[illegible]

1983 9992

**Burroughs Corporation**

COMPUTER SYSTEMS GROUP
PASADENA PLANT

# B2000/B3000/B4000 SPRITE REFERENCE MANUAL

COMPANY CONFIDENTIAL

**PRODUCT SPECIFICATION**

REVISIONS

| REV LTR | REVISION ISSUE DATE | PAGES REVISED, ADDED, DELETED OR CHANGE OF CLASSIFICATION | PREPARED BY | APPROVED BY |
|---|---|---|---|---|
| A | 03-03-83 | Review Copy - Initial | B. Wilkinson Belinda Wilkinson<br>B. Fishman Bill Fishman 3-3-83 | |
| | 04-20-83 | APPROVED | | A. Johnson [illegible] 4-22-83<br>R. Solt [illegible] 4-21-83<br>S. Talwar [illegible] 4/20/83<br>R. Airhart [illegible] 4-27-83 |

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

## DESCRIPTION OF MANUAL

This reference manual is organized so that all information needed to understand the implementation and use of the SPRITE language system is easily accessible.

To facilitate an understanding of this language system, this manual begins with an introduction to the SPRITE system process followed by explanations of the language semantics and syntax. Included in the appendices are SPRITE STANDARD OPERATIONS (Appendix A), EDIT PICTURE OPERATIONS (Appendix B) and FILE ATTRIBUTES (Appendix C). Explanations of the Railroad Syntax diagrams used throughout this manual can be found in Appendix D. Appendix D (SYNTAX DIAGRAMS) should be well understood before attempting to interpret the syntax diagrams used throughout this manual.

Prior to presentation of heavy detail on semantics and syntax of the language, the programmer is given explanations of the Module Interface Description facility, the program module, and SPRITE procedures. The balance of this manual is dedicated to detailed semantics and syntax explanations of the SPRITE language.

Items marked with an asterisk (*) are not implemented in the current release of this product.

## REQUIRED BACKGROUND

Users of this manual are presumed to understand programming in one or more high-level languages, such as ALGOL or PASCAL. Experience with PASCAL is especially useful. Familiarity with B4000/B3000/B2000 Series computers and their data representations will be useful when dealing with machine-dependent functions. Familiarity with the MCPVI operating system control statements is helpful when compiling, binding, and executing programs.

## INTRODUCTION

The SPRITE language system was designed to facilitate the division of major software projects into manageable program units or modules. The modular design approach is one of the major characteristics of the SPRITE language system.

This system is composed of its programming language, called SPRITE, for coding the modules; its MODULE INTERFACE DESCRIPTION FACILITY, whose grammar and syntax is couched in the SPRITE language, for specifying information about how the modules will interface and correspond with each other; and lastly composed of its related BINDER SPECIFICATIONS or specifications that handle the segmentation and overlay activity of the modules in memory. The grammar and syntax for these BINDER SPECIFICATIONS are different from the SPRITE language covered in this document. However, we will present basic BINDER SPECIFICATIONS for executing SPRITE modules. Below is a graphic illustration of the SPRITE system process:

1 SPRITE SYSTEM PROCESS

```
                                                               . . . . . . . . . . .
                                                              .                     .
                                                     //=>     .     ICM.MID         .
                                   . . . . . . . . . . . . .  ||   . . . . . . . . . . .
          . . . . . . . . . . .    .                       .  ==//
  1.      .MOD INTER-  .           .     SPRITE/P          .
          .FACE DESC.  .  ======>  .     COMPILER          .  ==\\       . . . . . . . . . . .
          . . . . . . . . . . .    . . . . . . . . . . . . .    ||      .      MID          .
                                                                ||      .     SYMBOL        .
                                                                \\=>    .     TABLE         .
                                                                         . . . . . . . . . . .
                                                                                  /\
                                                                                  ||
                                            //=================//
                                            ||
                                            ||
                                            \/
          . . . . . . . . . . .    . . . . . . . . . . . . .           . . . . . . . . . .
          .PROG SOURCE.            .                       .           .                  .
  2.      . MODULE.A   .  ======>  .      SPRITE           .======>    .     ICM.A        .
          . . . . . . . . . . .    .     COMPILER          .            . . . . . . . . . .
                                   . . . . . . . . . . . . .

(Repeat Step 2 for PROG SOURCE MODULE.B yielding ICM.B)

          . . . . . . . . . . .
          .                   .
          .   ICM.MID         .  ==\\
          . . . . . . . . . . .      ||
                                     ||
          . . . . . . . . . . .      ||     . . . . . . . . . . . . .       . . . . . . . . . . . . .
          .                   .      \\===>  .                       .     .  EXECUTABLE           .
  3.      .   ICM.A           .  =======>  .     BINDER            . =====>.     CODE              .
          . . . . . . . . . . .      //===>  .                       .     .     FILE              .
                                     ||     . . . . . . . . . . . . .       . . . . . . . . . . . . .
          . . . . . . . . . . .      ||
          .                   .      ||
          .   ICM.B           .  ==//
          . . . . . . . . . . .
```

1 SPRITE SYSTEM PROCESS

```
                                                               ...........
                                                              .           .
                                                        //=>  .  ICM.MID  .
                                  .............          ||    ...........
         ............            .             .      ==//
    1.  .MOD INTER- .            .   SPRITE/P  .
        .FACE DESC. .  ======>   .   COMPILER  .      ==\\     ...........
         ............             .............         ||    .    MID     .
                                                        ||    .   SYMBOL   .
                                                        \\=>  .   TABLE    .
                                                               ...........
                                                                   /\
                                                                   ||
                                     //================//
                                     ||
                                     ||
                                     \/
         ............             .............                ..........
                                 .             .              .          .
        .PROG SOURCE.            .             .
    2.  . MODULE.A  .   ======>  .   SPRITE    .======>       .  ICM.A   .
         ............            .   COMPILER  .               ..........
                                  .............

   (Repeat Step 2 for PROG SOURCE MODULE.B yielding ICM.B)

         ..........
        .          .
        .  ICM.MID . ==\\
         ..........    ||
                       ||
         ..........    ||            .............           .............
        .          .   \\===>       .             .         .  EXECUTABLE .
    3.  .  ICM.A   . =======>       .    BINDER   . =====>  .    CODE     .
         ..........    //===>       .             .         .    FILE     .
                       ||            .............           .............
         ..........    ||
        .          .   ||
        .  ICM.B   . ==//
         ..........
```

1 SPRITE SYSTEM PROCESS (Continued)

Below is a brief explanation of the SPRITE system process:

A. The MODULE INTERFACE DESCRIPTION (MID) is the first component of the programming system the programmer must submit to the SPRITE compiler. Assuming the compile is successful, the compiler proceeds to build an ICM.MID and a MID Symbol Table based on the data supplied from the MID source. The ICM.MID contains all shared data block initialization information.

B. After the ICM.MID and MID Symbol Table are built, the programmer must individually compile each source program module in his SPRITE system program. Here we are using two, MODULE.A and MODULE.B.

As the modules are being submitted to the SPRITE compiler, the compiler simultaneously accesses the associated MID Symbol Table and proceeds to build a 'pseudo code file' for each module. These 'pseudo code files' are called Independently Compiled Modules or more frequently, ICMs.

C. After the ICMs have been successfully created by the SPRITE compiler, the programmer must then submit, as a single group, the ICMs to the BINDER.

The BINDER will host the job of building a single executable code file. This 'executable codefile' can be thought of as the 'finally compiled version' of the SPRITE system source language program. This finally compiled version is the file the programmer will execute via the appropriate system EXecute command.

1 SPRITE SYSTEM PROCESS (Continued)

Below is a brief explanation of the SPRITE system process:

A. The MODULE INTERFACE DESCRIPTION (MID) is the first component of the programming system the programmer must submit to the SPRITE compiler. Assuming the compile is successful, the compiler proceeds to build an ICM.MID and a MID Symbol Table based on the data supplied from the MID source. The ICM.MID contains all shared data block initialization information.

B. After the ICM.MID and MID Symbol Table are built, the programmer must individually compile each source program module in his SPRITE system program. Here we are using two, MODULE.A and MODULE.B.

As the modules are being submitted to the SPRITE compiler, the compiler simultaneously accesses the associated MID Symbol Table and proceeds to build a 'pseudo code file' for each module. These 'pseudo code files' are called Independently Compiled Modules or more frequently, ICMs.

C. After the ICMs have been successfully created by the SPRITE compiler, the programmer must then submit, as a single group, the ICMs to the BINDER.

The BINDER will host the job of building a single executable code file. This 'executable codefile' can be thought of as the 'finally compiled version' of the SPRITE system source language program. This finally compiled version is the file the programmer will execute via the appropriate system EXecute command.

## 2 SPRITE LANGUAGE

SPRITE, a high level language, was developed for System Software implementors using the Burroughs 4000, 3000, and 2000 series of Computers.

It is a procedural, statement-oriented, strongly typed, Pascal-like language. The SPRITE language provides the means for the programmer to define, simply and clearly, the data structures and algorithms best suited to a problem. There are no GO TO statements or statement labels in the SPRITE language.

## 3 PROGRAM TEXT

SPRITE program text can be composed via the EDITOR formatted records or on standard 80 column data cards. In the EDITOR environment the first 8 positions are available for sequence numbers and the remaining 72 are available for program text. With standard 80 column data cards, positions 1-72 are used for program text and positions 73-80 are reserved for sequence numbers.

