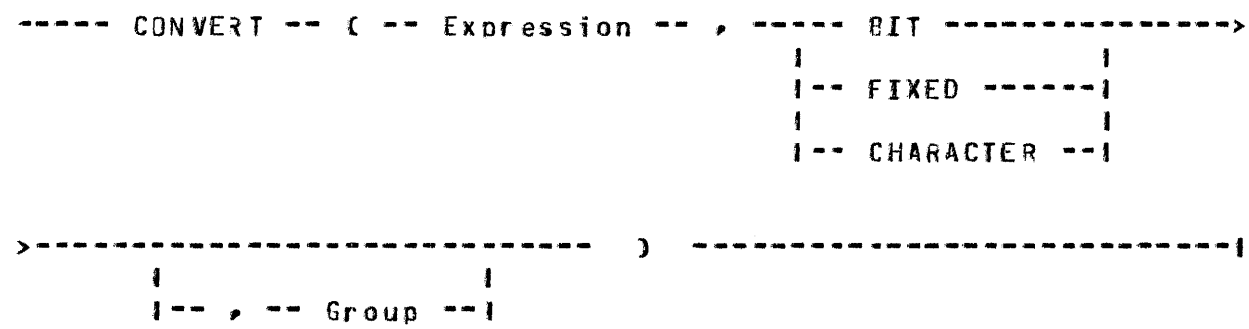
Returns the code address of Procedure Identifier as a 32 bit value. (See SDL2 S-MACHINE Product Specification for the format of a code address).

Communicate With Gismo

```
-----  COMMUNICATE_WITH_GISMO  --  (  --  Expression  --  )  --|
```

Transfers control to GISMO, passing Expression as a parameter and returning a value. See MCP Product Specification for formats of parameters and values returned. (See also Communicate With Gismo in STANDARD PROCEDURES Section). (For MCP use only).

Convert



Expesssion is converted to the type specified, according to these rules:

BIT to FIXED:	Up to 24 of the rightmost bits are converted.
FIXED to BIT:	The result is BIT(24).
CHARACTER to FIXED:	Leading blanks and sign are allowed. Up to seven of the rightmost characters are converted.
FIXED to CHARACTER:	The result is eight characters. The sign and any leading zeros are not supressed.
BIT to CHARACTER:	The radix = 2 ** Group.
CHARACTER to BIT:	The radix = 2 ** Group.

Group is only used with BIT to CHARACTER or CHARACTER to BIT conversions. It specifies the number of bits in the bit string which correspond to a character in the character string. The default Group is 4 (hexadecimal).

Example:

```
CONVERT ("-72581", FIXED)          = -72581
CONVERT (@(3)752@, CHARACTER,4)    = "1EA"
CONVERT (@(1)11011@, FIXED)        = 27
CONVERT ("132", BIT, 2)            = @(2)132@
CONVERT ("132", BIT, 4)            = @(4)132@
CONVERT ("2", BIT)                 = @(4)2@
CONVERT (-4, BIT)                  = @(4)FFFFFC@
CONVERT (-4, CHARACTER)            = "-0000004"
CONVERT (@0A0@, CHARACTER)         = "0A0"
```

Data Address

```
----- DATA_ADDRESS -- ( -- Selector -- ) ----------|
```

Returns the base relative address of Selector as a 24 bit value.

Data Length

```
----- DATA_LENGTH -- ( -- Selector -- ) ----------|
```

Returns the bit length of Selector as a 24 bit value.

Data Offset

```
----- DATA_OFFSET -- ( -- Selector -- ) ----------|
```

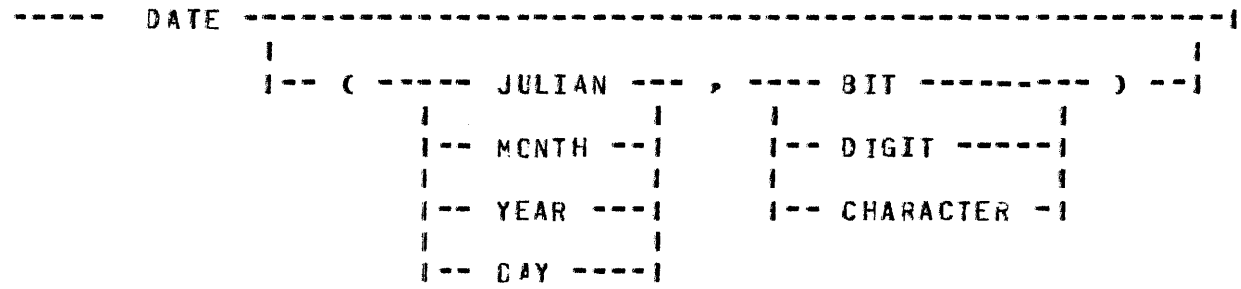
Selector must be a record subfield. Returns the offset in bits of the specified field from the beginning of the record.

Data Type

```
----- DATA_TYPE -- ( -- Selector -- ) ----------|
```

Returns the type bits of Selector, right-justified in a 24 bit field. (See SDL2 S-MACHINE Product Specification).

Date

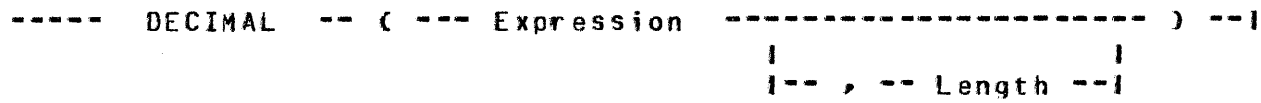


Returns a string containing the current date. The default format is (MONTH, CHARACTER).

Formats:

```
RECORD    JULIAN_FORMAT
             YEAR       BIT(7),
             DAY        BIT(9);

RECORD    MONTH_FORMAT
             MONTH      BIT(4),
             DAY        BIT(5),
             YEAR       BIT(7);

RECORD    YEAR_FORMAT
             YEAR       BIT(7),
             MONTH      BIT(4),
             DAY        BIT(5);

RECORD    DAY_FORMAT
             DAY        BIT(5),
             MONTH      BIT(4),
             YEAR       BIT(7);
```

Decimal



If Length is specified, it should be an Expression between 1 and 8 and a character string of Length characters containing the decimal representation of Expression will be returned. If Length is not specified, the result string will be from 1 to 8 characters long with leading zeroes suppressed. If Expression is greater than 24 bits, only the rightmost 24 bits will be used.

Decrement

```
----- DECREMENT -- Decrement Variable ------------------------|
                                          |                    |
                                          |- BY -- Expression -|
```

Decrement Variable must be an Address Generator. Expression (1 if BY Expression is omitted) is subtracted from Decrement Variable. The result is stored into Decrement Variable and returned. (See also Decrement in STANDARD PROCEDURES Section).

Dispatch

```
----- DISPATCH -- ( -- Port Channel -- , -- IOD Address -- ) --|
```

Port Channel should be an Expression whose value is the port and channel of the device. IOD Address should be an Expression whose value is the absolute address of the I/O descriptor to be dispatched. (See MCP Product Specification for layouts of the PortChannel field and I/O descriptors). (For MCP use only).

Dynamic Memory Base

```
-----  DYNAMIC_MEMORY_BASE ----------------------------------|
```

Returns the offset from the base register of the beginning of dynamic memory as a 24 bit value.

Fetch Communicate Message Pointer

```
-----  FETCH_COMMUNICATE_MSG_PTR ----------------------------|
```

Returns RS.REINSTATE_MSG_PTR. (See MCP Product Specification for layout of the Run Structure and meanings of the values returned).

File Resident

```
----- FILE_RESIDENT -- ( -- File Id -- ) ----------|
```

File Id must be a File, Switch File, or File Pointer, whose Title has previously been set. The file should have a hardware Kind of Disk. Returns True if the file is resident on disk, and False if it is not.

Hex Sequence Number

```
----- HEX_SEQUENCE_NUMBER ------------------------------|
```

Returns a 32 bit value containing the sequence number of the current line in hex digits.

Length

```
----- LENGTH -- ( -- Expression -- ) ----------------|
```

Returns a 24 bit value containing the length of Expression. If the type of Expression is character the value will be the number of characters, otherwise it will be the number of bits.

Limit Register

```
----- LIMIT_REGISTER ----------------------------------|
```

Returns a 24 bit value containing the base to limit size in bits of the program.

Name of Day

```
----- NAME_OF_DAY -------------------------------------|
```

Returns a 9 character string containing the name of the day of the week.

Null

```
-----  NULL  ------------------------------------------------|
```

Returns a null (zero-length) character string. Note that NULL can be used as a Selector.

Odt Input Present

```
-----  ODT_INPUT_PRESENT  -----------------------------------|
```

Returns a true value if there are any outstanding messages from an operator accept. (See 'AX' command in the MCP CONTROL SYNTAX Product Specification).

Ord

```
-----  ORD  --  (  --  Expression  --  )  --------------------|
```

Expression should be a set member. Returns the ordinal position of the member within its parent set. Note that single characters are considered to be members of any character set.

Example:

```
  ORD ("A")  = 193
  ORD (RED)  = 0   % Assuming SET FLAG_COLORS = RED, WHITE, BLUE;
```

Pack

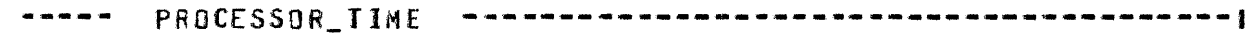
```
-----  PACK  (  --  Expression  --  )  ----------------------|
```

Expression should be an unpacked decimal expression. The zone bits will be stripped off, and the packed decimal equivalent of Expression will be returned. The result is of type Bit.

Example :

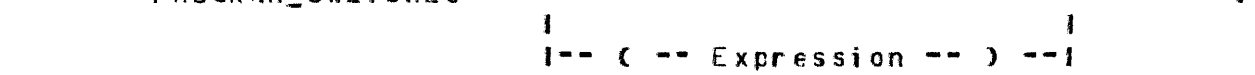
```
  PACK ("0012345")      = @00123450
  PACK (@F1F0F3@)       = @103@
```

Processor Time



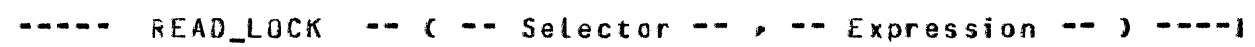
Returns a 24 bit value containing the amount of processor time that has accrued for this program in tenths of a second.

Program Switches



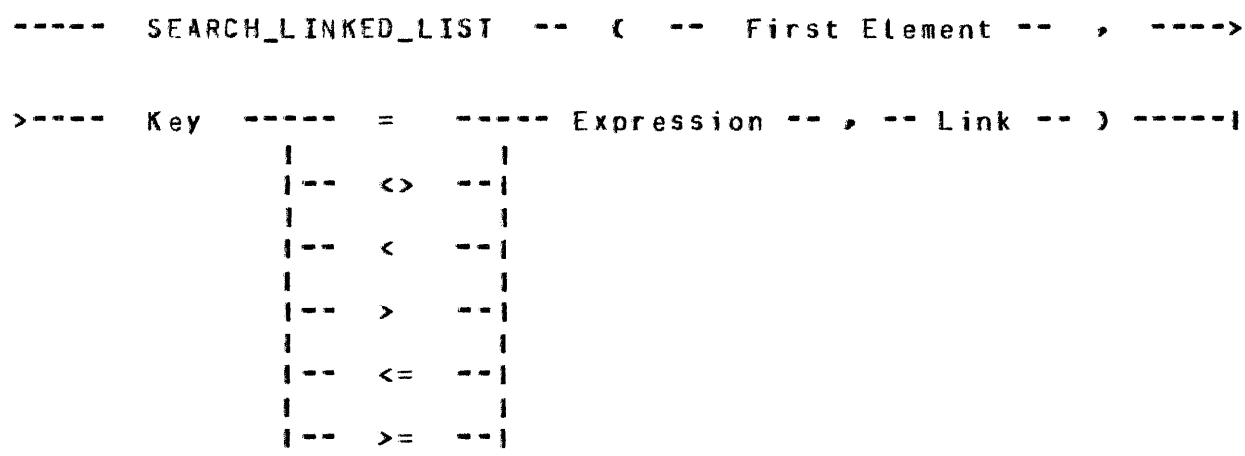
Returns a 40 bit value containing the Program Switches, unless Expression is specified. If Expression is specified, it must evaluate to a value between 0 and 9, and the function will return a 4 bit value containing that program Switch.

Read Lock



Expression is stored into Selector, and the original value of Selector is returned. If Selector is longer than 24 bits, only the rightmost 24 bits of Selector will be returned. The type of the result is always Bit.

Search Linked List



First Element must be an Address Generator with a type of Record or Reference. If First Element is a Record, then Key and Link must be fields within the Record type of the First Element. If First Element is a Reference variable, then Key and Link must be Reference variables reflecting the offsets of the key and link fields within First Element. Link must be a 24 bit field containing the base relative address of the next record in the chain. Both Key and Expression should be a maximum of 24 bits in length. If Expression is larger than 24 bits, only the rightmost 24 bits will be used in the comparision, and they will be compared against the leftmost 24 bits of the Key field.

A search is performed starting at First Element until either the relation is true, or a link of all F's is encountered, or the chain has linked back to the first element. The selected element is returned; if no match was found an Address Generator with a data address of all F's is returned.

Example:

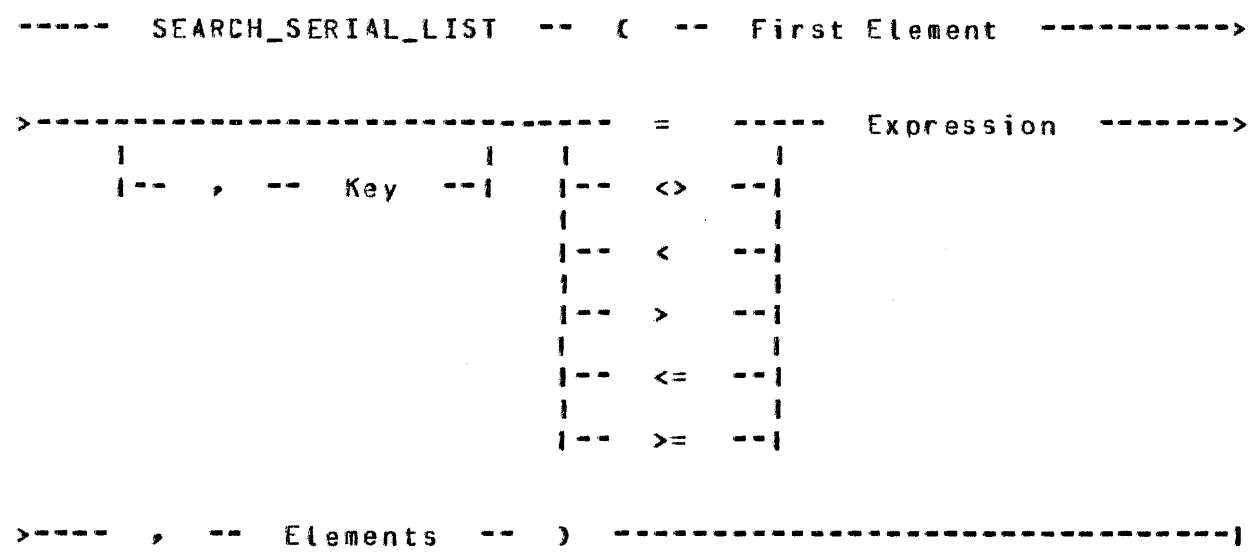
```
RECORD  INFO
          BACK_LINK       BIT(24),
          FORWARD_LINK    BIT(24),
          ID              BIT(20),
          DATA            BIT(100);

DECLARE  FIRST  INFO,
         CURRENT REFERENCE INFO;

REFER CURRENT TO
  SEARCH_LINKED_LIST (FIRST, ID = PATTERN, FORWARD_LINK)

IF DATA_ADDRESS (CURRENT) = @FFFFFF@
  THEN DISPLAY ("PATTERN NOT FOUND");
```

Search SDL Stacks

Lower must be an Expression whose value is the base relative address which is the lower bound of the range to be searched. Size must be an Expression whose value is the size of the range in bits. The descriptor stack is searched for a descriptor whose address is within the range. If a descriptor is found 1 is returned. If no descriptor is found 0 is returned.

SEARCH_SDL_STACKS is intended primarily for the use of the memory management intrinsic.

Search Serial List



First Element must be an Address Generator. If Key is specified, then First Element must be of type Record or Reference. If First Element is a Record, then Key must be a field within that Record type. If First Element is a Reference variable, then Key must be a Reference variable whose address is the offset of the key within First Element. If Key is not specified, then First Element may be of any constant length type and the complete element will be used as the key.

Elements is an Expression describing the bound of the search. A serial search is performed starting at First Element until either the relation is true or the end of the table is encountered. The element number (First Element is 0) of the selected element is returned. If no match was found Elements is returned.

This function should not be used if the elements to be searched are in more than one page of dynamic memory.

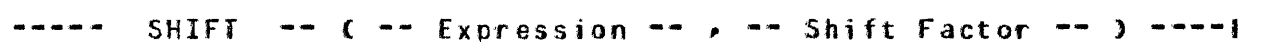
Example:

```
RECORD  JOB_INFO
          JOB_NO   BIT(16)
          NAME     CHARACTER(30)
          PRIORITY BIT(4);
DECLARE JOB(MAX_JOBS)   JOB_INFO,
        I               FIXED;

I := SEARCH_SERIAL_LIST (JOB(0), JOB_NO = THIS_JOB, MAX_JOBS);

IF I = MAX_JOBS
   THEN DISPLAY ("JOB NOT FOUND");
```

Sequence Number

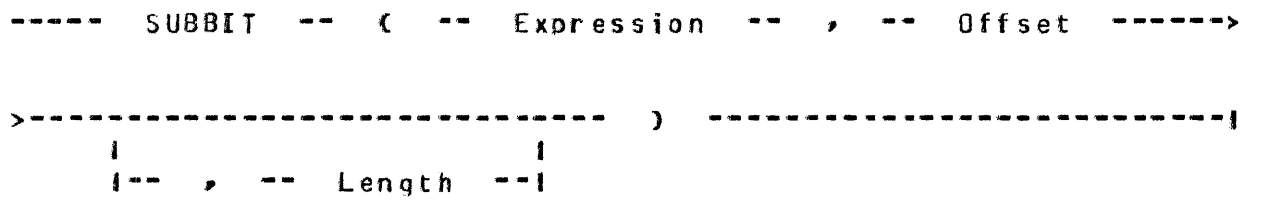
Returns an 8 character string containing the value of the sequence number of the current source line.

Shift



Returns Expression shifted the number of bits specified by Shift Factor. If Shift Factor is positive, Expression will be shifted to the left by concatenating Shift Factor bits of zero onto the right. If Shift Factor is negative, Expression will be shifted to the right by truncating the rightmost Shift Factor bits. For a bit value, this is equivalent to multiplying by 2 raised to the Shift Factor power.