## 4 TEXT FORMAT

All entries are completely free-format within the text position limits. The programmer is free to choose his own style and standard of incentation for readability. However, this Reference Manual does include text format guidelines in Appendix G.

## 5 COMMENTS

A comment is simply text usec to document the program. Comment text is ignored by the compiler after it is recognized to be a comment. Comments may appear in SPRITE text in either of two forms.

In the first form, the comment follows the percent (%) sign. In general, wherever a percent sign appears, the text following it on the line is regarded as a comment and is ignored by the compiler. The exception is when the % sign appears within a string literal, as in "Used 50% more disk space". In this case, the % sign is regarded as part of the string and not as a comment indicator.

Example:

```
    %This is one form of comment text
```

In the second form, the comment text appears between the reserved words, COM and MOC. In this form they may appear only where Declarations, Definitions, Statements and Module Entry Point Descriptions are allowed. Unlike the first form, this second form enables comment text to flow from one line to the next within text oosition limits.

```
    COM
         This is a comment of an-
            other form. This form allows
              comment text to be continually
                 listed from one line to the next.
    MOC;
```

## 6 ELEMENTS OF THE LANGUAGE

The most primitive elements in the SPRITE language are these basic symbols:

1. The upper-case letters A..Z

2. The lower-case letters a..z

3. The digits 0..9

4. The operational and punctuation symbols in Table 6-1.

Table 6-1

| Arithmetic | Logical | Relational | Assignment | Bracketing | Other |
|---|---|---|---|---|---|
| + | & | < | := | ) | ; |
| - | \| | > | :=: | ( | , |
| * | # | = | +:= | [ | . |
| / | ¬ | <= | -:= | ] | .. |
|  | && | >= | *:= | { | : |
|  | \|\| | ¬= | /:= | } | :: |
|  |  |  | &:= | blank | @ |
|  |  |  | \|:= | " | % |
|  |  |  | #:= |  | _ |

Upper-case letters (A..Z), digits (0..9), and the underscore (_) character are used to create names of a particular class of words called INDICANTS. These are used to designate a set of predefined data types or user defined data types, file attributes and their mnemonic values. In addition these letters, digits and the underscore character are used to designate predefined reserved words that bring structural and/or semantic context to the SPRITE program.

Lower-case letters (a..z), digits (0..9), and the underscore (_) character are used to create names of another particular class of words, called IDENTIFIERS, and to designate another set of predefined reserved words that invoke standard functions and represent constants.

## 6 ELEMENTS OF THE LANGUAGE (Continued)

Operational symbols have several uses. Some of these uses are indicated by the category groupings shown in Table 6-1. However, these groupings are not meant to be definitive of the symbol's only category of usage.

The blank must be used to separate names, reserved words, and literals from one another (unless they are separated by some other symbol). The usages of the other symbols will be discussed as the need for them arises.

## 6.1 WORDS OF THE LANGUAGE

The basic words used to construct the components of a SPRITE program are combined according to certain rules. Those basic words can be categorized as names (i.e., identifiers and indicants), reserved words, literals, and operational symbols.

### 6.1.1 Names

Names may be used to represent objects and definitions in SPRITE Modules (program units) and in SPRITE Module Interface Descriptions. If a name exceeds 30 characters, the SPRITE compiler will recognize only the first 30. Further, all names must be unique within their scope (see 10).

A name is considered 'defined' when it is associated with the object or definition it is to represent. Further, a name allows an object and definition to be referenced anywhere within its reference boundary limits or 'scope' of its name.

There are two major classes of names:

1. IDENTIFIERS

2. TYPE INDICANTS

Each of these two classes has a subset of predefined names.

## 6.1.1 Names (Continued)

## 6.1.2 Identifiers

Identifiers are a class of names used to label the following:

|  |  |
|---|---|
| Symbolic Value Names | Procedure Names |
| Defined Constant Names | Data Block Names |
| Variable Names | File Block Names |
| Field Names (STRUCtured data items) | File Names |
| Program Names | Port Block Names |
| Module Names | Port Names |

Identifiers are composed of:

Lower-case letters (a..z)
Digits (0..9)
Underscore Character ( _ )

Examples:

|  |  |
|---|---|
| equal_sign | mid_prog |
| upper_limit | disk_check_module |
| array_index | my_first_proc |
| name_field_part2 | token1_data_block |

The initial character of all identifiers **must** be one of the set of lower-case letters (a..z). The rest of the characters may be any combination of lower-case letters, digits, and the underscore character. Only the first 30 characters of any identifier name are recognized by the compiler. No user-created identifier may be the same as any SPRITE predefined name or SPRITE reserved word.

Predefined Identifiers

This subset of identifiers are used to accomplish the following:

Invoke Standard Module Procedures
Invoke Standard Functions
Access Standard Constants

## 6.1.2 Identifiers (Continued)

The Predefined Identifiers are:

| Modules | Functions | | Constants | Field identifier |
|---|---|---|---|---|
| port_io | abs | move_words | true | filler |
| prog | edit_number | ptr | false | |
| io | fill_array | ptr_add | nil | |
| mcp | fill_string | ptr_sub | | |
| | index | proc_ptr | | |
| | index_any | round | | |
| | index_inc | scale_ptr | | |
| | index_none | translate | | |
| | length | upb | | |
| | lwb | | | |

Examples:

```
prog.wait(length_of_time);  % This statement instructs the MCP to
                            % wait before reinstating a program.

io.close_purge(filename);   % This statement instructs the MCP to
                            % close and purge a file.

abs(variable_name);         % This function returns the absolute
                            % value of some named variable.

x_array := fill_array(0);   % This function sets all the elements
                            % of 'x_array' to zero.

always_on  := true;         % This condition sets the boolean
                            % 'always_on' to true.
```

These predefined identifiers may also be used as field names in what are called a STRUCtured data type names. See section 13.3.3 for further details.

The usage of these predefined identifiers as part of a STRUCtured name overrides their predefined connotation.

## 7 TYPE INDICANTS

TYPE indicants are used to name data types, i.e., BOOLEAN, EBCDIC, etc. TYPE indicants are composed of:

Upper-case letters (A..Z)
Digits (0..9)
Underscore character ( _ )

The initial character of all TYPE indicants must be one of the set of upper-case letters (A..Z). The rest of the characters may be any combination of upper-case letters, digits, and the underscore character. The only restriction is that the programmer may not compose TYPE indicant names that are the same as SPRITE reserved words or other predefined data type (TYPE indicant) names.

NOTE: SPRITE allows the programmer to define his own data types beyond those already provided in the language. Those already provided are simply termed "predefined type indicants".

The predefined type indicants are:

| | | | |
|---|---|---|---|
| BIT | BOOLEAN | CHAR | EBCDIC |
| HEX | *LONG_REAL | *REAL | TRANSLATE_TABLE |

An indicant must be defined before it is used.

## 8 LITERALS

Literals are source representations for numbers and strings. Literals are used to construct denotations to represent values of particular types.

String literals are delimited by quotemarks and may contain from 1 to 70 characters from the SPRITE character set. Any literal larger than 70 characters in length must use the string concatenation operator (+) (see 13.3.2). A string literal may not exceed a source text line boundary. An embedded quote is represented by two adjacent quotes. A string literal of a single character is automatically converted to TYPE CHAR (see 13.2.1.3) whenever context requires it.

Examples:

```
   3
 100
 " "
 " 0123456789"
 "he said ""help""."
"1.....69" + "70.....80"
```

## 9 CASTS AND COERCIONS

A cast is an explicit conversion of data of one type to another type. A coercion is an automatic, implicit conversion of data. Data types are compatible if one can be converted to the other by cast or coercion.

If the context of a data type is ambiguous, the desired type must be supplied explicitly.

Casts provide an unambiguous context for conversion where it would not otherwise exist. See 19.2.1.3.

See 13.7 for a list of possible casts and coercions.

## 10 SCOPE

The scope of a name is the extent of the program text in which it is known. The scope determines the accessibility of a named object or definition. A name in the SPRITE system may be at any one of four levels of scope: program-local, module-local, procedure-local, and statement-local.

The accessibility of a name, which appears in the MID can be restricted by specifying a KNOWS list (see 12.3) for the name. When a KNOWS list is used, the name is known only in the modules that appear in the KNOWS list.

Higher level scopes do not necessarily encompass lower level scopes (as in a strictly block-structured language). Instead some names with high level scopes must be specifically imported to be accessible in more restricted contexts. SHARES declarations (see 15.5) are used to import the names of variables from data blocks (see 18.2), the names of files, ports, and nsp files from file, port and nsp file blocks respectively (see 18.3, 18.4) into procedures.

Generally speaking, when we say "file blocks" we mean file blocks, port blocks and nsp file blocks. When we say "files" we mean files, ports, and nsp files.

All the primary names (names of modules, local procedures, data blocks, variables, constants, types) that are known at a specific point in the program must be uniquely defined. In particular this means that a name may not be redefined within its scope and that two identical names may not have scopes that intersect. Qualified names, i.e., structure field names and external procedures, need be unique only within the structure definition and module interface description, respectively, because they always appear elsewhere preceded by the structure or module name.

Global names are known in every program. Global names are the predefined identifiers and the predefined indicants. Global names may not be redefined at any point in the program.

### 10.1 PROGRAM-LOCAL NAMES

These names are defined in the program MID (see 12). Program-local names include names of modules, module entry points, data blocks, the variables they contain, the names of "file blocks", the various "files" they contain and constants and indicants defined in the MID.

## 10.1 PROGRAM-LOCAL NAMES (Continued)

The standard module procedures also fall in this category.

The scope of a program-local name is the MID and the complete text of any module listed in the KNOWS list of the name. Exceptions are the names of variables in a data block, and the names of files in a file block. The scope of these names is limited to those procedures that share the data block, file block or port block.

A further restriction on naming is that variables in any data block must have distinct names from the variables in all other data blocks of the program defined in the MID, and similarly for files and file blocks and ports and port blocks.

## 10.2 MODULE-LOCAL NAMES

Module-local names are defined and known in a single module (see 11). They are names of procedures, constants, types and data definitions that are known throughout the module where they are defined but nowhere else. Variables in the data definitions are also module-local but are known only in the procedures that SHARE the data block.

## 10.3 PROCEDURE-LOCAL NAMES

Procedure-local names are known only in a single procedure (see 18). These names incluce constant names, variable names, and data type names that are defined in the procedure. These names may be defined differently in other procedures.

## 10.4 STATEMENT-LOCAL NAMES

These names are known only within the context of a single statement (see 14). FOR and FIND statement control variables and UNTIL-CASE statement situation names are statement-local names. They must be unique with respect to all names of higher scope which are also known in the statement of the statement-local name.

## 11 SPRITE MODULE STRUCTURE

A module is the basic unit of compilation in the SPRITE system. It must consist of at least one procedure definition and may contain any number of constant definitions, type definitions, and data definitions. The syntax of a module is:

```
|
|                                    ________ ; ________
|                                   /                   \
\__module name___MOD____\____procedure def__/_________DOM__
                         \___constant def__/  \_ ; _/       \
                         |\__type def_____/|                 |
                         |\__data def_____/|                 |
                         \___comment_______/                 |
                                                             |
                                                             |
```

Figure 10-1

As can be seen, the SPRITE module is made up of a number of components braced by the reserved words, MOD and DOM. DOM is simply the reverse spelling of MOD.

PROCEDURE DEFINITION (See 16.1)

A procedure definition associates an identifier with a block of code and its data. The procedure definition is the only **mandatory** component in a module. The others may be included as the programmer's needs dictate.

CONSTANT DEFINITIONS (See 15.1)

A constant definition defines and associates an identifier with a value. Using this permanently unchangeable identifier name has the same effect as using the value in all contexts.

TYPE DEFINITIONS (See 15.2)

A type definition defines a data type and associates an indicant name with that type. Whenever the indicant name is used, the effect is the same as though the type definition were used.

DATA DEFINITIONS (See 18.1)

A data definition associates an identifier with a group of variables in a data block. Using this identifier name

## 11 SPRITE MODULE STRUCTURE (Continued)

allows the data block to be referenced or SHARED by other procedures in the module.

Example:

```
     scanner
     MOD

        CONST line_length = 80;
        TYPE LINE = STRING (line_length);

        input_block
        DATA

             current_position 1..line_length := 1,
             line LINE;

        next_char
        PROC RETURNS CHAR;

            SHARES input_block;
            COM statements go here MOC;

        CORP;

        get_token
        PROC (token VAR TOKEN_TYPE);

            SHARES input_block;
            VAR current_char CHAR;
            COM statements go here MOC;

        CORP;


     DOM
```

The names specified in the CONSTANT, TYPE, and DATA, and procedure definitions are known throughout this module but not in any others. For definitions in the MID, this condition of being known 'locally' can be modified by the use of the KNOWS List Specifications in the MID portion of the SPRITE system program. (See section 12.3).

## 12 MODULE INTERFACE DESCRIPTION (MID)

Each SPRITE program has associated with it a MID. The MID is used to specify the allowable interactions among the modules of the program, and to define names which may be used throughout the program (program-local names).

A MID may contain any number of constant definitions, type definitions, declaration blocks, DATA, FILE, PORT or NSP block definitions. and module descriptions. Each component may have associated with it a KNOWS list (see 12.3) that specifies which modules will know the names defined in the component.

A MID is also a basic unit of compilation in the SPRITE system.

The syntax of a MID is:

```
|                           __________ ; __________
|  program:                /                       \
\__ident____PRUG__\___________component___/_________GORP__
                    \_knows_/            /   \_ ; _/        \
                    |                    |                  |
                    \__comment__________/                   |
```

The syntax of component is described by:

```
|
|
|\________constant definition______________________________
|                                                          \
|\________type definition_________________________________ |
|                                                         \|
|\________data definition_________________________________ |
|                                                         \|
|\________file definition_________________________________ |
|                                                         \|
|\________declaration block_______________________________ |
|                                                         \|
|\________module description______________________________ |
|                                                         \|
\_________port definitions________________________________ |
                                                          \|
                                                           |
                                                           |
```

## 12.1 DECLARATION BLOCKS

A declaration block may be used to group some number of constant, type, and data definitions together. This allows a single KNOWS list to be associated with all the components of the declaration block.

The syntax of a declaration block is:

```
|
|        _____________ ; ____________
|       /                            \
\__DEC__\______constant definition_____/____________CED__
           \                        /   \__ ; __/       \
           |\__type definition_____/|                    |
           |                        |                    |
           |\__data definition_____/|                    |
           |                        |                    |
           |\__file definition_____/|                    |
           |                        |                    |
           |\__port definition_____/|                    |
           |                        |                    |
           |\__nsp definition______/|                    |
           |                        |                    |
           \___comment_____________/                     |
                                                         |
                                                         |
```

## 12.2 MODULE DESCRIPTIONS

A module description formally specifies the interface of a given module of the program. Exactly one of the module descriptions must designate where the program starts. This description houses the program entry point procedure. This particular program entry point has no formal parameters. All other module entry point procedures may specify formal parameters.

A module description contains an interface description of every procedure that may be called from other modules. Such a procedure is known as an entry point of the module. Procedures that may not be called from other modules do not need to appear in the module description. They may be described in the MID for documentation purposes.

The interface description of each entry point of the module specifies the name of the entry point and the name, access and type of each formal parameter of the entry point. Also, if the entry point is a function then the type of the return value must be specified.

12.2 MODULE DESCRIPTIONS (Continued)

The use of the reserved word ENV_DEPENDENT allows the module to use modified types, i.e. PACKED symbolics, DISPLAY (see 13.1.2), EBCDIC (see 13.2.1.3) and STRUCs or PACKED STRUCs which use omitted or disjoint tag fields. This implies that some implementation dependent effect is exploited and that consequently the module may not behave as intended in a different environment.

The syntax of a module description is:

```
|
|
\__module:ident__MOD_______________________
                     \__ENV_DEPENDENT__/    \
   ________________________________________/
  /
  |   _____________  ;  _____________
  |  /                               \
  |  |          entry point:         |
  \__\__________procedure description___/___________DOM__
      \_knows_/                      /     \__ ; __/       \
       \__comment___________________/                      |
                                                           |
                                                           |
```

The syntax of procedure description is:

```
|
|
\__proc:ident____PROC_____________________________________________
                 \          \__proc parameters__/ \__returns__/  \
                 |                                          |    |
                 \__ENTRY___________________________________/    |
                                                                 |
                                                                 |
```

The syntax of proc parameters is:

```
|          ___ , _______________________________
|         /  param: \                           \
\__ ( __\__ident__/___________________________type__/__ ) __
               \__CONST__/  \__UNIV__/                      \
               |\__VAR__/|                                  |
               \__VALUE__/                                  |
```

12.2 MODULE DESCRIPTIONS (Continued)

The syntax of returns is:

```
|
|
\_____RETURNS_____type___________________________________
                                                         \
                                                         |
                                                         |
```

Note that the reserved word ENTRY signals that the procedure is the program entry point. One and only one procedure name in each MID must be followed by ENTRY.

Also note that a KNOWS list may prefix a procedure description. This KNOWS list must be a subset of the module's KNOWS list. The procedure name is then known only in the modules in its KNOWS list.

The order, type, access (VAR, CONST, or VALUE), and presence or absence of UNIV of the formal parameters must match that specified in the procedure's definition. The appearance of UNIV preceding a formal parameter type means that the parameter is universal. The use of a universal parameter relaxes type checking on parameter passing.

The SPRITE compiler will check that a module meets its MID requirements.

## 12.3 KNOWS LIST

Each component of the MID may be prefixed by a KNOWS list. The KNOWS list contains the names of those modules within the program which may access the component. If a component has no KNOWS list, then all modules have access to it. If an object is known by a module, then the module must know all of the components used in the definition of that object. KNOWS lists restrict access to a component. A module name may appear in a KNOWS list which precedes the the description for that module. KNOWS lists themselves cannot appear in modules. Also, module local definitions cannot be exported.

Note that the keyword ALL may be used to explicitly declare that a program component is accessible in every module. ALL is equivalent to a missing KNOWS list.

12.3 KNOWS LIST (Continued)

The syntax of a KNOWS list is:

```
|                 _______ , ______
|                /                \
\__ { ____\__module:ident__/____ } _____KNOW_____________
           \                    /       \        /        \
            \_______ALL________/         \_KNOWS_/         |
                                                           |
```

The enclosure symbols for {...module:ident...} are curly braces. The use of parentheses here will cause a syntax error.