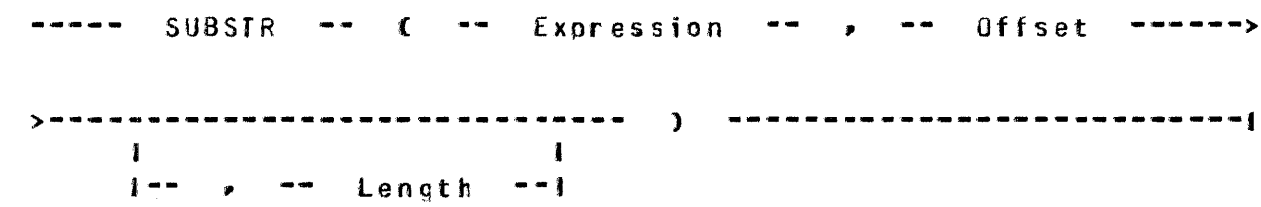
Subbit



Expression is assumed to be of type Bit. Returns a subfield of the value of Expression starting at Offset bits for Length bits. If Length is not present the length is calculated to be the remaining bits. Bits are numbered from left to right, starting with zero. The result will always have a type of Bit.

(SUBBIT may be used as an Address Generator, in which case Expression must be an Address Generator).

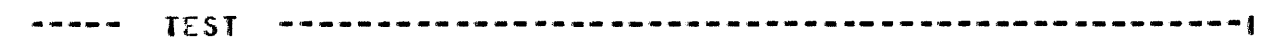
Suostr



Expression is assumed to be of type Character. Returns a subfield of the value of Expression starting at Offset characters for Length characters. If Length is not present the length is calculated to be the remaining characters. The result will always have a type of Character.

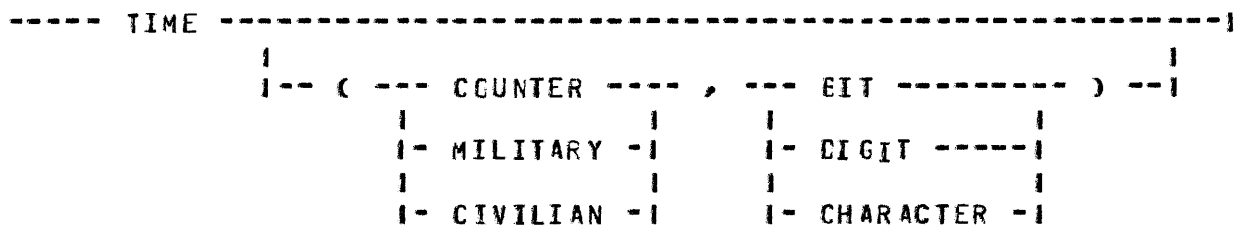
(SUBSTR may be used as an Address Generator, in which case Expression must be an Address Generator).

Test



For compiler and interpreter debugging only. (See also STANDARD PROCEDURES Section).

Time



Returns a string containing the current time of day. The default format is (CIVILIAN, CHARACTER).

Formats:

```
RECORD    COUNTER_FORMAT
             TENTHS    BIT(20);

RECORD    MILITARY_FORMAT
             HOURS     BIT(5),          % 24-HOUR CLOCK
             MINUTES   BIT(6),
             SECONDS   BIT(6),
             TENTHS    BIT(4);

RECORD    CIVILIAN_FORMAT
             HOURS     BIT(4),          % 12-HOUR CLOCK
             MINUTES   BIT(6),
             SECONDS   BIT(6),
             TENTHS    BIT(4),
             AM_OR_PM  CHARACTER(2);  % "AM" OR "PM"
```

Timer

Returns a 24 bit value with the current setting of the hardware Time Register.

Todays_Date

```
-----  TODAYS_DATE  ------------------------------------------|
```

Returns a 17 character string containing the date of compilation of this program.

The format of the string is:

```
RECORD TODAYS_DATE_FORM
MONTH               CHARACTER(2), % 01 TO 12
SLASH_1             CHARACTER(1), % LITERALLY "/"
DAY                 CHARACTER(2), % 01 TO 31
SLASH_2             CHARACTER(1), % LITERALLY "/"
YEAR                CHARACTER(2), % 00 TO 99
FILLER              CHARACTER(1), % LITERALLY " "
HOUR                CHARACTER(2), % 01 TO 12
COLON_1             CHARACTER(1), % LITERALLY ":"
MINUTE              CHARACTER(2), % 00 TO 59
FILLER              CHARACTER(1), % LITERALLY " "
AM_PM               CHARACTER(2); % LITERALLY "AM" OR "PM"
```

Type_Length

```
-----  TYPE_LENGTH  --  (  --  Type Identifier  --  )  --------|
```

Returns a constant value which is the length in bits of any variable of the given type.

Unpack

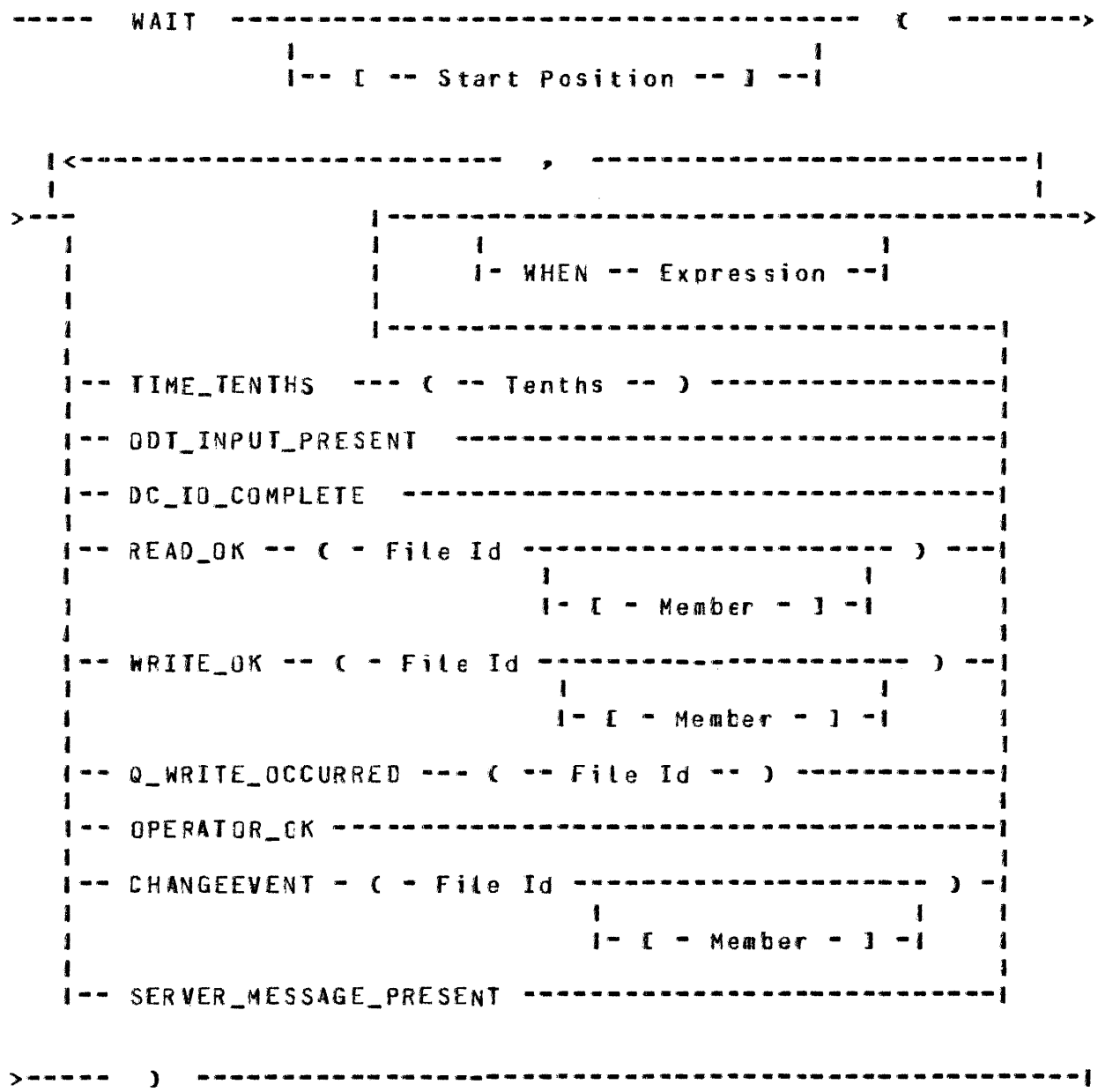
```
-----  UNPACK  --  (  --  Expression  --  )  -----------------|
```

Expression should be a packed decimal expression. The unpacked equivalent of Expression, with the zone bits filled in, will be returned. The result is of type Character.

Example :

```
UNPACK (@12345@)          = "12345"
UNPACK (@100@)            = "100"
```

Wait



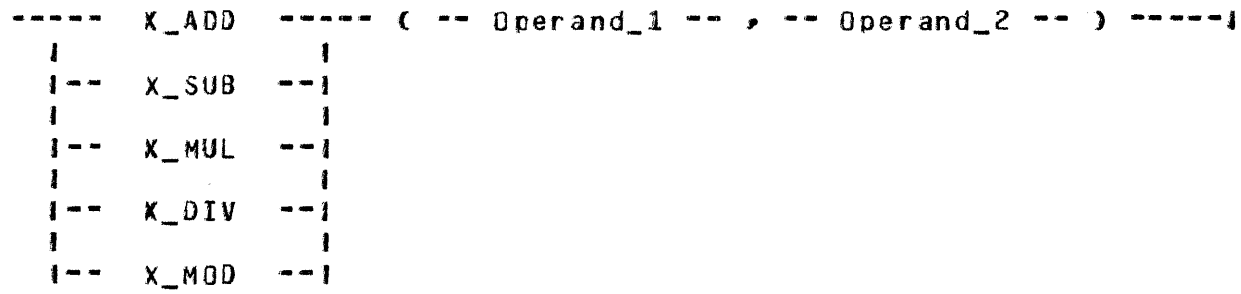
The program will be suspenced until one of the wait events becomes true. The value returned is the ordinal position of the event in the wait event list (first is 0). Start Postion can be used to specify which event in the list to test first. The default is to start at event 0.

If WHEN Expression is present the event will be included in the wait list only if expression yields a true result, however a null event will be inserted in the list to maintain the ordinal

position of the events.

If TIME_TENTHS is specified, it must be the first event in the list.

Extended Arithmetics



The extended arithmetic functions perform extended precision arithmetic, where either the operands or the result are longer than 24 bits. No sign analysis is performed. If both operands are not the same length unpredictable results may occur.

X_ADD adds Operand_1 to Operand_2 and returns the result as type Bit.

X_SUB subtracts Operand_2 from Operand_1 and returns the result as type Bit.

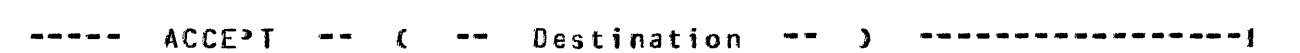
X_MUL multiplies Operand_1 by Operand_2 and returns the result as type Bit.

X_DIV divides Operand_1 by Operand_2 and returns the result as type Bit.

X_MOD modulo divides Operand_1 by Operand_2 and returns the result as type Bit.

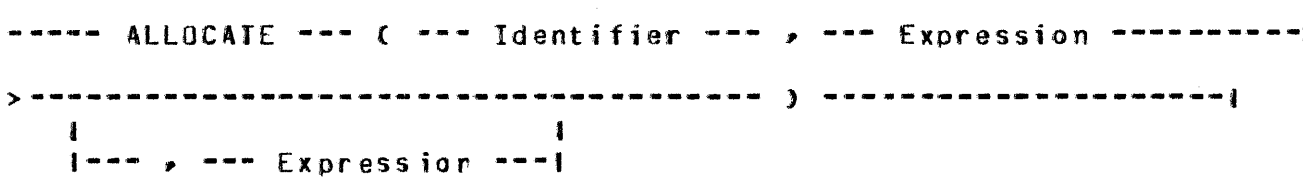
STANDARD PROCEDURES

Accept



Destination must be an Address Generator. The program will be suspended until the operator responds with an AX message on the OCT. The message entered by the operator will be read into Destination, and then program execution will be resumed.

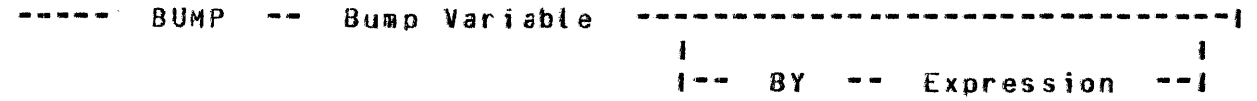
Allocate



The ALLOCATE procedure must be called from global (un-nested) code. Identifier must be a global dynamic variable. If this variable is an array with both varying bound and varying length, then both Expression arguments must be present, signifying the actual array bound and element size desired, in that order. Otherwise, a single Expression is supplied, indicating either the length or the bound, as the case may be. For example,

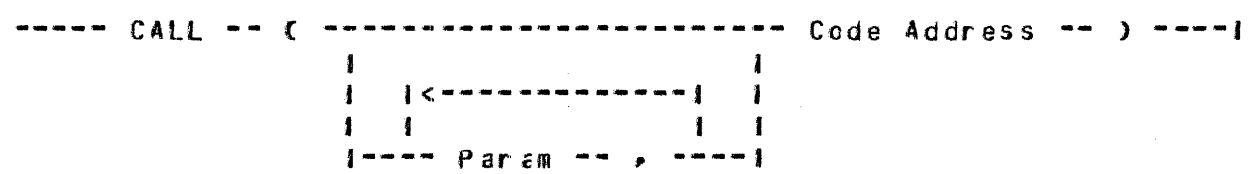
```
DECLARE gda        (*) BIT,
        buf        BIT VARYING,
        recsize    BIT(8),
        total_size BIT(24);

  % Calculate recsize and total_size here....

  ALLOCATE (buf, recsize);
  ALLOCATE (gda, total_size/recsize, recsize);

  FOR i := 0 TO ARRAY_BOUND (gda) - 1 DO;
       get_buf (buf);
       gda (i) := buf;
       END;
```

Bump



Bump Variable must be an Address Generator. If Expression is present Expression is added to Bump Variable, otherwise 1 is added to Bump Variable. The result is stored into Bump Variable. (See also Bump in STANDARD FUNCTIONS Section).

Call

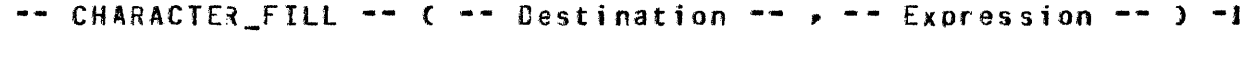


Performs an indirect call. Code Address must generate a 32 bit value containing the code address to be called (See SDL2 S-MACHINE Product Specification for format of a code address).

All parameters will be passed by reference, not by value. Array parameters are not allowed.

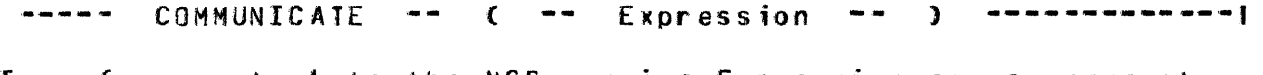
(For DMS use only).

Character Fill



Destination must be an Address Generator. The leftmost 8 bits (first byte) of Expression will be used to fill Destination.

Communicate



Transfers control to the MCP passing Expression as a parameter. (See MCP COMMUNICATES AND STRUCTURES Product Specification for format of parameters in a communicate).

Communicate With Gismo

```
-----  COMMUNICATE_WITH_GISMO  --  (  --  Expression  --  )  --|
```

Transfers control to GISMO passing Expression as a parameter. (See MCP Product Specification for formats of parameters. See also Communicate With Gismo in STANDARD FUNCTIONS Section). (For MCP use only).

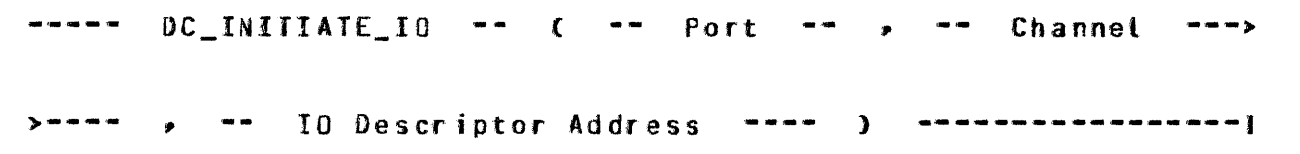
Compile Card Information

```
-----  COMPILE_CARD_INFO  --  (  --  Destination  --  )  ------|
```

Destination must be an Address Generator. Information about the program is returned into Destination in the following format:

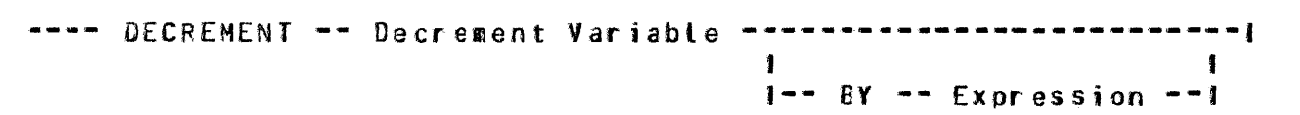
```
RECORD COMPILE_CARD_INFO_FORM
  OBJECT NAME                                  CHARACTER(30),
  EXECUTE TYPE                                 CHARACTER(2),
        % 01 = execute
        % 02 = compile and go
        % 03 = compile for syntax
        % 04 = compile to library
        % 05 = compile and save
        % 06 = go part of compile and go
        % 07 = go part of compile and save
  COMPILER PACK IDENTIFIER                     CHARACTER(10),
  INTERPRETER NAME                             CHARACTER(30),
  INTRINSIC NAME                               CHARACTER(10),
  PRIORITY                                     CHARACTER(2),
  SESSION NUMBER                               CHARACTER(6),
  JOB NUMBER                                   CHARACTER(6),
  COMPILER MFID AND FIO                        CHARACTER(20),
  CHARGE NUMBER                                CHARACTER(7),
  FILLER                                       CHARACTER(1),
  DATE AND TIME COMPILED                       BIT(36),
  FILLER                                       BIT(4),
  USERCODE                                     CHARACTER(10),
  PASSWORD                                     CHARACTER(10),
  PARENT JOB NUMBER                            CHARACTER(4),
  PARENT QUEUE IDENTIFIER                      CHARACTER(20),
  LOG SPO                                      CHARACTER(1);
```

Datacomm Initiate I/O



IO Descriptor Address must be the base-relative address of an IO descriptor which will be initiated on the specified Port and Channel. (See MCP Product Specification for more information).