12.4 MODULE INTERFACE DESCRIPTION EXAMPLE

```
compiler
PROC
{ ALL } KNOW
CONST
    max_name_length = 10,
    symbol_table_size = 40,
    data_area_size = 100;


{ scanner, parser, codegen } KNOW
DEC
   TYPE
        RESERVED_WORDS = SYMBOLIC ( proc, corp, for, var),
        SYM_TAB_RANGE  = 1..symbol_table_size,
        SYMBOL_TABLE_ENTRY =
                STRUC
                    name STRING (max_name_length),
                    CASE is_keyword BOOLEAN
                       IS true: word RESERVED_WORDS
                       OR false:location 1..data_area_size
                    ESAC
                CURTS;

    sym_tab_block
    DATA
        symbol_table ARRAY [SYM_TAB_RANGE]
                           OF SYMBOL_TABLE_ENTRY,
        next_entry SYM_TAB_RANGE;
CED;

{ scanner, parser } KNOW
TYPE
    SYMBOLS = SYMBOLIC ( ident, number, reserved_word),
    TOKEN   = STRUC
```

12.4 MODULE INTERFACE DESCRIPTION EXAMPLE (Continued)

```
                        CASE type SYMBOLS
                           IS ident, reserved_word:
                                 table_entry SYM_TAB_RANGE
                           OR number: value 0..99
                        ESAC
                    CURTS;


{ parser, codegen } KNOW
TYPE
    ARITH_INST  = SYMBOLIC ( inc, add, dec, sub, mpy, div ),
    BRANCH_INST = SYMBOLIC ( lss, eql, leq, gtr, neq, geq ),
    LABELS      = 0..99;


{ parser } KNOWS
scanner
MOD
    get_token PROC RETURNS TOKEN;
{ driver } KNOWS
    get_next_symbol PROC;
DOM;


{ driver } KNOWS
parser
MOD
    parse PROC;
    initialize_parser PROC;
DOM;
```

## 12.4 MODULE INTERFACE DESCRIPTION EXAMPLE (Continued)

```
{ parser } KNOWS
codegen
MOD
    generate_arith PROC (opcode ARITH_INST,
                         addr1 SYM_TAB_RANGE,
                         addr2 SYM_TAB_RANGE,
                         addr3 SYM_TAB_RANGE );
    generate_branch PROC (br_opcode BRANCH_INST,
                          label LABELS);
    generate_label  PROC (label LABELS );
DOM;

{ driver } KNOWS     % NOTE: a KNOWS list containing the
driver               % modules's own name prevents any other
MOD                  % module from accessing it. This may be
    start ENTRY;     % desirable for the module containing
DOM;                 % the program ENTRY point, as is the
                     % case here.
GORP
```

## 13 DATA TYPES

Data types describe classes of values and permissible operations. There are four basic categories of data types: simple, aggregate, pointer, and procedure pointer. Some data types may be modified. Type constructors are used to define more complex types (aggregate types) in terms of others. SPRITE is a strongly typed language; each variable has an associated type.

### 13.1 MODIFIED DATA TYPES

Basic data types may be modified to allow the programmer control over their run-time representation. Modified data types may be defined in the MID or in SPRITE modules that are declared ENV_DEPENDENT (see 12.2) in the MID. Non-ENV_DEPENDENT SPRITE modules may access MID data blocks declared with variables of modified or non-modified types. The type modifiers are PACKED, DISPLAY, and MODULO. Use of the modified types makes the module (or program) dependent on a particular machine or environment for its proper operation.

#### 13.1.1 PACKED Modifier

The PACKED type modifier has two basic purposes:

1) to direct the compiler to choose a minimal space representation for the type, and

2) to allow the actual representation of the type to be specified.

The use of the PACKED modifier shifts responsibility for the representation of the type from the compiler to the programmer, at some cost in code efficiency, safety, and ease of modification. Such a shift is justifiable where:

1) it is necessitated by the existence of predefined external interfaces, e.g., a machine data format (result descriptor), or an interface controlled by mechanisms outside the language (BCT formats).

2) Data space must be conserved, even at the expense of code space and time, e.g., in a large data base.

## 13.1.1 PACKED Modifier (Continued)

NOTE: PACKED STRUCs are not ENV_DEPENDENT unless they use a construct in their definition that is. (See 13.3.3).

Examples:
```
     TYPE
          PKD_STRUC = PACKED STRUC ch CHAR, b BOOLEAN CURTS,
                      % PKD_STRUC is not ENV_DEPENDENT!

          TAG_STRUC = STRUC CASE BOOLEAN
                            IS true:  ch_str STRING (10) OF EBCDIC
                            OR false: hx_str STRING (20) OF HEX
                            ESAC
                      CURTS,
                      % TAG_STRUC is ENV_DEPENDENT

          PKD_TAG   = PACKED STRUC a BOOLEAN,
                                   CASE a
                                   IS true:  v1 T1
                                   OR false: v2 T2
                                   ESAC
                            CURTS;
                      % PKD_TAG is ENV_DEPENDENT!
```

## 13.1.2 DISPLAY Modifier

The DISPLAY type modifier is used to specify an internal representation for non-negative integers. In particular, the use of characters (see 13.2.2.1).

## 13.1.3 MODULO Modifier

The MODULO type modifier specifies the modulo boundary at which a data object is to be aligned.

The syntax for the MODULO construct is:

```
 type
 |
 \_____ MODULO ___ integer ______ non-mod-type ____________
       \______________________/                            \
                                                            |
                                                            |
```

where non-mod-type is an indicant or any type which does not start with "MODULO" (e.g. "VAR junk MODULO 4 MODULO 2 BOOLEAN" is incorrect). If non-mod-type is an indicant, you may define that indicant either with or without its own MODULO requirement.

13.1.3 MODULO Modifier (Continued)

The integer must be an integer literal in the range 1..9999. When generating ICMs for use by BINDER, this integer will be restricted to 2 or 4 (this restriction does not apply when the MCPVI option is set).

Whenever the MODULO construct is specified, the resulting modulo is the least common multiple (LCM) of the specified modulo value and the existing modulo of the modified type. Thus, the modulo for "MODULO 3 EBCDIC" would be 6. This means that **modulos can never be lowered** by using the MODULO construct. The modulo of an aggregate (a structure or data block) is the LCM of the modulos of all its components. For example, the modulo of

```
STRUC x MODULO 3 HEX, y MODULO 5 HEX CURTS
```

would be 60 (remembering that the default modulo of a STRUC is 4). This example illustrates that the user of oddball modulos will pay a space penalty.

It is an error if the updated modulo value of a stack-relative item exceeds 4, or if the updated modulo value of any other item exceeds 9999.

Items with the same STRUC base type, but with different modulos, are compatible.

For example,

```
TYPE  BOOLEAN_MOD_4  =  MODULO 4  BOOLEAN;
junk
DATA
strange_bit  MODULO 2  BIT;

TYPE  INTERFACE  =
STRUC
first_thing                BOOLEAN
strange_thing  MODULO 4    0..3
other_stuff                STRING (8) OF HEX
CURTS
VAR  x              INTERFACE,
y  MODULO 8  INTERFACE;    %  x and y are compatible
```

## 13.2 SIMPLE DATA TYPES

Simple data types are finite scalars, subranges of scalars, and reals. A single range of values is associated with a simple type. This range of values may be ordered or not. If the values are ordered, the first value in the range is the lowest; subsequent values are monotonically increasing.

### 13.2.1 Finite Scalar Types

The finite scalar types are: booleans, symbolic ranges, integer ranges, binary ranges, characters, bits, and hex. Arithmetic and comparison operators may be applied to integer range values. Logical and equality operators may be applied to boolean values. Only equality operators (=,¬=) may be applied to unordered ranges and characters. All of the comparison operators (i.e.,=, ¬=, <, <=, >=, >) may be applied to ordered scalars and strings of ordered scalars.

#### 13.2.1.1 Booleans

BOOLEAN is a predefined scalar type with two logical values: true and false. Boolean values are not ordered.

Boolean Operations

Boolean operators are ¬, &, &&, |, ||, #, =, and ¬=. Their semantics are described in Sections 19.1.1 and 19.1.2. Other operations which return boolean values are the IN and comparison operators (see 19.1.2). See also 14.1.2, for the &:=, |:=, #:= variations of the assignment statement.

Example:

```
     VAR    complete   BOOLEAN := false,
            not_bit    BOOLEAN;
                .
                .
                .
     WHILE ¢complete
         DO
            not_bit := true;
            IF    token = "NOT"
            THEN  not_bit := false;
                .
                .
                .
            FI
         OD,
```

Boolean Denotations

Boolean denotations are symbolic constants which are the reserved identifiers:

```
true
false
```

13.2.1.2 Bits

The type BIT is similar to BOOLEAN, but has a particular representation on the machine -- a single bit. BITs that are declared consecutively are allocated consecutive bits. The order of allocation within each digit is most significant bit first, i.e., 8, 4, 2, 1. Boolean operations and denotations are applicable to BITs, except that BITs may not be pointed to and may not be passed as VAR parameters in SPRITE procedure calls. BIT is compatible with BOOLEAN, but not equivalent.

13.2.1.3 Characters

CHAR and EBCDIC are predefined scalar types. Their range of values is the character set accepted by the implementation. EBCDIC has the collating sequence of the machine on which it is implemented; the rules concerning ENV_DEPENDENT (see section 13.1) apply to EBCDIC.

It is more convenient to use declared data as STRING (1) (see 13.3.2) instead of CHAR. A CHAR denotation, or source text representation, looks the same as a STRING (1) denotation; program context determines the interpretation.

Character Operations

CHAR values may be compared for equality only. The applicable operators are = and ¬=. All of the comparison operators may be applied to EBCDIC.

Character Denotations

A character denotation has the form:

```
"<char>"
```

The quote character (") itself is represented as two consecutive quotes.

### Character Denotations (Continued)

Examples:

```
    "X"
    "a"
    "{"
    """"
```

## 13.2.1.4 Integers

There is no predefined integer scalar type. Subranges (see 13.2.2) of the integers may be defined. These provide a machine independent description of integer data. No integer may exceed its defined maximum or minimum range.

Example:

```
    TYPE INT = -999..999
```

INT, a user composed indicant name, describes the range of integer value that are expressible in three decimal digits and a sign.

### Integer Operations

The monadic operators + and - and the dyadic operators REM, *, /, +, -, <, >, <=, >=, = and ¬= may be applied to integer values. (See also 14.1.2, for +:=, -:=, *:=, /:= ). Their semantics are described in Sections 19.1.1 and 19.1.2. The standard operator 'abs' may be used to determine the absolute value of numeric data.

Examples:

```
    count +:= 1
    index_init := 5 * 2 + 9
    new_numb := abs (int_var)/10
```

### Integer Denotations

An integer denotation is a numeric literal represented as a decimal digit sequence, optionally preceded by a sign ( + or - ).

Integer Denotations (Continued)

Examples:

```
-99
327
  0
```

13.2.1.5 HEX

HEX is a predefined, ordered scalar type ranging over the hexadecimal values 0 through F.

HEX Operations

The comparison operators ( =, ¬=, <, <=, >, >= ) and the logical operators (&, |, #, ¬) may be applied to values of type HEX and to equal length strings of HEX. Their semantics are described in section 19.1.2.

HEX Denotations

A HEX denotation is a character literal in the range "0" to "9" and "A" to "F". The character is coerced to the hexadecimal value it represents when context requires it.

13.2.1.6 Symbolic Ranges

A symbolic range consists of one or more symbolic values represented by distinct identifiers. A symbolic range description must be prefixed by ORDERED or SYMBOLIC. The syntax of a symbolic range description is:

```
|
|                          ___________ , ___________
|                         /                          \
|\__SYMBOLIC____  (  __\___symbolic:identifier___/__  )  _____
\___ORDERED__/                                                 \
                                                                |
                                                                |
```

The SYMBOLIC range is unordered. If ORDERED appears, the value of an identifier is 'less than' the values of the identifiers succeeding it and 'greater than' the values of identifiers preceding it. Symbolic range identifiers may not be used in defining another symbolic range; all symbolic ranges must have unique identifiers. However, if the range is ORDERED, they may be used to define a subrange. No two symbolic ranges are equivalent or compatible; all are unique.

13.2.1.6 Symbolic Ranges (Continued)

Examples:

```
TYPE
     STATE =  SYMBOLIC (executing, waiting, stopped),
     DAY = ORDERED (mon, tue, wed, thur, fri, sat, sun),
     MONTH = ORDERED (jan, feb, mar, apr, may, jun, jul,
                      auc, sept, oct, nov, dec);
```

<u>Symbolic Range Operations</u>

Symbolic range values may be compared only for equality ( = and ¬= ). When the range is ORDERED, all of the comparison operators ( =, ¬=, <, <=, >, >= ) may be used. A symbolic range value may appear as the left operand of the IN operator (see 19.1.2).

<u>Symbolic Range Denotations</u>

A symbolic range denotation is a symbolic range value (identifier) chosen from the symbolic range description.

Examples:

```
sat
executing
```

13.2.1.7 PACKED Symbolics

The syntax for PACKED symbolics is:

```
|                        ------------ , -------------
|                       /                            \
\_PACKED__SYMBOLIC__ (  _\__   __packed scalar value__/_ ) __
        \_ORDERED_/      \ \1/                      /       \
                          \------------------------/        |
                                                            |
                                                            |
```

13.2.1.7 PACKED Symbolics (Continued)

```
packed scalar value
|
|
|
\___scalar:identifier_______________________________________
                      \                               /       \
                       \__ = ___constant:expr___/           |
                                                              |
                                                              |
```

A PACKED symbolic is used to assign specific, internal representations to each of the scalar values. PACKED SYMBOLIC types are not ordered; PACKED ORDERED types are ordered.

The internal representation of the scalar value is defined by a succession rule or an optional constant expression. The first scalar is represented by the value 0. Succeeding scalars are represented by consecutive hexadecimal values. Use of the optional expression overrides the succession rule. The hexadecimal equivalent of the expression's value is used and the succession rule resumes from that value until another expression overrides it. Successive commas increment the internal value without assigning it to a symbolic name.

The override expression may be either an integer constant or a HEX string. If it is an integer constant, the value is converted to its hexadecimal equivalent.

Examples:

```
    TYPE ENTRY_POINTS = PACKED ORDERED
                 (read_entry,          % hex value of  0
                  write_entry,         % hex value of  1
                  error_entry =  9,    % hex value of  9
                  abort_entry,         % hex value of  A
                  stop_entry  = 20),   % hex value of 14

         CNTLS = PACKED ORDERED (nul,, stx, etx, eot,
                                   ff = "C", cr);
         % nul=0, stx=2, eot=4, cr=D in internal hex
```

The scalar occupies as many four-bit digits as are necessary to contain the maximum internal value used.

## 13.2.2 Subrange Type

Integer subranges and symbolic subranges may be defined. The values of a subrange are ordered. To describe a subrange of a symbolic range the latter must be ORDERED. Subranges of BIT, CHAR, EBCDIC, REAL, HEX, BOOLEAN or PACKED symbolic type may not be defined. The type description specifies, in order, the minimum and maximum values in the subrange. The syntax of a subrange description is:

```
|
|
\__constant:simple expr__ .. __constant:simple expr______
                                                          \
                                                          |
                                                          |
```

The types of the expressions must be equivalent. This type is the scalar on which the subrange is defined (the "parent"). The bounds of a subrange description are evaluated at compile-time. The subrange includes all values of the parent scalar within the given bounds. Either limit may be signed, but the lower bound must be algebraically less than or equal to the upper bound in the ordering of the parent type.

Subranges that have the same parent are compatible. The relationships between compatible subranges are disjointness, overlap, and containment. These relationships determine type restrictions.

Examples:

```
0..max_size     % describes the range of integer values:
                % 0,1,2,...,max_size.

-3..3           % describes the range of integer values:
                % -3,-2,-1,0,1,2,3.

mon..fri        % describes the range of weekdays mon
                % through frigiven the ORDERED symbolic
                % range in 13.2.1.6.
```

### Subrange Operations

The operators that may be used on subrange values are the same as permitted on the parent type. A subrange type range inquiry or subrange expression may appear as the right operand of the IN operator. The left operand must have the type of the parent scalar or a compatible

Subrange Operations (Continued)

subrange.

Examples:

```
    VAR i, j, k  INT;
    IF i IN  j..2*k
```

Subrange Denotations

A subrange denotation is the same as a denotation of the parent type.

Examples:

```
    25
    -3
    thur      % see type DAY in 13.2.1.6
```

13.2.2.1 DISPLAY Integers

The syntax for DISPLAY types is:

```
|
|
\___DISPLAY____integer subrange:finite scalar type_______
                                                         \
                                                         |
                                                         |
```

The DISPLAY modifier is used to describe non-negative integers that are to be stored as numeric characters. Any arithmetic operation may be performed on DISPLAY items. They may be coerced to non-display numbers and vice versa. They may be coerced to strings and vice versa. A common use of the DISPLAY modifier is to provide a transitional data type for the conversion of a numeric character string to a number and back.

Example:

```
    TYPE
         CHAR_NUM    = STRING (5) OF CHAR,
         FDNUM       = 0..99999,
         DISP_FDNUM  = DISPLAY FDNUM;
```

13.2.2.1 DISPLAY Integers (Continued)

```
        VAR in1, in2 STRING(5) OF CHAR,
            output   CHAR_NUM,
            result   FDNUM

        result  := FDNUM (DISP_FDNUM (in1)) + FDNUM (DISP_FDNUM (in2)
                             % casts string to number
                             % (if possible), to allow
                             % addition of numeric chars
        output  := CHAR_NUM (DISP_FDNUM (result));
                             % casts number to string of
                             % numeric chars for printing
```

13.2.2.2 Binary Integers

The syntax for binary types is:

```
!
!
\____ BINARY __ ( __number_of_bits: integer expr__ ) ________
                                                             \
  ----------------------------------------------------------/
 /
 \___________________________________________________________
   \              lwb:                        upb:     /    \
    \_ RANGE ____integer expr_____ .. ___integer expr_/     !
                                                            !
                                                            !
                                                            !
```

The binary type is used to describe non-negative integers which are expressed in binary rather than decimal. The first integer expression specifies the number of bits the type is to occupy, which must be evenly divisible by four i.e., 4, 8, 24, 52.

If no RANGE part is specified, the bounds of the type default to 0 and (2 ** number_of_bits) -1. When a range is specified, its upper and lower bounds must fall within the default range. The maximum range allowed is (10**16-1). Therefore the maximum is 56 bits. This maximum is checked.

Binary integers are ordered. All arithmetic and comparison operators may be applied to them. Binary integers may be converted to decimal integers and vice versa. They may be mixed in the same expression.

## 13.2.3 *Reals

REAL and LONG_REAL are predefined simple types. Values of these types are floating point numbers and are ordered. The range of values (i.e., size of the exponent and fractional part) is implementation dependent.

### Real Operations

The operators for REAL values are the same operators (excluding REM) as for integers.

### Real Denotations

A REAL denotation has the form:

```
|
|
\__integer___ . ___integer_________________________________
                         \_E___________integer___/        \
                              \__ + __/                    |
                               \_ - _/                     |
```

The meaning of the 'E' preceding the exponent is 'times ten to the power of'. No blanks are permitted in the denotation.