Decrement

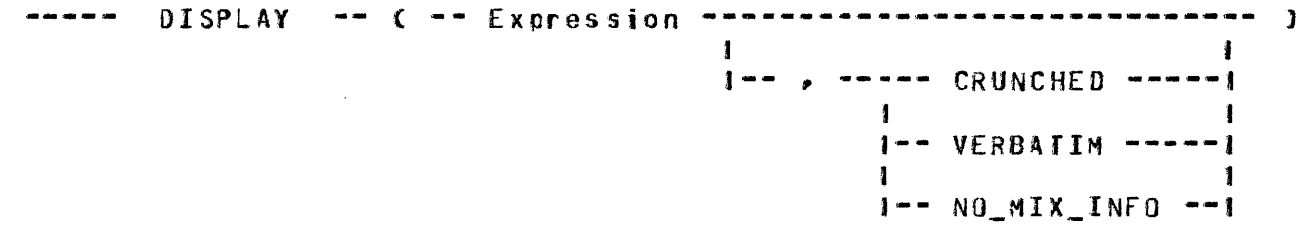


Decrement Variable must be an Address Generator. Expression (1 if BY Expression is omitted) is subtracted from Decrement Variable and the result is stored into Decrement Variable. (See also Decrement in STANDARD FUNCTIONS Section).

Disable Interrupts

All interrupts will be disabled until an ENABLE_INTERRUPTS is executed. (For MCP use only).

Display



Expression will be printed on the ODT. If CRUNCHED is specified, the MCP will delete all trailing blanks and substitute one blank for each occurrence of multiple embedded blanks in Expression. If VERBATIM is specified, mix information will appear on a separate line of output, and the expression will appear on the

ODT exactly as given. NO_MIX_INFO is like VERBATIM but no mix information is printed with the output. If VERBATIM or NO_MIX_INFO is specified, the leftmost column of output should contain only blanks or special terminal highlighting characters.

Dump

```
----- DUMP ------------------------------------------------|
```

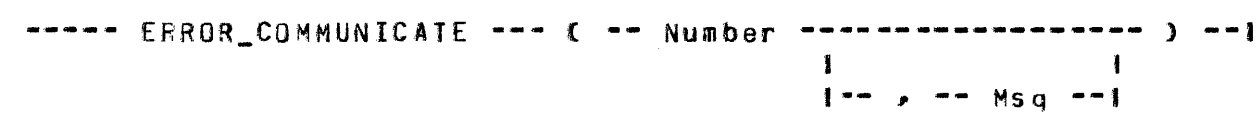
Generates a memory dump of the program.

Enable Interrupts

```
----- ENABLE_INTERRUPTS -----------------------------------|
```

If interrupts are disabled they will be enabled and normal processing of interrupts will continue. (For MCP use only).

Error Communicate



Transfers control to the MCP, using Number to select the appropriate run-time error message (number 29 is the null error message). If Msg is present, then it will be displayed following the canned error message text.

Fetch

```
----- FETCH -- ( -- IO Reference -- , ----------------->

>---- Result Descriptor -- , -- Port Channel -- ) -----|
```

IO Reference must be an Expression whose value is the reference address of an I/O descriptor. Result Descriptor must be an Address Generator. Port Channel must be an Address Generator.

The Standard Procedure FETCH fetches the result of an I/O operation. If there is a high priority interrupt, then that interrupt will be reported. Otherwise, if IO Reference is non-zero then only an interrupt on an I/O descriptor with that reference address will be reported. If not found then the first interrupt to occur will be reported. The port and channel number will be stored into Port Channel and the address of the result descriptor will be stored into Result Descriptor.

(See MCP Product Specification for more information on interrupts and I/O descriptors). (For MCP use only).

Freeze Program

```
----- FREEZE_PROGRAM ----------------------------------------|
```

This procedure prevents the program from being moved to a different memory location, or from being rolled out of memory.

Halt

```
----- HALT -- ( -- Expression -- ) -------------------|
```

The machine will halt with 0113 displayed in the L register and the low order 24 bits of Expression displayed in the T register. (See SDL2 S-MACHINE Product Specification for more information).

Hardware Monitor

```
----- HARDWARE_MONITOR -- ( -- Expression -- ) --------|
```

See SDL2 S-MACHINE Product Specification or B1000 PROCESSOR Product Specification for more information.

Message Count

```
----- MESSAGE_COUNT -- ( -- File Id -- , -- Destination -- ) --|
```

File Id must be the name of a queue file, and Destination must be an Address Generator. The number of messages in the queue will be returned as a fixed number into Destination. If File Id refers to a queue file family then an array of values, one for each family member, will be returned into Destination.

Refer Address

```
----- REFER_ADDRESS -- ( -- Referent -- , -- Expression -- ) --|
```

Referent must be a Reference or Pointer Variable. Expression will be stored into the address part of the descriptor for Referent.

Refer Bound

```
--- REFER_BOUND --- ( --- Referent --- , --- Expression --- ) ---|
```

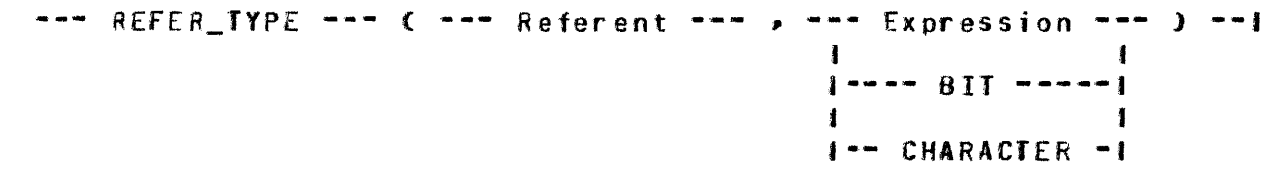
Referent must be a REFERENCE array. Expression will be stored into the array-bound part of the descriptor for Referent.

Refer Length

```
----- REFER_LENGTH -- ( -- Referent -- , -- Expression -- ) ---|
```

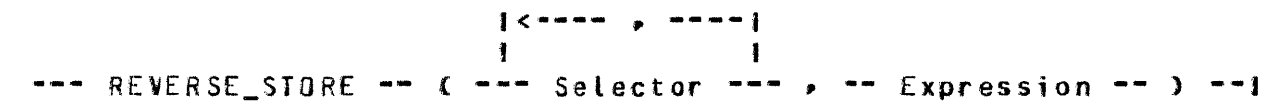
Referent must be an untyped Reference variable. Expression will be stored into the length part of the descriptor for Referent.

Refer Type



Referent must be an untyped Reference variable. The type part of the descriptor for Referent will be set to either Expression, the predefined type of BIT, or the predefined type of CHARACTER.

Reverse Store



Multiple store operations will be generated, evaluated from left to right.

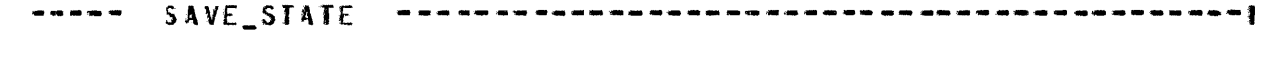
Example:

```
REVERSE_STORE(A,B,C,1);
```

is the same as

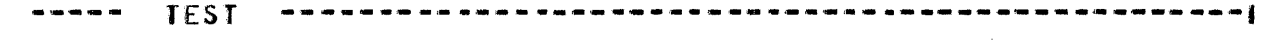
```
A:= B; B:= C; C:= 1;
```

Save State



The current state of the interpreter will be saved in the Run Structure Nucleus. (For MCP use only).

Test

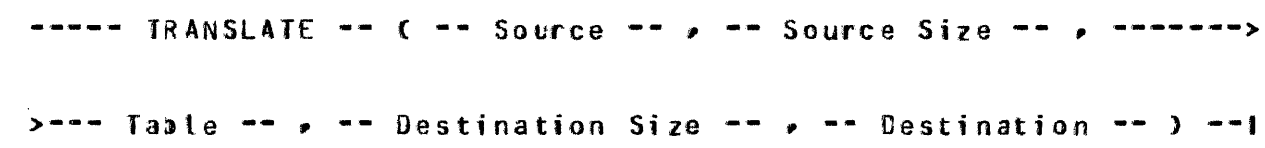


For compiler and interpreter debugging only. (See also Test in STANDARD FUNCTIONS Section).

Thaw Program

This procedure unfreezes the program, and allows it to be moved or rolled out of memory. It does not force the program to be rolled out.

Translate



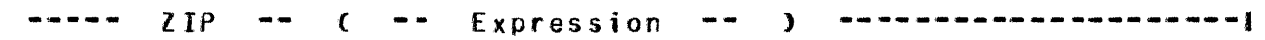
Source and Destination must be Address Generators. Source Size, Table, and Destination Size are Expressions. Source is assumed to consist of items of Source Size (in bits). Table and Destination are assumed to consist of items of Destination Size (in bits). Each item in Source is used as a subscript into Table to obtain an item which is placed into Destination in the same ordinal position as the source.

Example:

```
DECLARE
  BITSTRING       BIT (36),
  CHARSTRING      CHARACTER (9),
  TRANS_TABLE     BIT (16 * 8);

TRANS_TABLE := "0123456789ABCDEF";
BITSTRING := @F206541120;
TRANSLATE (BITSTRING, 4, TRANS_TABLE, 8, CHARSTRING);
DISPLAY ("BITSTRING = @" CAT CHARSTRING CAT "@");
```

The above example converts a bit string into the equivalent hexadecimal character string for display purposes.

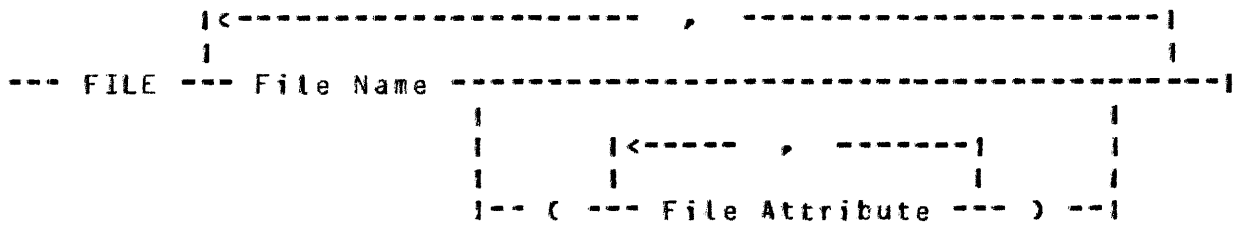
Zip



The Zip statement allows the program to pass control instructions to the MCP. Expression should generate a character string whose value is a valid MCP control command as documented in the MCP CONTROL SYNTAX Product Specification.

INPUT/OUTPUT

FILE DECLARATION



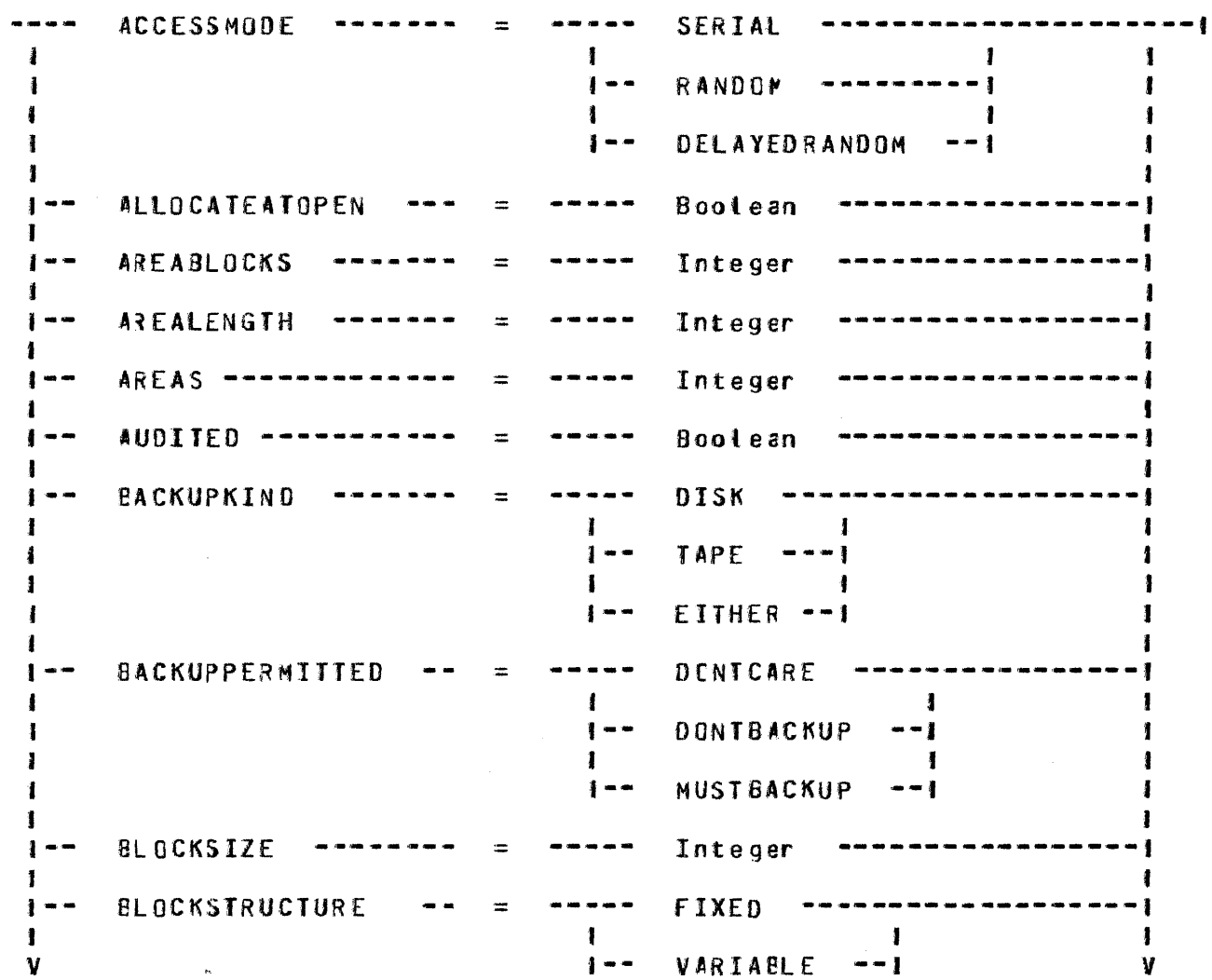
A File Declaration will generate an FPB with the default file attributes modified by the attribute list. The File Name, when used in an Expression, will yield its FPB number (the first file declared will be 0).

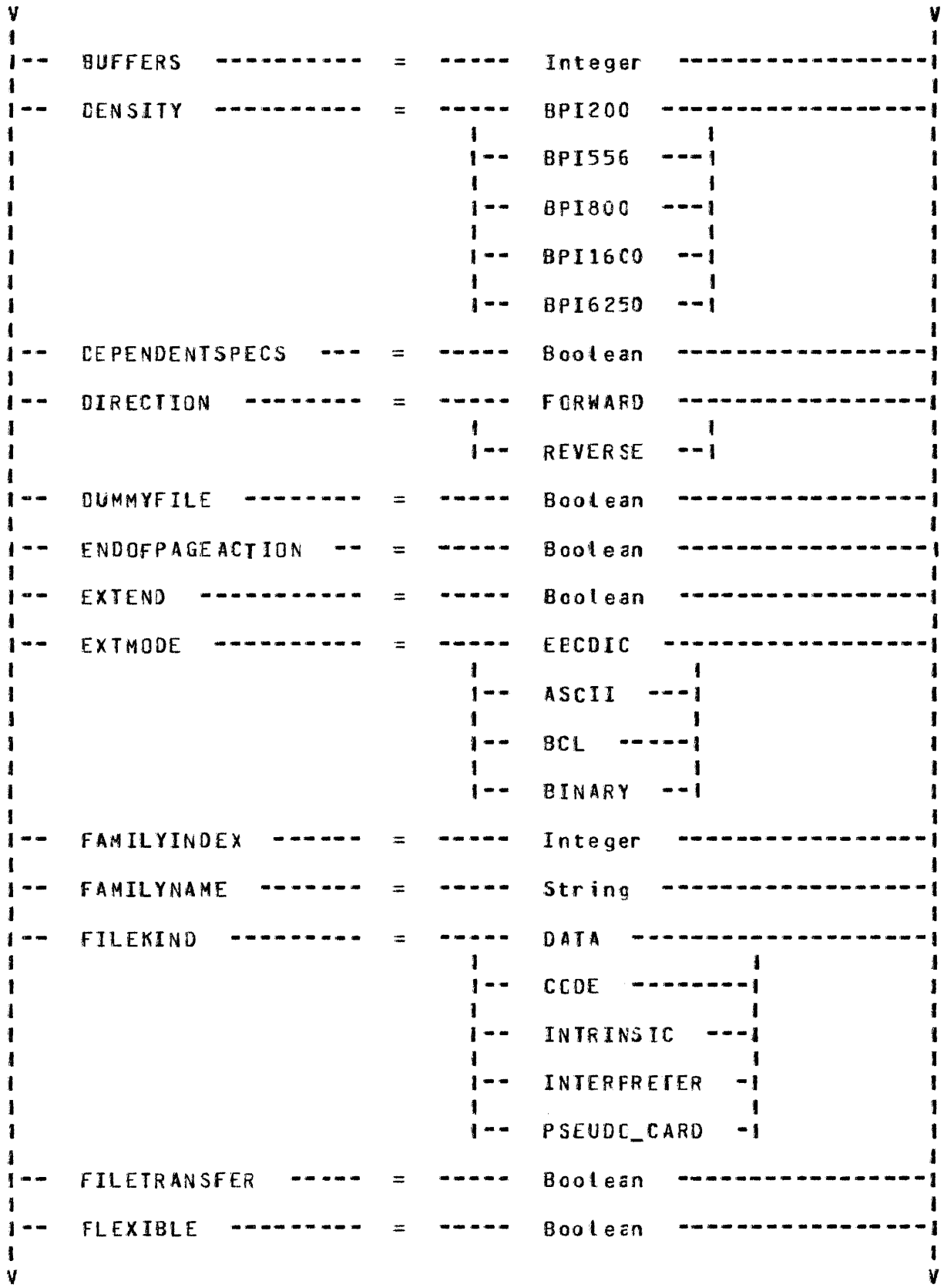
File Attributes

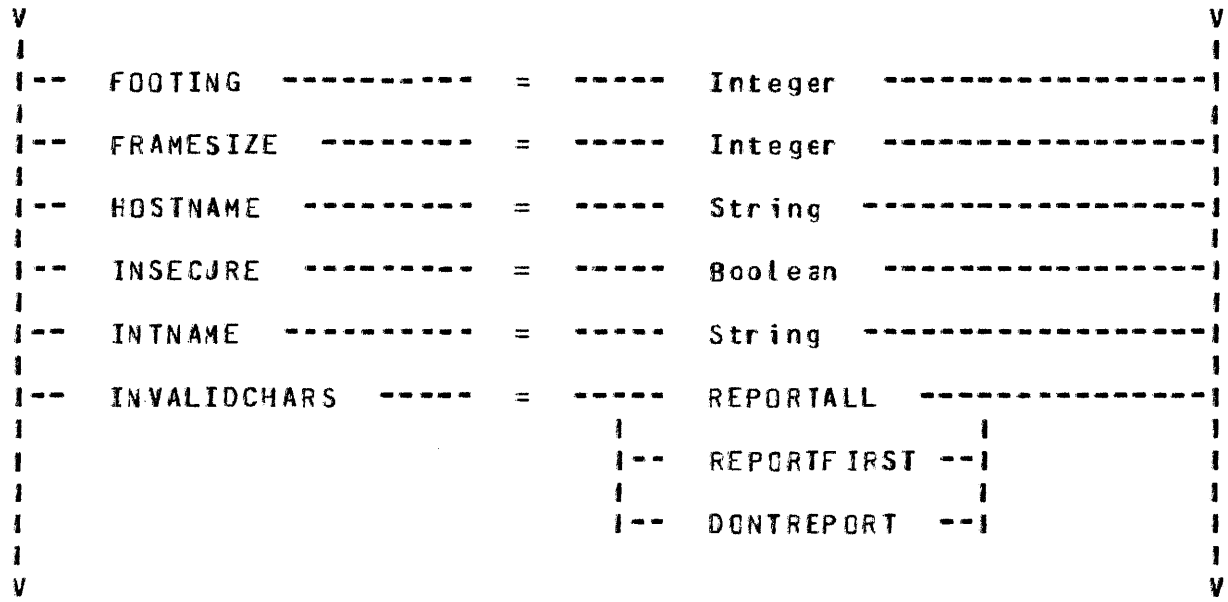
The following list of file attributes are those that are allowed in File Declarations.

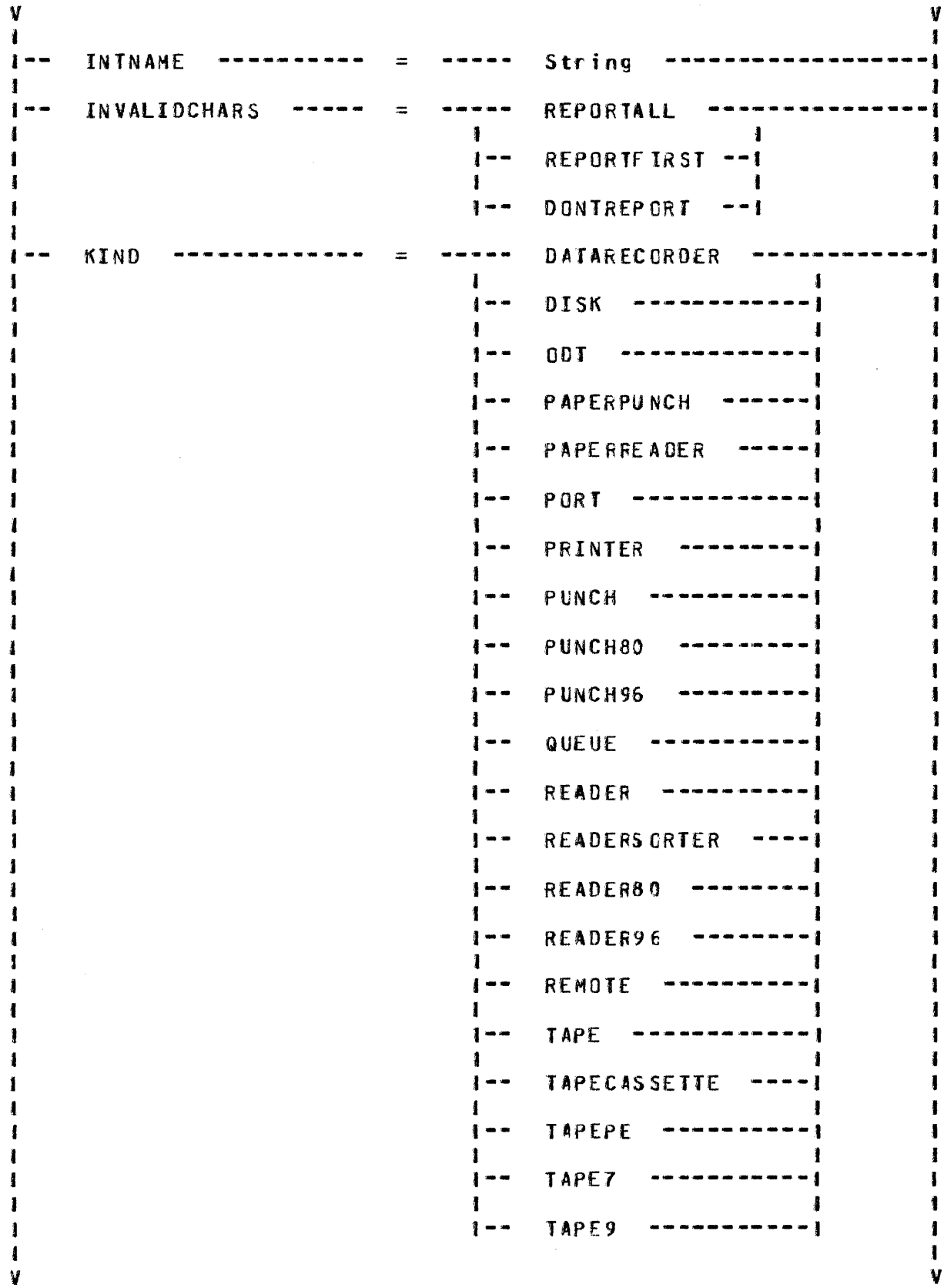
Where an attribute requires a Boolean value, the value must be TRUE or FALSE. Where an attribute requires an integer or string, the value must be a constant. Most attributes which expect mnemonic values will also allow an integer constant. However, caution should be used whenever substituting an integer for a mnemonic, since the integer values in the File Declaration statement are different from those in the Get/Put Attribute statements.

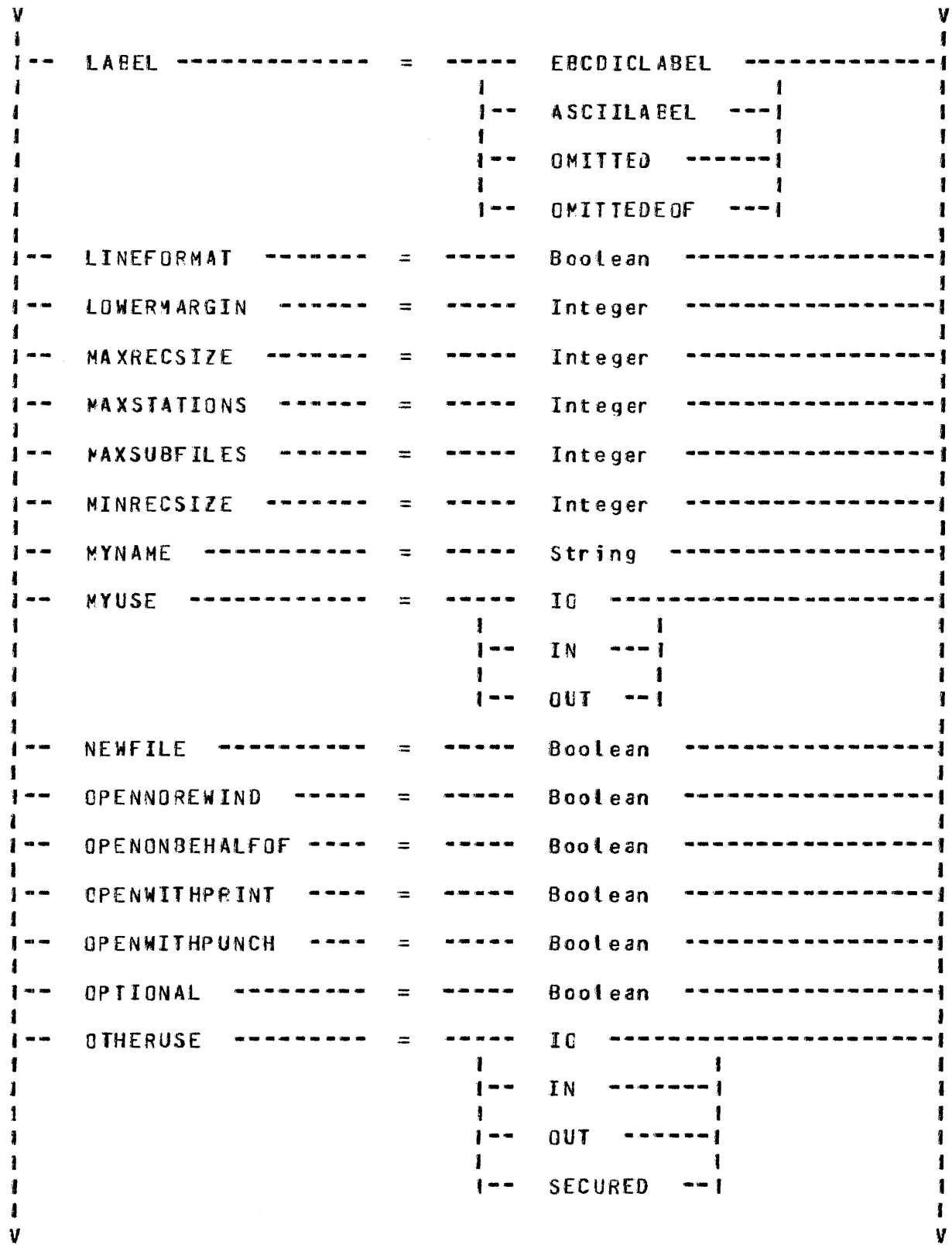
Refer to the MCP COMMUNICATES AND STRUCTURES Product Specification or the MCP CONTROL SYNTAX Product Specification for a more detailed explanation of the B1000 file attributes.

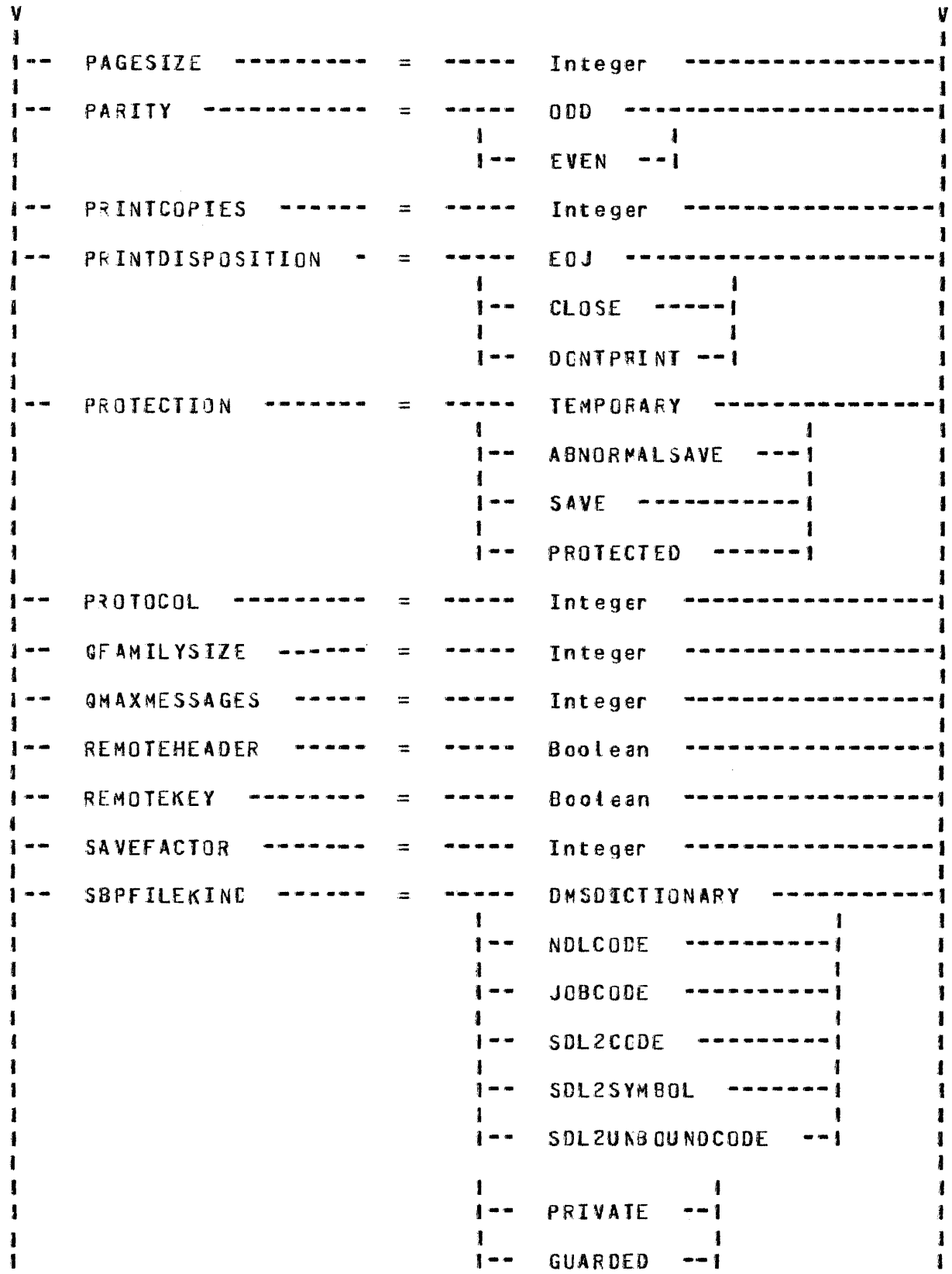


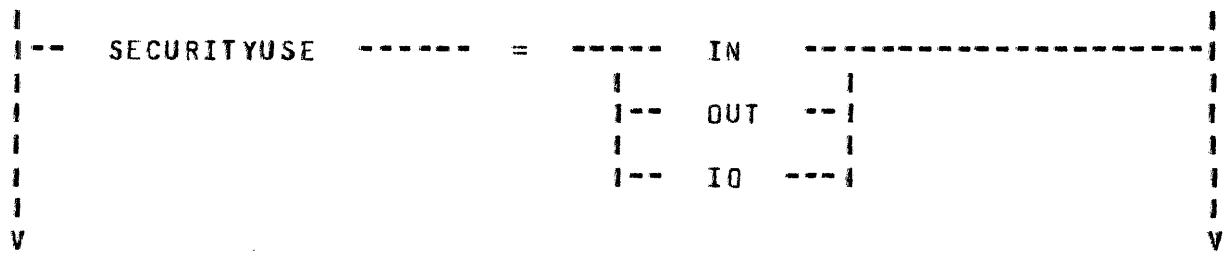

BUFFERS = Integer
DENSITY = BPI200
BPI556
BPI800
BPI1600
BPI6250
DEPENDENTSPECS = Boolean
DIRECTION = FORWARD
REVERSE
DUMMYFILE = Boolean
ENDOFPAGEACTION = Boolean
EXTEND = Boolean
EXTMODE = EBCDIC
ASCII
BCL
BINARY
FAMILYINDEX = Integer
FAMILYNAME = String
FILEKIND = DATA
CODE
INTRINSIC
INTERPRETER
PSEUDO_CARD
FILETRANSFER = Boolean
FLEXIBLE = Boolean

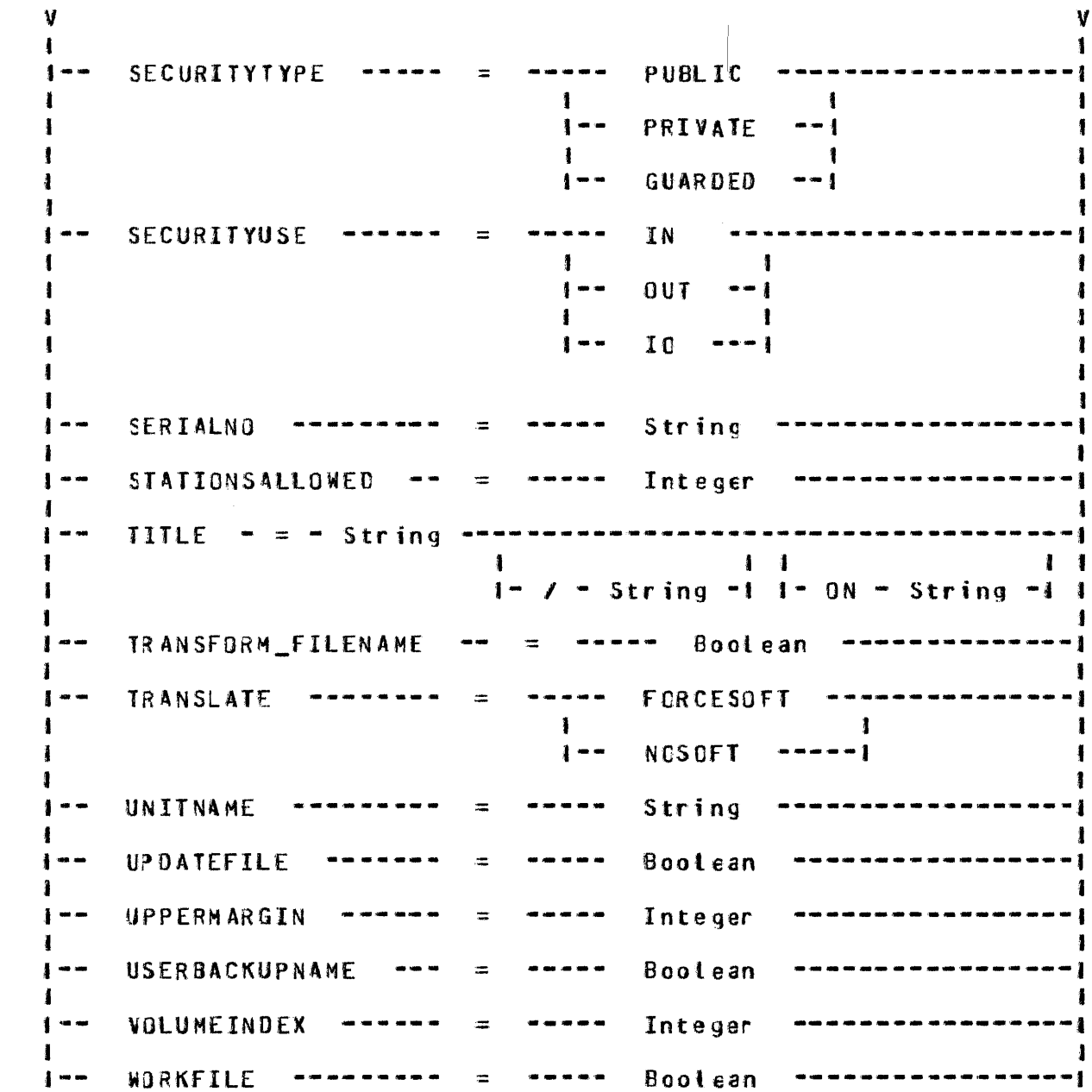

FOOTING = Integer
FRAMESIZE = Integer
HOSTNAME = String
INSECURE = Boolean
INTNAME = String
INVALIDCHARS = REPORTALL
REPORTFIRST
DONTREPORT


INTNAME = String
INVALIDCHARS = REPORTALL
REPORTFIRST
DONTREPORT
KIND = DATARECORDER
DISK
ODT
PAPERPUNCH
PAPERREADER
PORT
PRINTER
PUNCH
PUNCH80
PUNCH96
QUEUE
READER
READERSORTER
READER80
READER96
REMOTE
TAPE
TAPECASSETTE
TAPEPE
TAPE7
TAPE9


LABEL = EBCDICLABEL
ASCIILABEL
OMITTED
OMITTEDEOF
LINEFORMAT = Boolean
LOWERMARGIN = Integer
MAXRECSIZE = Integer
MAXSTATIONS = Integer
MAXSUBFILES = Integer
MINRECSIZE = Integer
MYNAME = String
MYUSE = IO
IN
OUT
NEWFILE = Boolean
OPENNOREWIND = Boolean
OPENONBEHALFOF = Boolean
OPENWITHPRINT = Boolean
OPENWITHPUNCH = Boolean
OPTIONAL = Boolean
OTHERUSE = IO
IN
OUT
SECURED


PAGESIZE = Integer
PARITY = ODD
EVEN
PRINTCOPIES = Integer
PRINTDISPOSITION = EOJ
CLOSE
DONTPRINT
PROTECTION = TEMPORARY
ABNORMALSAVE
SAVE
PROTECTED
PROTOCOL = Integer
QFAMILYSIZE = Integer
QMAXMESSAGES = Integer
REMOTEHEADER = Boolean
REMOTEKEY = Boolean
SAVEFACTOR = Integer
SBPFILEKIND = DMSDICTIONARY
NDLCODE
JOBCODE
SDL2CODE
SDL2SYMBOL
SDL2UNBOUNDCODE
PRIVATE
GUARDED


SECURITYUSE
=
IN
OUT
IO



Example:

```
FILE     INFILE     (KIND = DISK,
                     DEPENDENTSPECS = TRUE,
                     MYUSE = IN,
                     BUFFERS = 2,
                     TITLE = "INPUT"/"FILE" ON "P");
```

Default File Attributes

This section lists the default values which are used by SDL2 when an attribute is not specified in a File Declaration statement. The attributes which are not listed here default to zero, blanks, or FALSE depending on their type.