Examples:

```
 1.732
-3.0
 0.01
 3.2E-2
+1.0E5
 3.1415927
```

## 13.3 AGGREGATE TYPES

Aggregate types are characterized by the component type(s) and the method of organization. The aggregate types are arrays, strings, structures, sets and translate_tables. An aggregate type indicant must not appear as a component in its own definition. However, if T is the aggregate type being defined, the type PTR TO T may appear in the definition of T.

13.3.1 Arrays

An array is an n-dimensional collection of components of the same type. The component type may itself be an aggregate type. A component is selected by subscripting. The syntax of an array description is:

```
|                     _________ , ________
|                   /          index:      \            component:
\_ARRAY__ [ __\__finite scalar type__/__ ]  ________type___
                                              \_OF_/          \
                                                               |
                                                               |
```

where the index type is any finite scalar type. That is, the index is a scalar, possibly unordered, with a finite number of values. Equivalence is required of arrays in all contexts. An array may be cast from a compatible type to obtain equivalence. Compatible arrays have equivalent component types, the same number of indices, compatible corresponding indices, and the same number of elements in each index. The number of indices represents the dimensionality. The number of dimensions must be in the range 1..10.

Examples:

Given the types DAY, MONTH (defined in 13.2.1.6) and INT, the declaration

```
        ARRAY [DAY] OF INT
```

describes a vector of seven objects of type INT. The following declarations are some possible equivalent representations of 'years'.

```
        ARRAY [MONTH] OF ARRAY [1..5, DAY] OF INT
        ARRAY [MONTH] OF ARRAY [1..5] OF
            ARRAY [DAY] OF INT
        ARRAY [MONTH, 1..5, DAY] OF INT
```

Array Operations

Equivalent arrays may be compared for equality (= and ¬=). The operators allowed on array elements are those permitted on the element type.

Array Denotations

An array denotation has the syntax:

```
|
|            ____________ , ____________
|           /                            \
\___ [ __\___element value:expression___/__ ] ___________
                                                          \
                                                           |
                                                           |
```

Context determines the denotation's type. If necessary, the element values are coerced to the array component type. The number of values must be identical to the number of elements in the array.

Examples:

```
    [0,0,15,3,8]                  % denotes an array of five
                                  % integer elements.

    [[1,0,0],[0,1,0],[0,0,1]]     % denotes an array which
                                  % is the 3x3 identity
                                  % matrix.
```

Subscripts

A variable may select an element of an array by subscripting. This element may be any defined type including another array or a structure. The syntax of a subscripted variable is:

```
|
|                              ______ , ______
|                             /   subscript:   \
\__array:primary____ [ __\______expr______/__ ] _________
                                                          \
                                                           |
                                                           |
```

The type of the subscript expression must be compatible with the corresponding index type in the array description. The number of subscript expressions must be the same as the number of declared indices. The order in which subscripts are evaluated is undefined.

## Slices

A subarray of an array may be selected by slicing. Slicing selects part of an array instead of a single element. A slice may only be applied to the last dimension of an array. The syntax of a slice is:

```
|
|   array:        start index:          stop index:
\__primary__ [ __simple expr_____ .. ___simple expr_____ ] _
                                 \                     /        \
                                 |      numb elems:    |        |
                                 \__ :: __simple expr__/        |
                                                                |
                                                                |
```

The form [i..j] means that the slice selects all of the array elements from index i through index j. The resulting array contains (j - i)+1 elements. Its type is ARRAY [i..j] of <elem type>.

* The form [i::j] means that the slice selects j consecutive array elements beginning with the element at index i.

The bounds of the slice must be constant. (The object of a FIND statement may be an array slice with variable bounds).

* The bounds of the slice may be variable.

Examples:

```
TYPE SEVERITY =  SYMBOLIC (clear,mild,severe,hazardous);

VAR  x1 ARRAY [1..5] OF REAL,
     x2 ARRAY [1..30, 0..3] OF INT,
     x3 ARRAY [1..10] OF REAL,
     smog ARRAY [DAY] OF SEVERITY;

x1 [3] := x2 [30,0] + x2 [i, max(j, 2*n) + 1];

smog [thur] := hazardous;

x2 [1,0..3] := [0,0,0,1];   % x2 [1,0] := 0; x2 [1,1] := 0;
                            % x2 [1,2] := 0; x2 [1,3] := 1;

x3 [1..5] := x1;            % x3 [1] := x1 [1];
                            % x3 [2] := x1 [2];
                            % x3 [3] := x1 [3];
                            % etc.
```

## 13.3.2 Strings

A string is a sequence of characters. STRING is a type constructor much like ARRAY. The syntax is:

```
|
|                  integer
|                  constant:
\__STRING__ ( __simple expr__ ) ____________________________
                                 \        element:string  / \
                                  \__OF____base type___/   |
                                     \__/                  |
                                                           |
                                                           |
```

```
string base type
|
|
|\__HEX___________________________________________________
|                                                         \
|\__CHAR_________________________________________________ |
|                                                        \|
\___EBCDIC_______________________________________________ |
                                                         \|
                                                          |
                                                          |
```

The length of the string is specified by a constant expression with a positive integer value. Strings may be any length up to 499,999 bytes for characters and up to 999,999 digits for strings of HEX.

CHAR is the default STRING base type.

Strings of the same length and base type are equivalent. Strings of compatible base types (and possibly different lengths) are compatible.

13.3.2 Strings (Continued)

Examples:

```
STRING (132)

STRING (80) OF HEX
```

String Operations

Strings may be compared for equality with = and ¬=. The relational operators ( <, >, <=, >= ) may be applied to STRINGs of EBCDIC or HEX to test order. If the string operands are compatible, but differ in length, the shorter is coerced to the length of the longer.

The logical operators (¬, &, |, #) and their corresponding assignments (&:=, |:=, #:=) may be applied to STRINGs of HEX. They perform the usual bit-wise operations on their operands. There are two restrictions placed on the operands.

1. Both operands must be of the same length

2. Variable length strings must be 100 digits or less in length

If either of these two conditions is violated, a compile time or run time error, as appropriate, will be generated.

Strings may be concatenated with the plus operator (+). The length of the result is the sum of the lengths of the operands.

Strings and displays may be concatenated. In this case, the compiler will coerce the display to a string of its own length with the base type of the counterpart string.

The standard functions 'index', 'index_any', 'index_inc' and 'index_none' (see Appendix A) may be used to test the values of strings. The standard function 'length' (see Appendix A) may be used to determine the size of a string, including parametric strings (see 15.3).

String Selection (Substrings)

A string selection is similar to an array subscript. It returns a specified substring when used as a part of an expression or a reference to the specified substring (on the left hand side of a string assignment).

String Selection (Substrings) (Continued)

A substring may be extracted from a string by the operators :: or .. (*) as follows:

```
|
|  string:          start index:
\__primary__ [ __simple expr__________________________ ] __
                              \       numb elems:   /        \
                              |\_ :: _simple expr__/|        |
                              |                     |        |
                            * |        stop index:  |        |
                              \__ .. __simple expr__/        |
                                                             |
                                                             |
```

The simple expressions must be of an integer subrange type with values greater than zero.

The form [i::j] means that the substring to be extracted begins at the ith element of the string and is j elements long.

* The form [i..j] means that the substring to be extracted begins at the ith element and includes all subsequent elements through the jth element.

The form [i] produces a single element of the base type of the string. A string of length 1 can be extracted by [i::1] or [i..i] .

Substring selection may follow any string expression.

String Selection (Substrings) (Continued)

Example:

```
    buffer [73::8] := " ";
    text := card [1::72];
    result [i::j] := descriptor [k..l] [1..n];
                     % descriptor is a string.
                     % descriptor [k..l] is a substring
                     % from position k through position
                     % l of descriptor.
                     % descriptor [k..l] [1..n] is a
                     % substring of that substring from
                     % position 1 through n.

VAR string STRING (6)  := "ABCDEF",
    str    STRING (3),
    ch     CHAR;

str := string [2::3];      % str contains the substring "BCD"

str := string [2::3] [3]; % str contains the string "D  "
                           % where string [2::3] is a substring
                           % of string, string [2::3] [3] is a
                           % string element of string [2::3]

str := "abcd" [2::1];      % str contains the string "b"
```

String Conversions

Truncation shortens a string to the length of the destination string and checks that the truncated characters are all 'blanks'. 'Blankfill' lengthens a string to the length of the destination string by inserting 'blanks'. Truncation and blankfill both occur on the right end of the string. The 'blank' character depends on the string element type:

| element type | blank |
|---|---|
| CHAR | " " |
| EBCDIC | " " |
| HEX | hex zero (0) |

String Denotations

A string denotation has the form:

String Denotations (Continued)

```
|                              _________________
|                             /                 \
\______________ " __\____character____/__ " ______________
                                                          \
                                                          |
                                                          |
```

The type of a string denotation is STRING (length) OF EBCDIC.

The quote is represented within a string denotation by two adjacent quote symbols. There is no empty string since a STRING must have a positive length; i.e., "" is an illegal string.

A single character in quotes is type STRING (1) OF EBCDIC. The coercion STRING (1) OF EBCDIC to EBCDIC (or CHAR) occurs whenever context requires it.

Examples:

```
"?"
"This is a string."
"xyz"
"""Hello,"" he said."
```

(See also 13.6 and 15.3, parametric types and parametric type definitions).