ACCESSMODE = SERIAL

AREALENGTH = blocksize * 10C

AREAS = 25

BACKUPPERMITTED = DONTCARE

BLOCKSIZE = maxrecsize

BUFFERS = 1

FRAMESIZE = 8

INVALIDCHARS = REPORTFIRST

KIND = DISK

MAXRECSIZE = 80 or 96 if card device
132 if printer device
2240 if remote device
180 in all other cases

MYUSE = IN if input only device,
OUT if output only device,
IO in all other cases

NEWFILE = TRUE if printer device,
FALSE in all other cases

PRINTCOPIES = 1

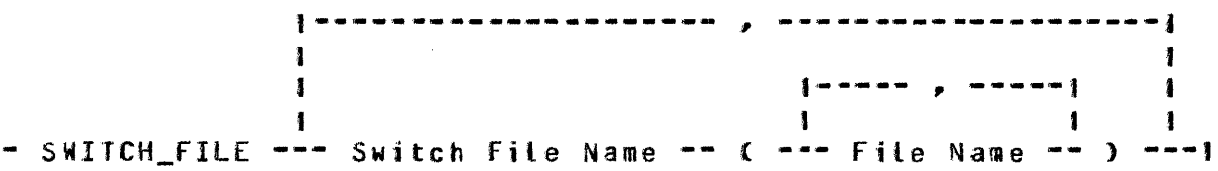
PRINTDISPOSITION = EOJ

SAVEFACTOR = 1

TITLE = internal file name

VOLUMEINDEX = 1

SWITCH FILE DECLARATION



A Switch File Declaration allows a group of files to be selectively accessed. A Switch File Name with a subscript may be used anywhere that a File Name is allowed, except within a Switch File Declaration. The files are numbered from 0 to N-1, where N is the number of files in the switch file list. If any of the files have a kind of PORT or REMOTE, then all the files in the list must be of kind PORT or REMOTE.

Example:

```
FILE USER1 (KIND=DISK),
     USER2 (KIND=DISK),
     USER3 (KIND=DISK),

SWITCH_FILE USER_FILE (USER1, USER2, USER3);

DECLARE
   USER FIXED;
.
.
.
OPEN USER_FILE (0);                    % opens file USER1
.
.
.
FOR USER := 0 TO 2
  WRITE USER_FILE (USER) (BUFFER);     % writes to all three files
.
.
.
CLOSE USER_FILE (2);                   % closes file USER3
```

INPUT/OUTPUT STATEMENTS

All Input/Output Statements operate on a File Id. A File Id may be any File, Switch File, or File Pointer.

The SDL2 Input/Output Statements map directly onto the MCP file handling communicates. Refer to the MCP COMMUNICATES AND STRUCTURES Product Specification for a more detailed explanation of each I/O communicate.

For the purposes of random I/O, record numbers are zero-relative.

On Branch

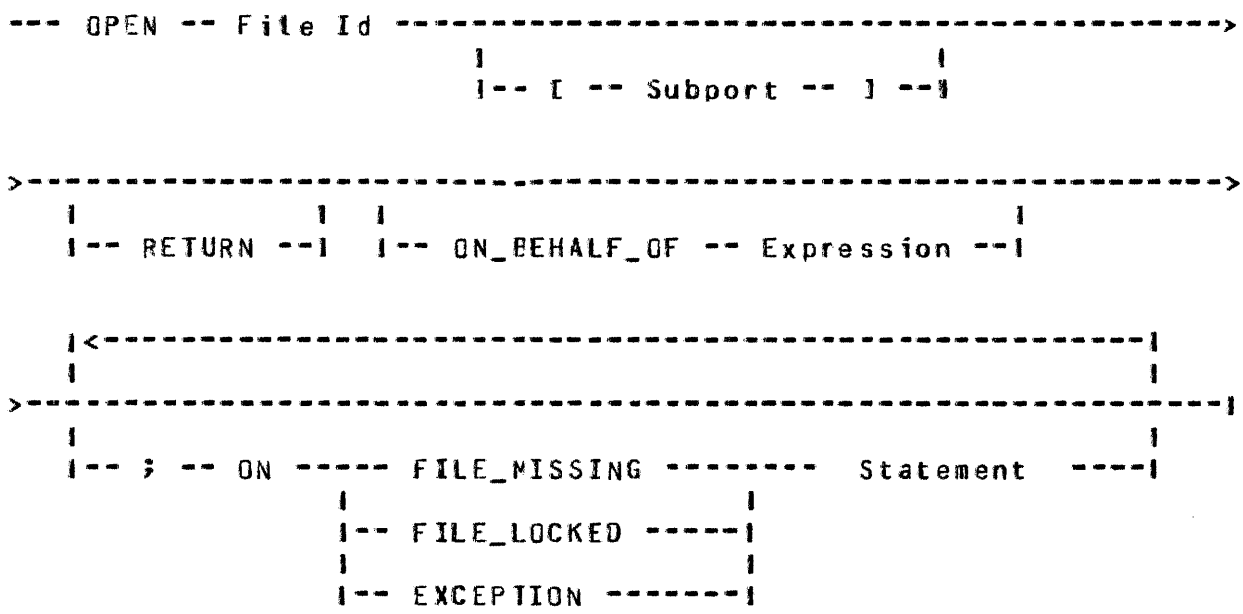
```
--- ON --- Condition --- Statement ---------------------|
```

Some of the Input/Output Statements allow On Branches to be specified. An On Branch is used to indicate that the program wishes to handle the associated error condition instead of letting the MCP handle it. If the Condition specified in the On Branch is true as a result of executing the Input/Output Statement, then the Statement following the ON Condition will be executed. If the Condition is false, then the Statement will be ignored. More than one On Branch may be specified in a single Input/Output Statement, although only one Condition can be true after executing the Input/Output Statement.

Example:

```
READ_OK := TRUE;
OPEN INFILE;
  ON FILE_MISSING DO;
     DISPLAY ("FILE NOT FOUND");
     READ_OK := FALSE;
     END;
  ON FILE_LOCKED DO;
     DISPLAY ("FILE IN USE BY ANOTHER PROGRAM");
     READ_OK := FALSE;
     END;

WHILE READ_OK DO;
  READ INFILE (BUFFER);
    ON EOF READ_OK := FALSE;
  WRITE OUTFILE (BUFFER);
  END;
```

Open



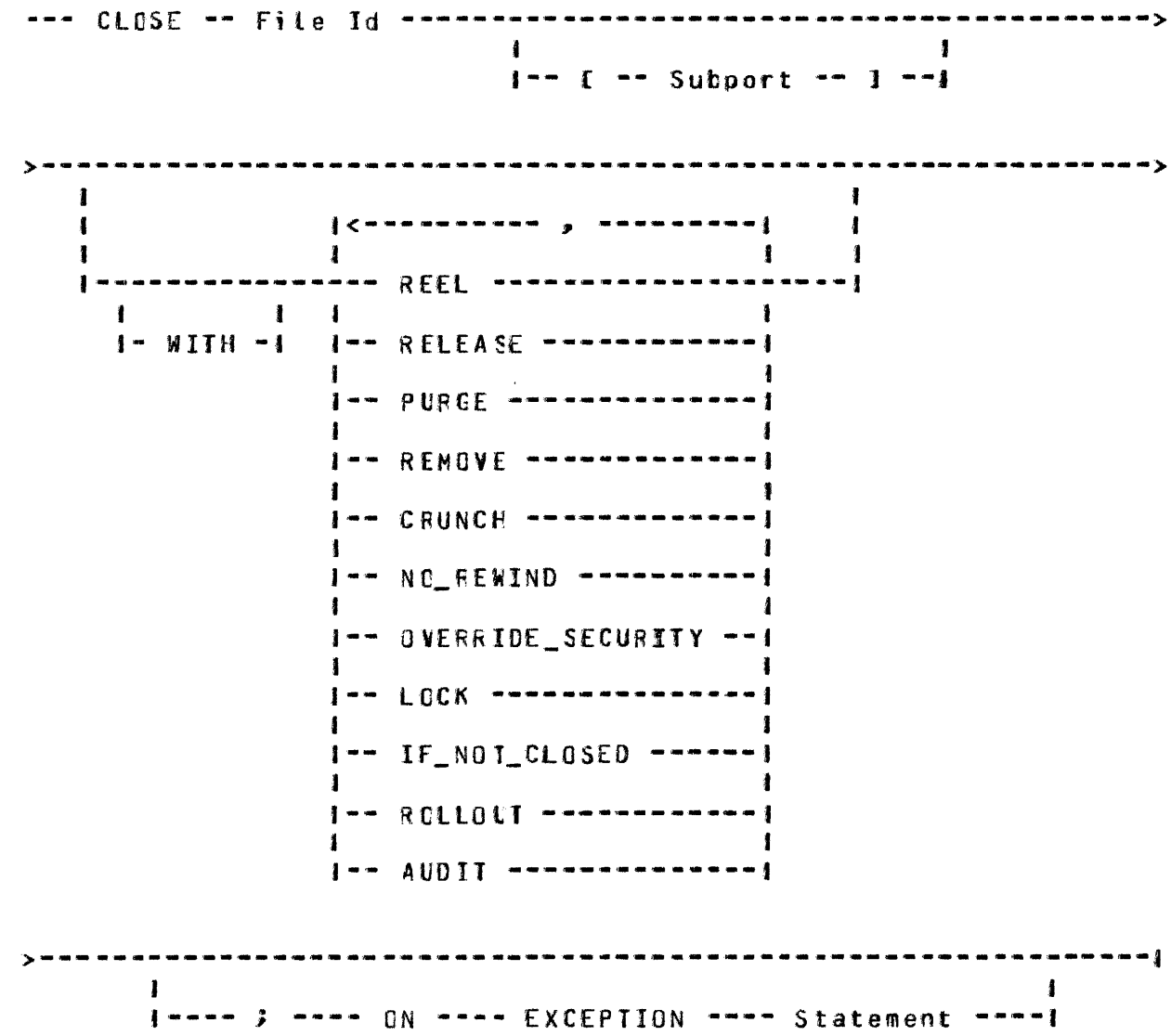
The Open Statement allows the programmer to explicitly open a file and to handle error conditions.

Note that all open options should be set before the Open Statement is executed, using either a File Declaration Statement or a Put Attribute Statement. For example, OPENONBEHALFOF must be set to True for the ON_BEHALF_OF clause to take effect.

Example:

```
OPEN SOURCE;
  ON FILE_MISSING  SOURCE_PRESENT := FALSE;
```

Close

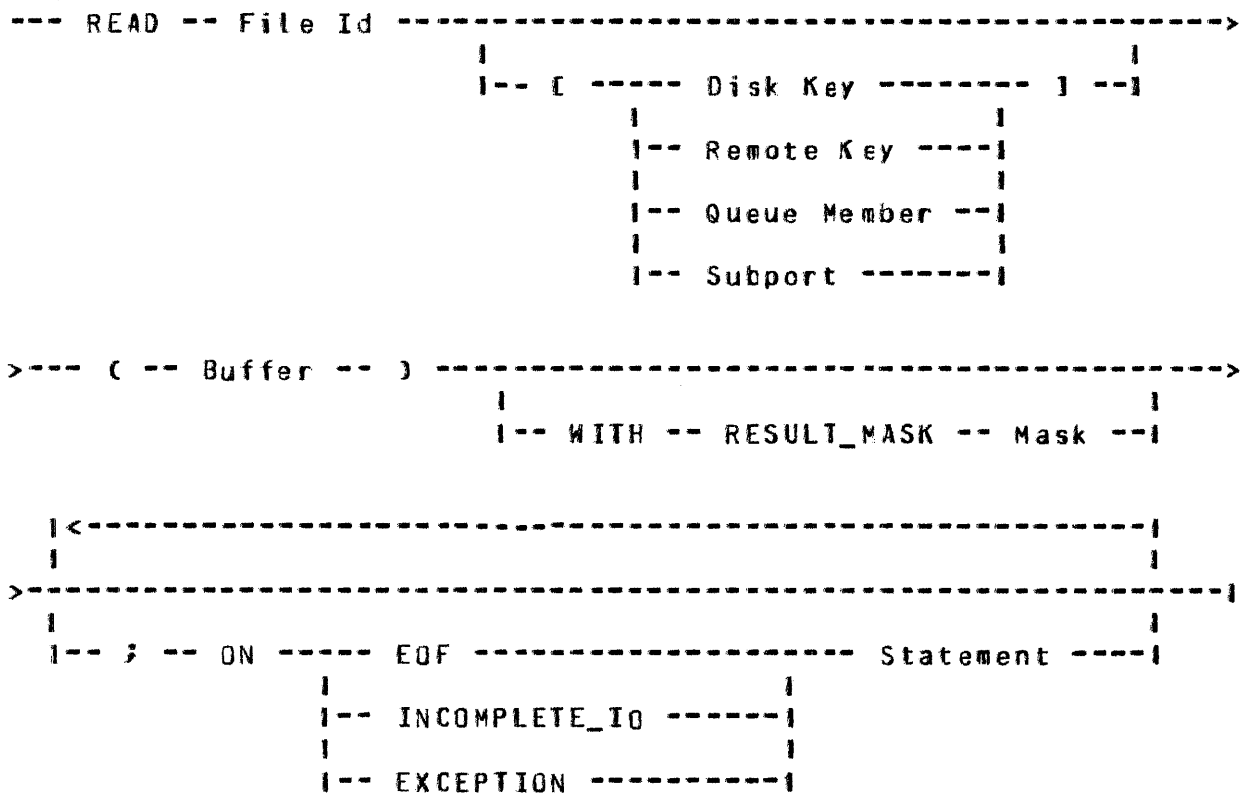


The Close Statement allows a programmer to explicitly close a file and specify how it is to be disposed.

Example:

```
CLOSE SOURCE PURGE;
CLOSE NEWSOURCE LOCK, REMOVE;
```

Read

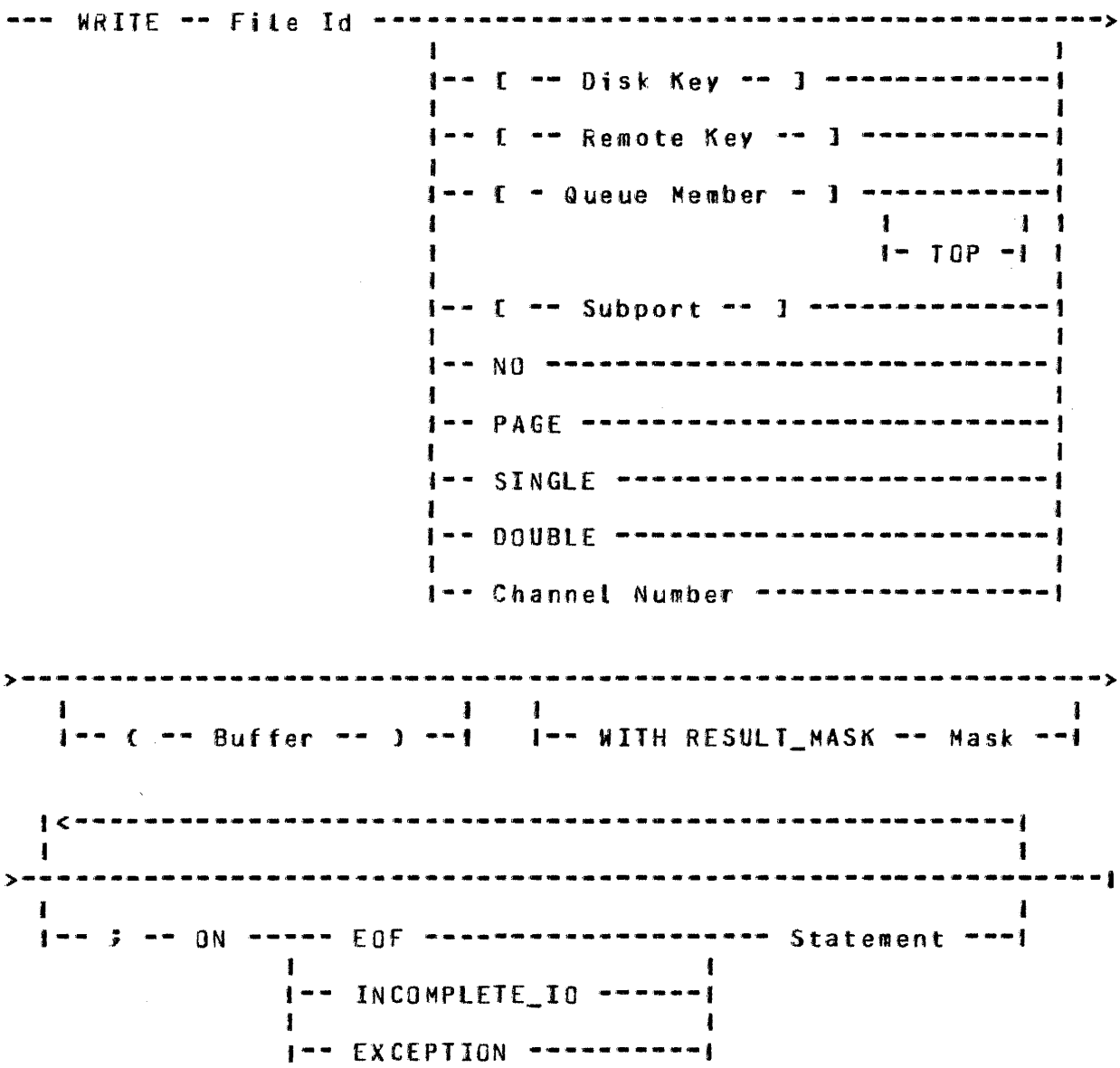


The Read Statement allows a record to be read from a file. Buffer must be an Address Generator.

Example:

```
READ RANDOM_FILE [KEY] (BUFFER);
    ON EOF BAD_KEY := TRUE;
```

Write

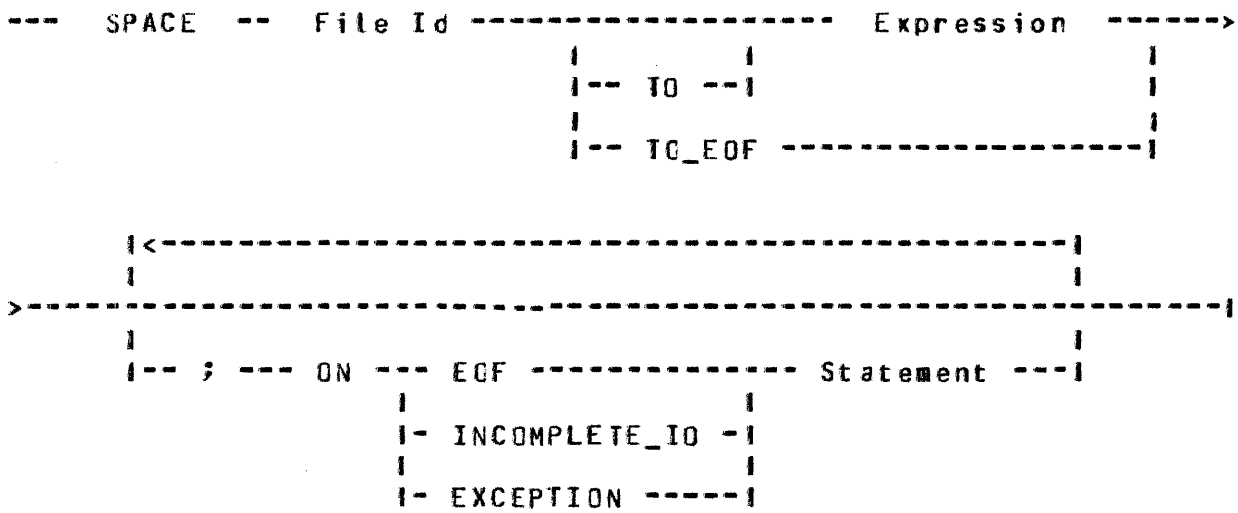


The Write Statement allows a record to be written to a file. The spacing may also be specified, and will be done after the Buffer is written.

Example:

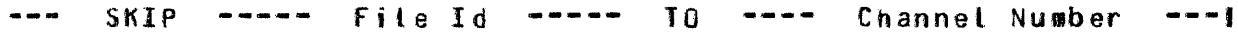
```
WRITE LINE DOUBLE ("HEADING LINE");
WRITE LINE;
```

Space



The Space Statement is used to skip over records in a sequential file. If TO Expression is used it must be a non-negative number, and will position the file at that record. Expression alone specifies the number of records to space from the current record, with a positive value spacing forward and a negative value spacing backward. TO_EOF will space the file to its current end.

Skip

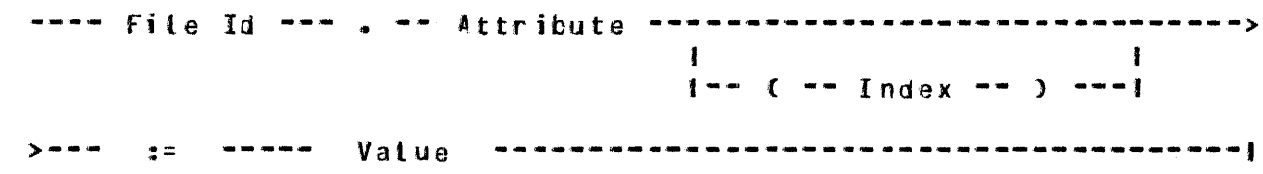


The Skip Statement will cause the line printer to skip to the specified channel number on its carriage tape. Channel Number must be a constant ranging from 1 to 12.

Seek

The Seek Statement is used with random disk files to read a record into the MCP's buffer area in preparation for a Read of that record. A Seek Statement performed immediately before a Read Statement is less efficient than just reading the record.

Put File Attribute



The Put File Attribute Statement allows an attribute of a file to be changed dynamically at run-time. Value can be either a predefined mnemonic for the attribute (as specified in the File Attributes Section), or an Expression.

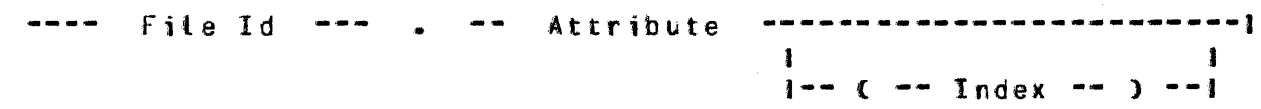
The following attributes are allowed. Attributes marked by an asterisk must be indexed.

ACCESSMODE	FLEXIBLE	PAGESIZE
ALLOCATEATOPEN		
AREAADDRESS *	FOOTING	PARITY
		PRINTCOPIES
AREABLOCKS	FRAMESIZE	PRINTDISPOSITION
AREALENGTH	HOSTNAME	PROTECTION
	INSECURE	PROTOCOL
AREAS	INTNAME	QFAMILYSIZE
AUDITED	INVALIDCHARS	QMAXMESSAGES
BACKUPKIND	KIND	REMOTEHEADER
BACKUPPERMITTED	LABEL	REMOTEKEY
BLOCKSIZE	LASTRECORD	SAVEFACTOR
BLOCKSTRUCTURE	LOWERMARGIN	SBPFILEKIND
BUFFERS	MAXRECSIZE	SECURITYTYPE
	MAXSTATIONS	SECURITYUSE
COMPRESSION *	MAXSUBFILES	SERIALNO
DENSITY	MINRECSIZE	STATIONSALLOWED
DEPENDENTSPECS	MYNAME	TITLE
		TRANSFORM_FILENAME
DIRECTION	MYUSE	TRANSLATE
DUMMYFILE		
ENDOFPAGEACTION		UNITNAME
EXTEND	NEWFILE	UPDATEFILE
EXTMODE	OPENNOREWIND	UPPERMARGIN
FAMILYINDEX *	OPENONBEHALFOF	USERBACKUPNAME
FAMILYNAME	OPENWITHPRINT	VOLUMEINDEX
FILEKIND	OPENWITHPUNCH	WORKFILE
FILENAME	OPTIONAL	YOURHOSTNAME *
FILESECTION	OTHERUSE	YOURNAME *
FILETRANSFER		YOURUSERCODE *

Example:

```
CARDS.TITLE := "SDL2_SRC/EXPRESSION ON SDL2";
```

Get File Attribute



The Get File Attribute Statement returns the value of a file attribute. The value returned will be either boolean, numeric or character, as specified in the MCP COMMUNICATES AND STRUCTURES Product Specification. Note that the ATTRIBUTE_VALUE standard function can be used to determine the numerical equivalent of a mnemonic attribute value (See STANDARD PROCEDURES AND FUNCTIONS Section).

The following attributes are allowed. Attributes marked with an asterisk must be indexed.

ACCESSMODE
ALLOCATEATOPEN
ALTERDATE
AREAADDRESS *
AREAALLOCATED *
AREABLOCKS
AREALENGTH
AREAS

ATTERR

ATTVALUE
ATTYPE

AUDITED

AVAILABLE
BACKUPFILENAME
BACKUPKIND
BACKUPPERMITTED
BLOCK

BLOCKSIZE

BLOCKSTRUCTURE

BUFFERS
CENSUS *
CHANGEDSUBFILE
COMPRESSION *

CREATIONDATE
CREATIONTIME

FAMILYNAME

FILEKIND
FILENAME

FILESECTION
FILESTATE *
FILETRANSFER
FLEXIBLE

FOOTING
FRAMESIZE

HOSTNAME
INSECURE
INTNAME
INVALIDCHARS
KIND
LABEL
LASTRECORD

LASTSUBFILE

LINEFORMAT

LINENUM
LOWERMARGIN
MAXCENSUS *
MAXRECSIZE
MAXSTATIONS
MAXSUBFILES
MINRECSIZE

OPENWITHPRINT

OPENWITHPUNCH
OPTIONAL

OTHERUSE
PAGESIZE

PARITY
PRINTCOPIES
PRINTDISPOSITION
PROTECTION
PROTOCOL
QFAMILYSIZE

QMAXMESSAGES
RECORDNUMBER
REMOTEHEADER
REMOTEKEY
SAVEFACTOR
SBPFILEKIND
SECURITYTYPE
SECURITYUSE
SERIALNO
SIMPLEHEADERS
STATE
STATIONSALLOWED
SUBFILEERROR *
TITLE
TRANSFORM_FILENAME
TRANSLATE
TRANSLATING
UNITNAME

CURRENTBLOCK
DENSITY
DEPENDENTSPECS
DIRECTION
DUMMYFILE
ENDOFPAGEACTION
EXTEND
EXTMODE
FAMILYINDEX *
MYHOSTNAME
MYNAME
MYUSE
NEWFILE
NEXTRECORD
OPEN
OPENNOREWIND
OPENONBEHALFOF
UPDATEFILE
UPPERMARGIN
USEDATE
USERBACKUPNAME
VOLUMEINDEX
WORKFILE
YOURHOSTNAME *
YOURNAME *
YOURUSERCODE *

Example:

```
WRITE LINE ( "SOURCE FILE = " CAT SOURCE.TITLE );
IF SOURCE.KIND = ATTRIBUTE_VALUE (KIND, READER) THEN
  INPUT_FROM_CARDS := TRUE;
```

SEPARATE COMPILATION

SDL2 provides a mechanism for dividing a program into several units, each of which can be incividually compiled. There are three types of units : Global units, Separately Compiled units, and Independently Compiled units. These incividually compiled units are later bound together by the SDL2/BINDER, and calls across unit boundaries are resolved.

A Global unit containing all global data declarations and the outer loop of the program must be compiled (without syntax errors) prior to the compilation of any of the Separate units. The symbol table for all global declarations (excluding procedures) is saved in the unbound global code file for later use by the Separate units.

When a Separate unit is compiled, the compiler loads the symbol table from the Global unit, making all global declarations accessible to the Separate unit. A Separate unit may not declare any global variables or files, but it may declare global types, constants, sets, records and defines which are accessible to any procedure in that unit. Note that compiler control options and conditional compilation booleans set in the Global unit will not affect any of the Separate units.

Both the Global unit and the Separate units have access to all global data declared in the Global unit. In addition, they may call any lexic level one procedure which is declared in another unit, provided the procedure is identified by an EXTERNAL procedure declaration in the calling unit. External procedure calls will be resolved by the binder. The binder will also do any parameter checking which would have been done by the compiler had the procedure been declared locally.

An Independent unit is one which accesses no global data or external procedures, and so may be bound with any Global unit. An Independent unit may not declare any global variables or files, but it may declare global types, constants, sets, records and defines which are accessible tc any procedure in that unit.

Each unit is allocated one page in the Segment Dictionary, so there is a limit of 64 units to a program. The Global unit will always be allocated to Page 0.

The following compiler control options are used when compiling unbound units. These options should appear at the front of the input file, preceding all conditional compilation statements, declarations and executable statements. The only statements that can precede these options are other compiler control statements.

CREATE_GLOBAL

> This is used to specify to the compiler that a Global unit is being compiled for later binding with Separate and/or Independent units.

USE_GLOBAL [file_name]

> This is used to specify to the compiler that a Separate unit is being compiled for later binding with a specific Global unit. The symbol table from the Global unit will be loaded by the compiler for use by the Separate unit.

INDEPENDENT

> This is used to specify to the compiler that an Independent unit is being compiled for later binding with any Global unit.

All Global, Separate and Independent 'code files' will be typed as SDL2 Unbound Code files and will not be executable.

The SDL2/BINDER will bind together a single Global unit and one or more Separate or Independent units, procucing an executable SDL2 Code file. The binder will verify that all units were compiled with the same version of the compiler and that all Separate units were compiled with the same version of the Global unit.

See the BINDER EXECUTION Section for a description of how to execute the SDL2/BINDER.

COMPILER CONTROL

Compiler control options must be preceded by a single or double dollar sign ($ or $$), beginning in column 1 of the source image. If a double '$$' is used, then the control options are considered to be permanent, and will be included in the new source file (if one is requested). If a single '$' is used, then the control options are considered to be temporary and will not be included in the new source file.

[NO] AUTO_SEGMENT (Default = TRUE)

The compiler will automatically break the program into segments. The segment size is determined by the AUTO_SEGMENT_SIZE option.

Segment breaks will occur at the end of a procedure when the code size exceeds the size specified for AUTO_SEGMENT_SIZE.

AUTO_SEGMENT_SIZE (Default = 20000)

Specifies the segment sizes in bits for automatic segmentation.

Example:

$ AUTO_SEGMENT_SIZE 16000

[NO] CHAR60 (Default = TRUE)

When TRUE the scanner will not distinguish between uppercase and lowercase letters in identifiers, including reserved words.

[NO] CLEAR_LOCALS (Default = FALSE)

When TRUE the compiler will generate code to clear the local data space to zeroes upon procedure entry. In addition, all descriptors will be initialized as though the $INITIALIZE_REFERENCES option was set.

[NO] CODE (Default = FALSE)

When TRUE the compiler will list the code generated.

CREATE_GLOBAL (Default = FALSE)

Specifies to the compiler that the Global unit of an individually compiled program is being compiled.

[NO] DEBUG <Option>

This is for compiler debugging. Option may be either CHECKS, DUMP, DUMP_SYMBOL_TABLE, RECOVERY, SCANNER, SYMBOL_TABLE, TYPES, or XREF.

DYNAMIC_MEMORY (default = 0 or minimum of 2000)

Specifies the amount of dynamic memory in bits. Dynamic memory is used for paged arrays. The default dynamic memory calculated by the compiler is sufficient for storing one page of each global paged array, plus one page of each local paged array in the lexic level 1 procedure which requires the most dynamic memory. The user can make a better estimate of the dynamic memory requirements based on the actual number of paged array elements that will be accessed at any one time. Note that each memory link requires 100 bits, and each page table requires 32 bits per entry. In addition, approximately 200 bits are required as work space for the intrinsic.

ERRORLIMIT (Default = 100)

Specifies the maximum number of errors which will be allowed. The compile will terminate when this limit is exceeded.

[NO] ERRORLIST (Default = FALSE)

When TRUE, the compiler will print the error messages in a separate error listing file (ERRORS).

FAMILY (Default = system disk)

If a control card file title does not explicitly specify a pack name, the pack specified in the FAMILY control option is used. The control cards which allow file titles are INCLUDE, MERGE, NEW, and USE_GLOBAL.

[NO] INCLNEW (Default = FALSE)

When TRUE, the source images read in from an INCLUDE file will be copied to the new source file (NEWSOURCE). The INCLNEW option takes effect only when the NEW option is specified.

INCLUDE [library file title]

Specifies that a library source is to be read in and compiled. The remainder of the source image may contain the library file title. If no title is specified, then the external name of the LIBRARY file will be used. No other control options may follow the $INCLUDE parameters on the same compiler control image. Images from the LIBRARY file will be flagged with an "I" in the output listing. A library file may not contain an INCLUDE statement.

INDEPENDENT

Specifies that an individually compiled unit is being compiled that does not access any global data. The Independent unit created can be bound into any program.

[NO] INITIALIZE_REFERENCES (Default = FALSE)

When TRUE, the compiler will generate code to initialize all reference variables to their declared type and length (if specified) with an invalid address. Use of an uninitialized reference variable will then cause a run-time Read or Write Out of Bounds error.

INTRINSIC

Specifies that an intrinsic is being compiled.

[NO] INTRINSICS_ALLOWED (Default = TRUE)

If FALSE, any use of intrinsics will be flagged as an error. If INTRINSIC is specified, then INTRINSICS_ALLOWED is automatically set to FALSE.

[NO] LIST (Default = TRUE)

Specifies if a listing is to be produced.

[NO] LISTAMPERSAND (Default = TRUE)

If TRUE, conditional compilation statements beginning with an '&' will be included in the listing. If FALSE, statements beginning with an '&' will not be listed. NO LIST overrides LISTAMPERSAND.

[NO] LISTDOLLAR (Default = TRUE)

If TRUE, all temporary compiler control statements (those beginning with a single '$') will be included in the listing. If FALSE, the temporary compiler control statements will not be listed. Permanent compiler control statements (those beginning with a double '$$') are always printed, regardless of the setting of the LISTDOLLAR option. NO LIST overrides LISTDOLLAR.

[NO] LISTHEX [id chars] (Default = FALSE)

If TRUE, the code addresses on the output listing will be printed as hex values instead of decimal values. <id chars> will be used instead of the code page number to preface the segment and offset. $VOID 00479500

[NO] LISTINCL (Default = TRUE)

If TRUE, source images from an included library file will be listed. NO LIST overrides LISTINCL.

[NO] LISTOMITTED (Default = TRUE)

If TRUE, source images that are not compiled because of conditional compilation will still be listed. The omitted images will be flagged with an "O" in the output listing. NO LIST overrides LISTOMITTED.

MERGE [source file title] (Default = FALSE)

Normally appears as the first line of the CARD file. Specifies that the CARD file be merged with the SOURCE file. The file title of the source file may be specified on the rest of of the source image. If it is not specified, then the external name of the SOURCE file will be used. Images from the CARD file will be flagged with a "P" in the output listing.

NEW [new source file title] (Default = FALSE)

Specifies that a new source file be created. The title of the new source file may be specified on the rest of the source image. If it is not specified, then the external name of the NEWSOURCE file will be used.