13.3.3 Structures

A structure is a collection of components called 'fields'. A field may be of any type, including another aggregate type. The associated identifier names the field. Separately described structure types are not equivalent or compatible. (See 13.9).

The syntax of a structure description is:

```
|
|                                   ____ , ____
|                                  /           \
\______________STRUC_____\___field___/_____CURTS_________
   \_PACKED_/                                            \
                                                         |
                                                         |
```

13.3.3 Structures (Continued)

field

```
|
|         _______ , ________
|        /    field name:   \
|\_______\____identifier____/____type____________________
|                                                        \
\___CASE___tag field___IS___variants___ESAC______________ |
                                                         \|
                                                          |
                                                          |
```

The Predefined Identifier "filler" denotes an unused field. It may be used as an identifier anywhere in a structure definition or a data declaration (see 18.1). The parts of the structure or the fields in the data definition identified by "filler" cannot be referenced. "filler" may be used any number of times in a given structure definition or data definition.

Use of the Predefined Identifier "filler" outside of structure definitions or data blocks will cause syntax errors.

Example:
```
     TYPE  JUNK  =
                     STRUC
                          first_part   0..99               ,
                          filler       STRING (4) OF HEX   ,
                          goodies      BOOLEAN             ,
                          filler       CHAR                ,
                          filler       0..9999999
                     CURTS
```

Reminder: The compiler generates its own internal fillers (or pads) as needed. In the above example, it would allocate 1 digit after "goodies" to put the CHAR at a mod 2 address, and it would allocate 3 digits after the last "filler" to make the size of the structure mod 4.

Variants

Variants (Continued)

The syntax for tag field is:

```
tag field
|
|
|\____normal tag______________________________________
|                                                      \
|\____omitted tag____________________________________  |
|                                                     \|
\_____disjoint tag___________________________________  |
                                                      \|
                                                       |


normal tag
|
|      tag id:              tag type:
\______identifier_________finite scalar type_____________
                                                         \
                                                         |
                                                         |


omitted tag
|
|      tag type:
\______finite scalar type________________________________
                                                         \
                                                         |
                                                         |


disjoint tag
|
|      tag id:
\______identifier________________________________________
                                                         \
                                                         |
                                                         |
```

Use of omitted or disjoint tags requires that the type be defined in an ENV_DEPENDENT module or the MID. This applies whether the declaring structure is PACKED or unpacked. (See 13.1.1).

Variants (Continued)

The syntax for variants is

```
|
|  _____________________OR___________________________
| /   ________ , _________                           \
| |  /                    \                          |
| |  |  variant label:     |                         |
\_\__\__value collection__/__ : ______________________/___
                                 \  ______ , _____  /     \
                                 | / variant alt: \ |     |
                                 \_\____field_____/_/     |
                                                          |
                                                          |
```

value collection

```
   |
   |
   \________expr__________________________________________
*                \____ .. ____simple expr____/           \
                                                          |
                                                          |
                                                          |
```

The list of variant alternative fields may be empty.

A variant label is a constant expression whose type is compatible with the tag field type. If the type is ordered, a subrange of labels may be used (*). The variant part groups several mutually exclusive alternatives. Each alternative is a field list. The value of the tag field specifies a particular variant alternative field list. The tag field of a variant part may be any finite type. That is, it may be BOOLEAN, symbolic, subrange, CHAR, EBCDIC, BINARY, HEX or SET (*). The variant labels are possible values of the tag field. There need not be a variant alternative for each possible value of the tag field. The tag field can be accessed like other fields. Assignment to a normal tag field must precede assignments to any field of the associated field list. Normal tag field assignment will set the fields in the variant part to an uninitialized state.

* References to a variant alternative field cause a check of the tag field to ensure that the referenced field is in the active variant. If the tag field does not have the value of the variant whose field is referenced, a run-time error results.

Variants (Continued)

The variant part allow selection of fields by case and may be one of three forms: normal, disjoint tag or omitted tag. In the normal form, the tag field is allocated as the first field in the variant part. If the tag field type is disjoint, the tag field identifier must be declared previously in a non-variant part of the structure. For example:

```
    PACKED STRUC
                 b BOOLEAN,
                 r REAL,
                 CASE b                  % variant part with
                 IS true:  i   1..10     % disjoint tag
                 OR false: j -10.. 0
                 ESAC
           CURTS
```

If the tag field identifier is omitted, no tag field is allocated and no tag field validity checking is performed. The type is used to provide context for the variant labels. For example:

```
    PACKED STRUC
                 CASE BOOLEAN    % omitted tag
                 IS true:   numeric_string ALFA
                 OR false:  ordinal        NUM_ALFA
                 ESAC
           CURTS
```

Examples:

```
STRUC                                   % describes  a  structure
    re,im REAL                          % of two REAL fields named
CURTS                                   % 're' and 'im' representing
                                        % a complex number.

STRUC                                   % describes a structure of
    year  1..2099,                      % three   integer   fields
    month 1..12,                        % representing a Gregorian
    day   1..31                         % date.
CURTS


IDCLASS = SYMBOLIC (const,ident,varbl,proc);

STRUC                                   % describes  a   structure
    name    STRING(10),                 % which might be used in a
    idtype SYMBOLIC (int,bool,real),    % compiler: it groups
    CASE class IDCLASS                  % together information
```

Variants (Continued)

```
    IS const: values VALUE              % about an identifier.
    OR varbl: vkind SYMBOLIC (actual,formal),
              vaddr   ADDR_RANGE,
              initval VALUE
    OR proc:  CASE pkind DECLKIND
              IS  standard : key 1..10
              OR  declared : paddr CODERANGE
              ESAC
    ESAC
CURTS

STRUC
    CASE  b BOOLEAN
    IS true:  r REAL
    OR false:
    ESAC
CURTS
```

Structure Operations

Structures of the same type may be compared for equality with = and ¬=. The operators allowed on structure fields are those permitted on values of the field's type.

Structure Denotations

A structure denotation has the syntax:

```
|
|                 ____________ , _____________
|               /                              \
\_______ [ __\_______field value:expr_______/___ ] _______
                                                           \
                                                            |
                                                            |
```

Context determines the type. If necessary, the field values are coerced to the required type. The expression corresponding to a tag field must be constant.

For a structure denotation whose type has an omitted tag, a value representing the type which provides context must be included.

Examples:

```
[3777, nil, true]                    % denotes a structure of
                                     % three fields: integer,
```

Structure Denotations (Continued)

```
                                        % pointer, and boolean.

[3,[1.0,5.0,2.5],"*",{mult,div}]        % denotes a structure of
                                        % four fields: integer,
                                        % array of 3 reals,
                                        % character, and a set.
```

Field Selection

A variable may select a field of a structure. The syntax is:

```
|
|
\_____structure:primary_____ . _____field:identifier______
                                                          \
                                                           |
                                                           |
```

Examples:

Given:

```
    VAR     new_ref    INTER_PROC_REF
```

Where:

```
    TYPE    INTER_PROC_REF = PACKED
        STRUC
            CASE   destination   J_TARGET
            IS     intr_dest : dest_block     BLOCK_NUM,
                               dest_label     LABEL_NUM
            OR     extr_dest : external_ref   EXTERNAL_NUM
            ESAC
        CURTS
```

Hence the following field selections are valid:

```
    new_ref.dest_block
    new_ref.dest_label
    new_ref.external_ref
```

## 13.3.4 Sets

A set is a group of discrete values, which are called the elements of the set. A set value may include from zero to 256 elements. The syntax of a set description is:

13.3.4 Sets (Continued)

```
|
|
\____SET________________base:finite scalar type____________
          \__OF__/                                         \
                                                           |
                                                           |
```

The base type specifies the type of the set elements. The base type is any finite scalar type. That is, the base type is a scalar type, possibly unordered, with a finite number of values. The maximum number of elements the base type may contain is 256. The possible number of distinct elements in a set is restricted to the number of values in the base type.

Examples:

```
SET OF SYMBOL
SET OF 1..10
```

Equivalence is required of sets in all contexts except a cast.

* A set may be converted to a compatible set (i.e., with a compatible base type). Conversion is possible only if the cast base type 'includes' the base type of the argument.

Elements of a set must all be type equivalent. The base type of a set value must be provided by context. The left operand of the set IN operator must be equivalent to the base type of the right operand.

Set Operations

Set operators are the monadic operator - and the dyadic operators *, +, -, #, =, ¬=, >, >=, <, and <=. Their semantics are described in Sections 19.1.1 and 19.1.2.

Both operands must have equivalent base types. A set may be the right operand of the IN operator. The left operand is a scalar value of the set's base type.

Set Denotations

Set denotations have the form:

Set Denotations (Continued)

```
|
|             ______________ , __________________
|            /                                    \
\__ { ___\___expression___________________________/____ } __
          \                 \_ .. ___expression__/  /        \
           \_______________________________________/         |
                                                              |
                                                              |
```

The type of a set denotation is determined by context. The elements are coerced to the base type if necessary. The elements may be variable or constant. The empty set is {}.

* The range of values may be used if the base type is ordered.

Examples:

```
    {slash, "?", ch, CHAR (id [5::1]) }
    {prime (i), 2*i}
    ch IN {"a", "b", "c", "x", "z"}
    {tues}
    {}
```

## 13.3.5 Translate_tables

TRANSLATE_TABLE is a special purpose predefined indicant name. It is an aggregate type indicant name. TRANSLATE_TABLE is used only in constructing tables for use with the 'translate' standard function. The actual representation of this table is defined by the underlying machine, and imposes certain restrictions on the creation and use of objects of type TRANSLATE_TABLE.