PAGE

Causes a page eject if listing.

PROCEDURE_STACK_SIZE (Default = 20)

Specifies the maximum depth of nested procedure calls in this program.

[NO] RUNTIME_CHECKS (Default = TRUE)

When FALSE, all run-time checking will be suppressed. (For MCP use only).

SEGMENT

Causes a segment break to be generated. This option is allowed only when it immediately follows the END of a PROCEDURE declaration.

SEGMENT_PAGE

Causes a page and segment break to be generated. This option is allowed only when it immediately follows the END of a PROCEDURE declaration.

[NO] SEQCHECK (Default = FALSE)

When TRUE, the compiler will verify that the sequence number of the current source image contains only numeric characters, and is greater than the sequence number of the previous source image. The sequence number may be all blanks only if the previous sequence number was also all blanks. An invalid sequence number will generate a syntax error.

STATIC_MEMORY (default = minimum of 10000)

Specifies the amount of static memory in bits. Increasing STATIC_MEMORY automatically enlarges the operand stack, but not the procedure stack.

[NO] SUMMARY (Default = FALSE)

When TRUE, the compiler will produce summary statistics on the output listing. If NO LIST is specified, then this summary includes all information normally found at the end of the source listing, in addition to the summary statistics.

[NO] SYMBOLIC_DUMP (Default = TRUE)

When set, symbolic information will be included in the code file for use by the Analyzer. When reset, this information will not be included in the code file. The code file will be smaller when this option is reset, but the dumps will not be informative.

[NO] TYPE_CHECKS (Default = TRUE)

When FALSE, all compile-time type checking will be suppressed. Note that the compiler currently does only a limited amount of type checking.

USE_GLOBAL [global file title]

Specifies to the compiler that a separate unit is being compiled. The title of the global code unit may be specified on the rest of the source image. If no title is specified, then the external name of the GLOBAL file will be used.

VOID [sequence number]

The VOID option may be followed by an optional Sequence Number. All subsequent records in the SOURCE file with sequence fields less than or equal to the Sequence Number will be deleted. If the Sequence Number is omitted, only the record with its sequence field equal to the sequence field of the VOID record will be deleted. The VOID option may only appear in the CARD file, and will only delete records in the SOURCE file.

[NO] WARNFATAL (Default = FALSE)

When TRUE the compiler will treat all warnings as fatal syntax errors.

[NO] XREF (Default = FALSE)

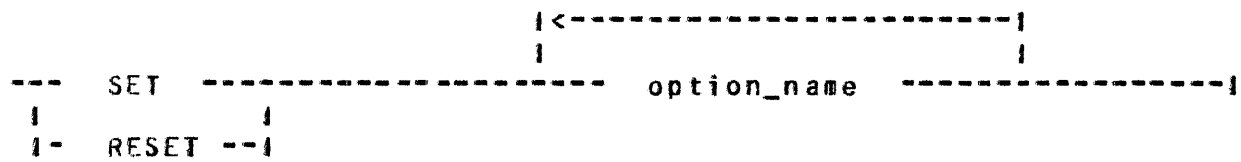
When TRUE the compiler will produce a cross-reference listing at the end of the normal program listing. The cross-reference includes all defined and predefined identifiers used in the program, excluding reserved words. It also indexes literals, ampersand booleans, dollar options, and do group labels.

BURROUGHS CORPORATION
COMPUTER SYSTEMS GROUP
SANTA BARBARA PLANT

CONDITIONAL COMPILATION

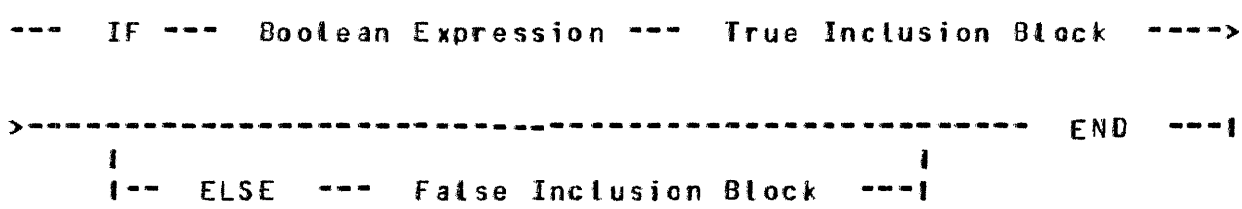
The conditional compilation facility allows the programmer to selectively compile blocks of code without the necessity of physically adding or removing records. Conditional compilation control records must begin in column 1 with an "&".

SET AND RESET STATEMENT

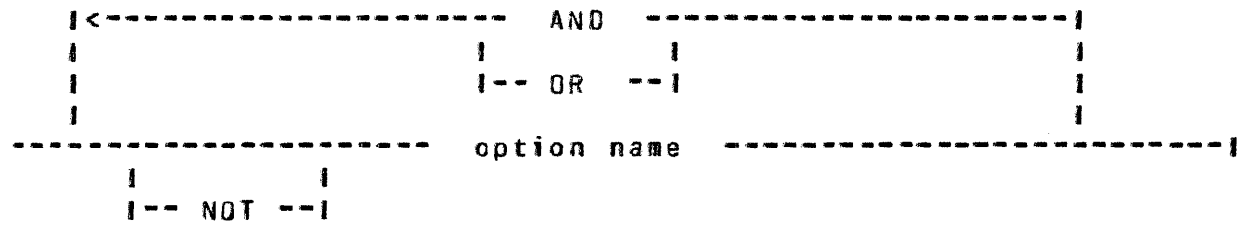


The SET and RESET statements will both declare, if not already declared, a conditional compilation option name, and SET it to true, or RESET it to false. A reference to a conditional compilation option name that has not been SET or RESET is an error.

IF STATEMENT



Boolean Expression



The conditional compilation IF statement is used to bracket blocks of code to be compiled conditionally on the result of the Boolean Expression.

Example:

```
&RESET  DEBUG_ALL
&SET    DEBUG_A
A := B;
&IF DEBUG_ALL OR DEBUG_A
   WRITE LINE ("A CHANGED TO" CAT DECIMAL(A,8));
&ELSE
   WRITE LINE PAGE;
&END
```

COMPILER EXECUTION

The primary input to the compiler is from the file CARD.

FILES

CARD	primary source input file (default device type is DISK)
SOURCE	used if MERGE option is specified
NEWSOURCE	new source file, if NEW option is specified
CODE	the generated code file
GLOBAL	code file of GLOBAL unit, if USE_GLOBAL is specified
LINE	output listing
ERRORS	error message listing, if ERRORLIST option is true
XREF	internal file for saving token references if XREF is specified
LIBRARY	file used to read in library source if INCLUDE is specified
FPBS	internal file for saving file parameter blocks

See MCP CONTROL SYNTAX Product Specification for details of MCP syntax to compile programs.

Example:

```
? CO SLD2/GLOBAL SDL2 LIBRARY;
? FI CARD NAME SDL2/SOURCE/GLOBAL;
```

BINDER EXECUTION

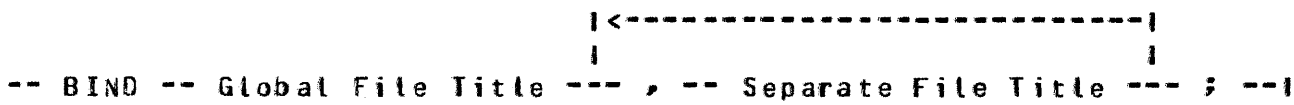
See SEPARATE COMPILATION Section for a description of how to create unbound code files.

FILES

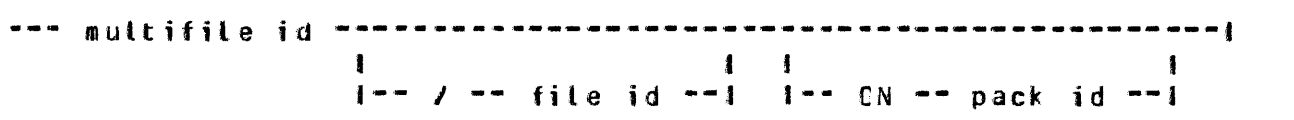
CARD	source input (default device type is DISK)
CODE	the generated code file
LINE	output listing
OBJ	the unbound code files used as input
SYMTAB	internal file for saving symbol table info

DESCRIPTION

The CARD file contains the bind request, listing the files to be bound along with any options. The syntax for the binder input file is:



Global File Title and Separate File Title are file titles conforming to the following syntax.



Binder control options are allowed on any input record with a dollar sign ($) in column one, similar to compiler control options. Valid control options and their default values are listed below.

```
[NO] LIST                 (True)
[NO] LISTHEX              (False)
[NO] SUMMARY              (False)
[NO] SYMBOLIC_DUMP        (True)
[NO] DEBUG_HASH           (False)
[NO] DEBUG_SYMBOL_TABLE   (False)
```

The first four options correspond to compiler options with the same name. If LIST is True, a listing of procedures and code addresses will be produced. If LISTHEX is True, code addresses will be printed as hex values; otherwise they will be printed as decimal values. If SUMMARY is True, summary statistics will be printed at the end of the listing. If SYMBOLIC_DUMP is False, no symbolic dump information will be included in the final code file. The DEBUG options are for binder debugging.

See MCP CONTROL SYNTAX Product Specification for details of MCP syntax to compile programs.

Example:

```
? CO XSDL2 SDL2/BINDER LIBRARY;
? FI CARD CARD.READER;
? DATA CARD
$ SUMMARY
BIND SDL2/GLOBAL ON SDL2,
     SDL2/DECLARE ON SDL2, SDL2/STATEMENT ON SDL2;
? END
```

CODE ANALYZER EXECUTION

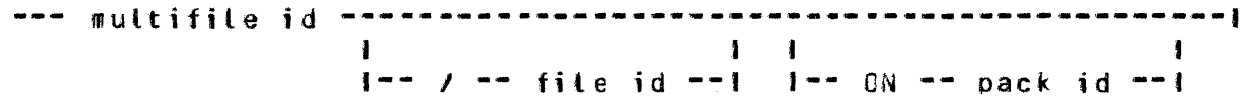
FILES

CARD	optional input file (if SWO = 1)
LINE	output listing

DESCRIPTION

The SDL2/SA program analyzes SDL2 code files. Input is via Accept commands or the CARD file, and is free format with one or more tokens per Accept command or input record. A blank Accept command or end-of-file terminates the program.

The program will first ask for the title of the code file. The entire title must appear on a single Accept command or input record, conforming to the syntax shown below. Other commands may follow the title in the same Accept command or input record.



After the file is opened and verified, the program will prompt for commands. The allowable commands are listed below.

CC

The code dictionaries will be printed. Unbound code files have only a segment dictionary, while executable code files have both a page and segment dictionary.

CODE [page] segment

Prints the contents of the code file within the specified page and segment. The page and/or segment may be ALL instead of an integer, in which case all pages or segments will be analyzed. If an unbound unit is being analyzed, then page cannot be specified.

COUNT [page] segment

Counts the number of occurrences of each S-op within the specified page and segment. As with the CODE command, ALL may be used instead of an integer value for the page and/or segment. Both the frequency count and the percentage of total operands will be printed for each S-op. If an unbound unit is being analyzed, then page cannot be specified.

DMS segment

Prints the contents of the code portion of a DMS Dictionary file within the specified segment. The segment may be ALL instead of an integer, in which case all segments will be analyzed.

END

Terminates the program.

NEXT

The current code file will be closed, and the user will be asked for the name of the next code file to analyze.

PPB

The contents of the Program Parameter Block and Scratchpad will be printed. This command is not allowed with an unbound code file.

PSS entries

The number of Procedure Stack entries will be changed to the specified number. This command is not allowed with an unbound code file.

SYM

Prints the symbol table information from a bound or unbound code file.

TEACH

Displays commands on the ODT.

UH

The contents of the Unit Header will be printed. This command is only allowed with an unbound code file.

Example :

```
SA     : ENTER FILE TITLE OR BLANK TO TERMINATE
User   : TEST/PROGRAM ON P
SA     : COMMANDS: CODE COUNT PPB PSS SD NEXT END
User   : CODE
SA     : ENTER PAGE NUMBER OR 'ALL'
User   : 0
SA     : ENTER SEGMENT NUMBER OR 'ALL'
User   : ALL
SA     : COMMANDS: CODE COUNT PPB PSS SD NEXT END
User   : CODE 1 5 PPB SD END
```

Note that command prompts are issued only when SDL2/SA expects a command but no command has been entered. If a command has already been entered, then the prompt will be suppressed.

RESERVED WORDS

This section lists the identifiers which are reserved words in SDL2.

AND
AS
BY
CAND
CAT
COR
CONSTANT
CONTAINS
DECLARE
DEFINE
DO
DOWNTO
ELSE
END
EXOR
EXTERNAL
FOR
FILE
FOREVER
FORMAL
FORMAL_VALUE
FORWARD
IF
IN
INTERSECT
MEMBER
MOD
NOT
OF
ON
OR
POINTER
PROCEDURE
RECORD
REDUCE
REFER
REFERENCE
REPEAT
RETURN
* SEGMENT
* SEGMENT_PAGE
SET
SETTING
STOP
SWITCH_FILE
THEN
TYPE
UNDO
UNION
UNTIL
* USE
WHILE

Identifiers marked with an asterisk are reserved only to improve error recovery for programs converted from SDL. They have no special function in SDL2.

APPENDIX A -- UPL2

UPL2 is a "safe" version of the SDL2 compiler. It is a subset of SDL2, and does not include the following constructs.

```
ALLOCATE_MEMORY
BASE_REGISTER
BINARY_SEARCH
CALL
CODE_ADDRESS
COMMUNICATE
COMMUNICATE_WITH_GISMO
DATA_ADDRESS
DATA_OFFSET
DATA_TYPE
DC_INITIATE_IO
DISABLE_INTERRUPTS
DISPATCH
DYNAMIC_MEMORY_BASE
ENABLE_INTERRUPTS
ERROR_COMMUNICATE
FETCH
FETCH_COMMUNICATE_MSG_PTR
FREEZE_PROGRAM
HALT
HARDWARE_MONITOR
LIMIT_REGISTER
REFER_ADDRESS
REFER_LENGTH
REFER_TYPE
SAVE_STATE
SEARCH_LINKED_LIST
SEARCH_SDL_STACKS
SEARCH_SERIAL_LIST
SHIFT
TEST
THAW_PROGRAM

Delete right assignment expression (:=)
If or Case expressions used as address generators
POINTER variables
REFERENCE and POINTER arrays

$ INITIALIZE_REFERENCES (default is True for UPL2)
$ INTRINSIC
$ LISTHEX
$ RUNTIME_CHECKS
```

REFERENCE and POINTER arrays are excluded from UPL2.

APPENDIX B -- SDL TO SDL2 CONVERSION

This Appendix provides information useful when converting SDL programs to SDL2. The first section describes SDL constructs which must be changed and/or avoided when converting to SDL2. The second section describes the standard routines which are available in SDL but not in SDL2. The third section describes how to convert SDL-style file attributes to the CSG-Standard file attributes used by SDL2. The fourth section describes how to convert SDL compiler control options to SDL2 compiler control options.

Note that this Appendix is intended as a conversion aid, not a tutorial, and for this reason does not document new SDL2 features which were not available in SDL. In particular, the Type, Constant, For, Repeat, While, and Labeled Case statements are all new in SDL2, and are described elsewhere in this document.

DIFFERENCES BETWEEN SDL AND SDL2

The following differences between SDL and SDL2 can be resolved by direct text substitution.

1) Cospatial record declarations must use '!' instead of "," to delimit fields.

2) The relational operators listed on the left, below, are no longer allowed. They must be replaced with the operators on the right.

/=	<>
NEQ	<>
EQL	=
GTR	>
LSS	<
GEQ	>=
LEQ	<=

3) SWAP must be replaced by READ_LOCK.

4) DUMP_FOR_ANALYSIS must be replaced by DUMP.

5) SPO_INPUT_PRESENT must be replaced by ODT_INPUT_PRESENT.

6) The vertical bar "!" cannot be used for assignments. It must be replaced by ":=".

7) SORT_SWAP must be replaced by ":=:".

*) LAST_LIO_STATUS must be replaced by the STATE file attribute.

8) DYNAMIC is no longer required within dynamic variable declarations, and it must be replaced by blanks.

9) Valid SDL identifier names may cause reserved-word conflicts with SDL2. New reserved words include:

CAND	MEMBER
CONSTANT	POINTER
CONTAINS	REPEAT
COR	SET
DOWNTO	TYPE
EXTERNAL	UNION
FOR	UNTIL
IN	WHILE
INTERSECT	WITH

The next set of differences cannot be resolved by direct textual substitution. However, all occurrences of these constructs in an SDL program can be easily converted to the equivalent SDL2 construct.

10) There is no UNDO (*) construct. Do-loops using this construct must be converted to named do-loops, and an undo of the outermost loop substituted for each instance of UNDO (*).

11) All formal declarations must be specified in the same order as the variables appear in the procedure heading.

12) All close options must be separated by commas.

13) While the CHAR_TABLE function is still implemented in SDL2, it no longer produces a bit table in the same order as SDL. Thus, programs which relied on the bit positions of a CHAR_TABLE string must be rewritten. For example, the following SDL code checks whether CH is a digit:

```
SUBBIT (CHAR_TABLE("0123456789"), CH, 1)
```

It must be replaced by the following SDL2 code:

```
CH IN CHAR_TABLE ("0123456789")
```

14) The GROW construct is no longer necessary, since paged arrays will automatically grow in size. The number of elements in the array may be tested by the function ARRAY_BOUND.

15) The LOCATION function, which returned a 33-bit value, must be replaced by CODE_ADDRESS, which returns a

32-bit value.

16) SDL2 treats comments and end-of-lines as separators. This means tokens cannot be split across two source images, and comments cannot be embedded within identifiers or literals. For example, the following SDL construct:

```
A := @4236/*
   */7DFC@;
```

must be replaced by the following SDL2 construct:

```
A := @4236@ CAT
     @7DFC@;
```

17) ACCEPT, DISPLAY, and ZIP are standard procedures in SDL2, so their parameters must be enclosed in parentheses.

18) Paged arrays are declared differently in SDL2. SDL required the "PAGED (elements per page)" part of the declaration before the identifier name, while SDL2 requires this part of the declaration to be placed between the array bound and the scalar type.

19) Conditional compilation booleans must always be explicitly SET or RESET before they can be used in an &IF statement. SDL initialized all undeclared conditional compilation booleans as though they were RESET.

The next set of differences requires more thought than those above.

20) No descriptor manipulation is allowed. All descriptor-relatec constructs in SDL must be replaced by the use of reference variables, and the constructs DATA_LENGTH, DATA_ADDRESS, DATA_TYPE, REFER_ADDRESS, REFER_LENGTH, and REFER_TYPE.

21) Self-relative variables will not be initialized to zero in SDL2, as they were in SDL. Instead, they must be explicitly set to zero if this is what the programmer intended. This could lead to subtle programming bugs, if any usages of non-initialized variables are not detected during the conversion process. However, if the $CLEAR_LOCALS compiler control option is specified, then the compiler will generate code to set all local data space to zero and all descriptors to invalid addresses.

22) File declarations must use CSG-Standard file attributes. The CHANGE, READ_FPB, and WRITE_FPB statements have been replaced by direct references to standard file attributes. See the FILE ATTRIBUTES Section of this Appendix for more information.

23) The OPEN statement does not allow any options, eg. OPEN INPUT. All open options must be set before the open request, using file attribute assignments.

24) The result of the logical operations OR, EXOR, and AND is always of type bit. In SDL, the result was of the same type as the longest operand, and so could sometimes be of type character.

25) All unstructured records in SDL had a type of bit. In SDL2, the record will be of type bit only if at least one of its subfields is not of type character. If all of the subfields are character fields, then the record will be of type character.

26) The extended arithmetic operators X_ADD, X_SUB, X_MUL, X_DIV, and X_MOD always returned bit strings of a predictable length in SDL. In SDL2, the length of the result can vary, depending on the number of bits required to maintain precision. In SDL, if both operands were of the same length, they they were assumed to be in twos-complement form, allowing for sign analysis. In SDL2, both operands are assumed to be positive and no sign analysis is done.

The following list describes those features which will be the most difficult to change when converting from SDL to SDL2.

27) No indexing is allowed. All indexing must be replaced by the use of record or reference variables.

28) No structured declarations are allowed, and consequently no USE statements. All such declarations must be converted to records. These changes may be simplified by the use of structured records and the WITH statement.

29) SEARCH_LINKED_LIST and SEARCH_SERIAL_LIST use records instead of templates, and are formatted slightly differently than in SDL. See the CONVERSION OF STANDARD ROUTINES Section of this Appendix for more information.

30) REMAPS is not allowed. It must be replaced either by cospatial record fields, or the use of reference variables.

The following differences between the two languages should also be noted.

31) The CREATE_MASTER and RECOMPILE capabilities do not exist in SDL2. They have been replaced by the compilation of separate program units, which are later bound together.

32) Segmentation is less flexible in SDL2. Segmentation may only be done at procedure boundaries, and only by requesting that a new segment and/or page be generated. Note, however, that SDL2 provides automatic segmentation, which was not available in SDL. User-specified segmentation is done through the use of the $SEGMENT and $SEGMENT_PAGE compiler control options. SDL SEGMENT statements will generate an advisory message, not a syntax error. Eventually, however, specifying SEGMENT outside of a compiler control option will be considered a syntax error. Note that SEGMENT cannot be defined as blanks, since this will work incorrectly when used in an IF or CASE statement.

33) There have been some changes to the compiler control options (dollar cards). Any usage of compiler control options in SDL should be checked against the allowable options in SDL2. In particular, the monitoring, profiling, and timing capabilities of SDL are not implemented in SDL2. Also, permanent compiler control options must be preceded by '$$' instead of '$'.

34) In conformance with the CSG Compiler Control Images Standard, the internal names of some files have been changed. The primary input file is CARD instead of CARDS, and the error message file is ERRORS instead of ERROR_LINE.

The following differences beteween SDL and SDL2 concern performance. Some SDL constructs are not as efficient as alternate constructs available in SDL2.

35) The SDL Reduce statement did not allow a reduction condition of "NOT IN". All statements of the form:

```
REDUCE TEXT UNTIL FIRST IN NOT SPECIALS;
```

should be replaced by the more efficient form:

```
REDUCE TEXT UNTIL FIRST NOT IN SPECIALS;
```

36) Type and Constant statements were not available in SDL. Converting Defines to Types and Constants should improve compile speed.

37) In SDL the statement

```
BUMP I;
```

was more efficient than

```
I := I + 1;
```

In SDL2 the reverse is true.

STANDARD FUNCTIONS AND PROCEDURES

UNIMPLEMENTED STANDARD ROUTINES

The following standard SDL functions and procedures are not available in SDL2.

ACCESS_FILE_INFORMATION
CHANGE_STACK_SIZES
CLEAR
CONSOLE_SWITCHES
CONTROL_STACK_BITS
CONTROL_STACK_TOP
DEBLANK
DELIMITED_TOKEN
DISPLAY_BASE
ENTER_COROUTINE
EVALUATION_STACK_TOP
EXECUTE
EXIT_COROUTINE
GROW
HASH_CODE
INTERROGATE_INTERRUPT_STATUS
INITIALIZE_VECTOR

M_MEM_SIZE
MAKE_DESCRIPTOR
MAKE_READ_ONLY
MAKE_READ_WRITE
MONITOR
NAME_STACK_TOP
NEXT_ITEM
NEXT_TOKEN
PARITY_ADDRESS
PREVIOUS_ITEM
READ_CASSETTE
READ_FILE_HEADER
READ_FPB
READ_OVERLAY
REINSTATE
RESTORE
S_MEM_SIZE
SAVE
SEARCH_DIRECTORY
SORT
SORT_MERGE
SORT_SEARCH
SORT_STEP_DOWN
SORT_UNBLOCK
THREAD_VECTOR
TRACE
VALUE_DESCRIPTOR
WRITE_FILE_HEADER

CONVERSION OF STANDARD ROUTINES

Clear

SDL syntax:

```
CLEAR ARRAY;
```

SDL2 syntax:

```
PROCEDURE CLEAR (A);
   FORMAL A (*) BIT;

   DECLARE INDEX FIXED;

   FOR INDEX := 0 TO ARRAY_BOUND (A) - 1
      A (INDEX) := 0;
   RETURN;
END CLEAR;
.
.
.
.
CLEAR (ARRAY);
```

Hashcode

SDL syntax:

```
HASH_CODE (TOKEN);
```

SDL2 syntax:

```
PROCEDURE HASH_CODE (TOKEN) BIT (24);
  FORMAL
    TOKEN           CHARACTER;
  DECLARE
    TOKEN_LENGTH  FIXED,
    INDEX         FIXED,
    CH            REFERENCE,
    HASH          BIT (27);
HASH := TOKEN_LENGTH  := LENGTH (TOKEN);
IF TOKEN_LENGTH > 15 THEN
  TOKEN_LENGTH :=15;
INDEX := 0;
WHILE INDEX < TOKEN_LENGTH DO;
  REFER CH TO SUBSTR (TOKEN, INDEX, 1);
  BUMP INDEX;
  BUMP SUBBIT (HASH, 3 - (INDEX MOD 4), 24) BY CH;
  END;
RETURN SUBBIT (HASH, 3, 24);
END HASH_CODE;
```

Note that this procedure is by no means an optimal hash function. It should be used only if it is essential to compute the identical hash values computed by SDL.

Search Linked List

SDL syntax:

```
SEARCH_LINKED_LIST  (START_RECORD,
                     COMPARE_FIELD,
                     COMPARE_VALUE,
                     RELATION,
                     LINK_FIELD)
```

SDL2 syntax:

```
SEARCH_LINKED_LIST  (START_RECORD,
                     COMPARE_FIELD RELATION COMPARE_VALUE,
                     LINK_FIELD)
```

In SDL, START_RECORD could be either an expression or an address generator. In SDL2, it must be an address generator with a RECORD type. COMPARE_FIELD and LINK_FIELD must be fields within that record type.

SDL returned the base-relative address of the first structure where the expression COMPARE_VALUE RELATION COMPARE_FIELD was true. If no match was found it returned @FFFFFF@. SDL2 returns the first structure where the expression COMPARE_FIELD RELATION COMPARE_VALUE is true, or an address generator with a data address of @FFFFFF@.

Example:

```
SDL:    RS_ADDR := SLL(START_ADDR, RS_JOB_NUMBER [0],
                       JN, =, RS_Q_LINK [0]);
        IF RS_ADDR = @FFFFFF@
           THEN NOT_FOUND;

SDL2:   DECLARE FIRST REFERENCE RS_NUCLEUS,
                RSN   REFERENCE RS_NUCLEUS;
        REFER_ADDRESS (FIRST, START_ADDR);
        REFER RSN TO SLL(FIRST,
                         RS_JOB_NUMBER = JN,
                         RS_Q_LINK));
        IF DATA_ADDRESS (RSN) = @FFFFFF@
          THEN NOT_FOUND;
```

Search Serial List

SDL syntax:

```
SEARCH_SERIAL_LIST (COMPARE_VALUE,
                    RELATION,
                    COMPARE_FIELD,
                    START_RECORD,
                    TABLE_LENGTH,
                    RESULT_VARIABLE)
```

SDL2 syntax:

```
SEARCH_SERIAL_LIST (START_RECORD,
                    COMPARE_FIELD RELATION COMPARE_VALUE,
                    TABLE_ITEMS)
```

START_RECORD must be an address generator with a RECORD type, and COMPARE_FIELD must be a field within that record type. Or, START_RECORD can be any address generator, and COMPARE_FIELD must be omitted; in that case START_RECORD will be used as the comparison field.

In SDL, TABLE_LENGTH specified the number of bits to be searched. In SDL2, TABLE_ITEMS must be used instead of TABLE_LENGTH. TABLE_ITEMS specifies the number of sequential entries that are to be searched.

SDL returned a 1 if the search succeeded, and a 0 if it failed, with RESULT_VARIABLE being set to the base relative address of the entry containing COMPARE_VALUE. SDL2 does not use a parameter to return the result of the search. Instead, the function returns the element number of the matching entry (the first element is 0). If no match is found, then TABLE_ITEMS will be returned.

Example:

```
SDL :  IF SSL(JN, =, UNIT_JOB_NUMBER [0],
          IOAT_REC [HINT.IOAT_POINTER],
          HINTS.IOAT_END - HINTS.IOAT_POINTER,
          IOAT_ADDR) = 0
        THEN NOT_FOUND;

SDL2: DECLARE IOAT REFERENCE IOAT_REC;
      REFER_ADDRESS (IOAT, HINTS.IOAT_POINTER);
      ENTRY := SSL (IOAT, UNIT_JOB_NUMBER=JN, IOAT_ENTRIES);
      IF ENTRY = IOAT_ENTRIES
        THEN NOT_FOUND;
        ELSE IOAT_ADDR := HINTS.IOAT_POINTER +
                          ENTRY * IOAT_SIZE;
```

FILE ATTRIBUTES

SDL2 uses CSG-Standard file attributes in File Declarations and in Get/Put Attribute Statements. The CSG-Standard attributes implemented by the B1000 MCP are documented in the MCP COMMUNICATES AND STRUCTURES Product Specification.

This section lists the attributes allowed in SDL, along with the equivalent standard attributes available in SDL2. It does not list all of the standard attributes allowec by SDL2. These are documented in the INPUT/OUTPUT Section.

Attributes Used In File Declaration Statements

SDL	SDL2
ALL_AREAS_AT_OPEN	ALLOCATEATOPEN = TRUE
AREA_BY_CYLINDER	not implemented
AREAS = areas/blks_per_area	AREAS = areas AREALENGTH = chars_per_rec * recs_per_blk * blks_per_area or, AREABLOCKS = blks_per_area
BUFFERS =	BUFFERS =
DEVICE = (in alphabetical order) CARD CARD_PUNCH forms backup CARD_READER CASSETTE DATA_RECORDER_80 DISK access DISK_FILE access DISK_PACK access DISK_PACK_CENTURY access DISK_PACK_CAELUS access PAPER_TAPE_PUNCH forms backup PAPER_TAPE_READER PORT PRINTER forms backup PUNCH forms backup PUNCH_PRINTER forms backup QUEUE (max msgs) QUEUE (max msgs) FAMILY (size) READER_PUNCH_PRINTER forms backup READER_SORTER	KIND = DATARECORDER DISK ODT PAPERPUNCH PAPERREADER PORT PRINTER PUNCH PUNCH80 PUNCH96 QUEUE READER READERSORTER READER80 READER96 REMOTE TAPE TAPECASSETTE TAPEPE TAPE7

READER_96	TAPE9
REMOTE (max msgs) remote_options	
SORTER_READER	
SPO	
TAPE	
TAPE_NRZ	
TAPE_PE	
TAPE_7	
TAPE_9	
access = null SERIAL RANDOM DELAYED_RANDOM SERIAL_WITH_OVERWRITE	ACCESSMODE = SERIAL RANDOM DELAYEDRANDOM not implemented
forms = null | FORMS	not implemented
backup = null BACKUP BACKUP DISK BACKUP TAPE NO BACKUP OR BACKUP OR BACKUP DISK OR BACKUP TAPE	BACKUPKIND = DISK TAPE BACKUPPERMITTED = DONTCARE DONTBACKUP MUSTBACKUP
queue max msgs	QMAXMESSAGES =
queue family size	QFAMILYSIZE =
remote options = WITH HEADERS FAMILY	REMOTEHEADER = TRUE QFAMILYSIZE > 1
END_OF_PAGE_ACTION	ENDOFPAGEACTION = TRUE
EU_INCREMENTED = EU_SPECIAL =	\ > FAMILYINDEX = /
EXCEPTION_MASK =	not implemented
FILE_TYPE = DATA INTERPRETER CODE INTRINSIC PSR_DECK	FILEKIND = DATA INTERPRETER CODE INTRINSIC PSEUDO_CARD
HOST_NAME =	HOSTNAME =
INVALID_CHARACTERS =	INVALIDCHARS =
LABEL = mfid/fid	TITLE =

LABEL_TYPE =	
ANSI	LABEL = EBCDICLABEL or ASCIILABEL
BURROUGHS	not implemened
UNLABELED	LABEL = OMITTED or OMITTEDEOF
LOCK	not implemented
MODE =	
ODD	PARITY = ODD
EVEN	PARITY = EVEN
EBCDIC	EXTMODE = EBCDIC
ASCII	EXTMODE = ASCII
BCL	EXTMODE = BCL
BINARY	EXTMODE = BINARY
MULTI_PACK	not implemented
NUMBER_OF_STATIONS =	STATIONSALLOWED =
OPEN_OPTION =	
INPUT	MYUSE = IN
INPUT/OUTPUT	MYUSE = IO
INTERPRET	OPENNOREWIND = TRUE
LOCK	OTHERUSE = IN
LOCK_OUT	OTHERUSE = SECURED
NEW	NEWFILE = TRUE
NO_REWIND	OPENNOREWIND = TRUE
ON_BEHALF_OF	OPENONBEHALFOF = TRUE
OUTPUT	MYUSE = OUT
PUNCH	OPENWITHPUNCH = TRUE
PRINT	OPENWITHPRINT = TRUE
REVERSE	DIRECTION = REVERSE
STACKERS	not implemented
OPTIONAL	OPTIONAL = TRUE
PACK_ID =	FAMILYNAME =
PROTECTION =	SECURITYTYPE =
PROTECTION_ID =	SECURITYUSE =
RECORDS = chars_per_rec/recs_per_blk	MAXRECSIZE = chars_per_rec BLOCKSIZE = chars_per_rec * recs_per_blk
REEL =	VOLUMEINDEX =
REMOTE_KEY	REMOTEKEY =
SAVE =	SAVEFACTOR =
SECURITYTYPE =	SECURITYTYPE =

SECURITYUSE =	SECURITYUSE =
SERIAL =	SERIALNO =
TRANSLATE = fileid	TRANSLATE = FORCESOFT fileid not implemented
USE_INPUT_BLOCKING	DEPENDENTSPECS = TRUE
USER_NAMED_BACKUP	USERBACKUPNAME = TRUE
VARIABLE	BLOCKSTRUCTURE = VARIABLE SBPFILEKIND = 11
WORK_FILE	WORKFILE = TRUE

Attributes User In Change (Get/Put) Statements

This section lists the file attributes which were allowed in an SDL Change Attribute Statement, but not in a File Declaration. For a complete list of file attributes which are allowed in SDL2 Get/Put Attribute Statements, refer to the INPUT/OUTPUT Section.

SDL	SDL2
BLOCKS_PER_AREA = blks_per_area	AREABLOCKS = blks_per_area
EU_DRIVE =	FAMILYINDEX =
EU_INCREMENT =	FAMILYINDEX =
FILE_ID =	FILENAME =
MULTI_FILE_ID =	FILENAME =
NUMBER_OF_AREAS =	AREAS =
OPEN_ON_BEHALF_OF =	OPENONBEHALFOF =
QUEUE_FAMILY_SIZE =	QFAMILYSIZE =
QUEUE_MAX_MESSAGES =	QMAXMESSAGES =
REMOTE_HEADERS =	REMOTEHEADER =
RECORDS_PER_BLOCK = recs_per_blk	BLOCKSIZE = chars_per_rec * recs_per_blk
RECORD_SIZE = chars_per_rec	MAXRECSIZE = chars_per_rec

COMPILER CONTROL OPTIONS

SDL2 allows different compiler options than SDL. Most of the changes were made to conform to the CSG Compiler Control Images standard. Permanent compiler options must be preceded by '$$' in SDL2, instead of '&' as in SDL.

This section lists the options allowed in SDL, along with the equivalent options available in SDL2. It does not list all of the compiler options allowed by SDL2. These are documented in the COMPILER CONTROL Section.

SDL	SDL2
ADVISORY	not implemented
AMPERSAND	LISTAMPERSAND
CHECK	SEQCHECK
CODE	CODE
CONTROL	LISTDOLLAR
CONVERTDOTS	not implemented
CREATE_MASTER	not implemented
CSSIZE	PROCEDURE_STACK_SIZE
DEBUG	various
DETAIL	not implemented
DOUBLE	not implemented
DYNAMICSIZE	DYNAMIC_MEMORY
ERROR_FILE	ERRORLIST
ESSIZE	STATIC_MEMORY
EXPAND_DEFINES	not implemented
FORMAL_CHECK	not implemented
FREEZE	not implemented
INTERPRETER	not implemented

INTRINSIC	INTRINSIC
LIBRARY	INCLUDE
LIBRARY_PACK	FAMILY
LIST	LIST
LISTALL	LISTOMITTED and LIST
LOCKI	not implemented
MERGE	MERGE
MONITOR	not implemented
NEST_PROCEDURE_TIMES	not implemented
NEW	NEW
NO_DUPLICATES	not implemented
NO_SOURCE	not implemented
NSSIZE	STATIC_MEMORY
PAGE	PAGE
PASS_END	not implemented
PPSSIZE	PROCEDURE_STACK_SIZE
PPROFILE	not implemented
PROFILE	not implemented
RECOMPILE	not implemented
RECOMPILE_TIMES	not implemented
SEQ	not implemented
SINGLE (SGL)	not implemented
SIZE	not implemented
SUPPRESS	not implemented
TIME_BLOCKS	not implemented
TIME_PROCEDURES	not implemented
TIME_MCP	not implemented

USEDOTS	not implemented
VOID	VOID
VSSIZE	STATIC_MEMORY
WORKING_SET_BYTES	not implemented
XMAP	not implemented
XREF	XREF
XREF_ONLY	not implemented

INDEX