Translate_table Operations

Nc operations other than assignment are defined for objects of type TRANSLATE_TABLE.

Translate_table Denotations

Denotations for objects of type TRANSLATE_TABLE do not exist. However, a constant of type STRING (256) OF EBCDIC may be coerced to type TRANSLATE_TABLE in initializations only.

Set Denotations (Continued)

```
|
|
|          _______________ , ____________________
|         /                                      \
\__ { ___\___expression__________________________/____ } __
        \                 \_ .. ___expression__/  /        \
         \______________________________________/          |
                                                           |
                                                           |
```

The type of a set denotation is determined by context. The elements are coerced to the base type if necessary. The elements may be variable or constant. The empty set is {}.

* The range of values may be used if the base type is ordered.

Examples:

```
{slash, "?", ch, CHAR (id [5::1]) }
{prime (i), 2*i}
ch IN {"a", "b", "c", "x", "z"}
{tues}
{}
```

13.3.5 Translate_tables

TRANSLATE_TABLE is a special purpose predefined indicant name. It is an aggregate type indicant name. TRANSLATE_TABLE is used only in constructing tables for use with the 'translate' standard function. The actual representation of this table is defined by the underlying machine, and imposes certain restrictions on the creation and use of objects of type TRANSLATE_TABLE.

Translate_table Operations

No operations other than assignment are defined for objects of type TRANSLATE_TABLE.

Translate_table Denotations

Denotations for objects of type TRANSLATE_TABLE do not exist. However, a constant of type STRING (256) OF EBCDIC may be coerced to type TRANSLATE_TABLE in initializations only.

Translate_table Restrictions

The following restrictions apply to the declaration and use of objects of type TRANSLATE_TABLE:

1) May only be declared in data blocks or as STATIC local variables; they may not be generated (via the GENERATE statement).

2) May not be used as a component of an aggregate type; they may be the referenced type of a pointer.

Example:

```
DATA

     % where e_to_a and a_to_e are constants
     % of type STRING(256) of EBCDIC

     ebcdic_to_ascii   TRANSLATE_TABLE := e_to_a,
     ascii_to_ebcdic   TRANSLATE_TABLE := a_to_e;


PROC;


     VAR  a   STRING(6),
          e   EBCDIC;
          .
          .
          .
     a := translate (e, ebcdic_to_ascii);
     e := translate (a, ascii_to_ebcdic);
          .
          .
          .

CORP
```

## 13.4 POINTER TYPES

A pointer is a reference to an object of some particular type. Pointers to BITs are not allowed. A storage level may be associated with the referenced type (*). The syntax is:

13.4 POINTER TYPES (Continued)

```
|
|
\_PTR_________________________________________referenced:type__
      \_TO_/ \__CONST__/ \__EXTERNAL__/                        \
             \__VAR__/    \__LOCAL___/                          |
                                                                |
                                                                |
```

CONST indicates that the pointer has read-only access to the referenced object; that is, the object's value may be used but not changed. VAR indicates that it has read/write access; that is, the object's value may be changed. The default access is VAR.

* The storage levels are:

| <u>Level name</u> | <u>meaning</u> |
|---|---|
| LOCAL | referenced object is in a temporary storage location |
| EXTERNAL | referenced object is in a static storage location local to the program |

A pointer may not reference data stored at a level lower (LOCAL is lowest) than the level associated with its referenced type. The default level is LOCAL.

* Equivalent pointers have the same level, the same access and equivalent referenced types.

Parametric STRING types may be used as the base type of pointer variables. When this is the case, the pointer variable may take on values which reference strings of different lengths as long as they are all compatible with the parametric type.

Examples:

```
PTR TO A_STRUCTURE               % describes a pointer
                                 % to a structure.

PTR TO ARRAY [1..5] OF BOOLEAN   % describes a pointer
                                 % to an array of BOOLEANs.
```

## Pointer Operations

Pointers of equivalent types may be compared for equality with = and ¬=. The standard function 'ptr' (Appendix A) returns a pointer to its parameter. No other operations are defined.

## Pointer Denotations

The only pointer denotation is the constant

    nil

'nil' is a reference to no value whatsoever. Its type is equivalent to any pointer type.

## Dereferencing

A variable may select the value referenced by a pointer. The syntax is:

```
|
\______pointer:primary______ @ ___________________________
                                                          \
                                                          |
```

A run-time error will occur if a dereferenced pointer is undefined or has the value 'nil'.

The function 'ptr' produces a pointer value; it is the inverse of dereferencing.

Examples:
DISPLAY subrange <==> subrange range check

```
1. TYPE LIST_PTR = PTR TO LIST;

   TYPE LIST = STRUC
                 CASE key NODEKIND
                   IS atom:  s STRING(10)
                   OR lists: head,
                             tail LIST_PTR
                             % LIST's description
                             % may not contain a
                             % field of type LIST,
                             % but may have a type
                             % PTR TO LIST,
                             % i.e. LIST_PTR
                 ESAC
               CURTS;
```

Dereferencing (Continued)

```
    VAR  pr PTR TO INT,
         i  INT := 1,
         j  INT;

         pr := ptr(i);

         pr@ +:= 1;
         j     := pr@;          % j has the value 2.



2. TYPE
      SEGMENT_DICTIONARY_ENTRY =
         STRUC
            segment_number                     STRING (2) OF HEX,
            overlay_call_count                 0..99,
            quick_overlay_addr_kd              1..3333,
            quick_overlay_status               STRING(1) OF HEX,
            low_order_disk_addr                3..99999,
            segment_size                       0..999999
         CURTS;

   TYPE
      SEGMENT_DICTIONARY = ARRAY [1..200] OF
                                   SEGMENT_DICTIONARY_ENTRY;

   TYPE
      COLD_START_INFO =
         STRUC
            mem_dump_addr                      1..555335,
            mem_size_in_pages                  1..1334,
            seg_dict_ptr                       PTR SEGMENT_DICTIONA
         CURTS;


      VAR   cold_start_info                    COLD_START_INFO;
      VAR   work6                              1..999999;

      work6:= cold_start_info.seg_dict_ptr@[2].segment_size/10000;
```

Initialization of pointer variables

Variables of type pointer (but not pointer to procedure) may be initalized at compile time. The address of the referent must be determinable at compile time. All rules apply that apply to the use of the ptr function. The following table shows the valid referent kinds for each

Initialization of pointer variables (Continued)

kind of pointer.

| POINTER KIND | REFERENT KIND (1) | (2) | (3) | (4) | (5) |
|---|---|---|---|---|---|
| (A) | YES | YES | n/a | n/a | n/a |
| (B) | YES | n/a | YES | YES | NO |
| (C) | YES | n/a | YES | YES | YES |

where

POINTER KIND

(A) Data block pointer variables

(B) STATIC pointer variables

(C) Stack pointer variables

REFERENT KIND

(1) Constants

(2) Data block variables (for the same block only).

(3) Data block variables (for the shared blocks only)

(4) STATIC variables in the same procedure only

(5) Stack variables in the same procedure only

For example,

```
VAR  junk       JUNK,
     junk_ptr   PTR TO JUNK           :=  ptr (junk),
     ptr_ten    PTR TO CONST 1..10    :=  ptr (10);
```

## 13.5 Procedure Pointer Types

A procedure pointer is a reference (pointer) to a procedure. Dereferencing a procedure pointer invokes the procedure which the procedure pointer references. This

13.5 Procedure Pointer Types (Continued)

allows run-time determination of the procedure which will be called at a given place in the program. The syntax of a procedure pointer description is:

```
|
|
\_____PTR____ TO ____PROC________________________________________________
           \______/          \                                          /
                              \_____________ attributes _________/
```

```
attributes
|
|
\_________________________________________________________________
      \    parameter      / \                      /    \
       \___descriptions___/   \___RETURNS___type___/     |
                                                         |
                                                         |
```

```
parameter descriptions
|
|          ___ , _________________________________
|         /  param: \                              \
\__ (  __\__ident__/__________________________type__/__ ) __
                  \___CONST___/  \_UNIV_/                  \
                   \___VAR___/                              |
                    \_VALUE_/                               |
                                                            |
                                                            |
```

For a description of parameters, access, univ and return type see Section 16.1, Procedure Definitions.

A procedure pointer description identifies a class of procedures that have equivalent parameter lists and return types. Procedure pointer descriptions may be used to declare a procedure pointer variable or to pass a procedure pointer as a parameter. A procedure pointer variable (or parameter) may only be assigned (or passed) a reference to a procedure that is defined or the pointer value nil. When the procedure pointer is dereferenced, the procedure that is the current value of the pointer is called.

Equivalent procedure pointers may be compared for equality using = and ¬=. The standard function 'proc_ptr' may be used to create a procedure pointer. No other operations

13.5 Procedure Pointer Types (Continued)

are defined on procedure pointers. A procedure pointer variable may invoke the referenced procedure. The syntax is:

```
|
|
\_proc ptr: primary_ @ ________________________________________
                      \                 ______ , ______         / \
                      |               /   actual param:\        | |
                      \__ ( __\__expression___/__ ) __/ |
                               \_________________/               |
                                                                  |
                                                                  |
```

Dereferencing a nil or undefined procedure pointer results in the same run-time errors as would be associated with a normal pointer in the same circumstances.

Examples:

1.
```
PTR TO PROC (float REAL, int INTEGER) RETURNS REAL
PTR TO PROC (flag BOOLEAN)
PTR TO PROC RETURNS INT
PTR PROC
```

2.
```
truncating
PROC (x REAL) RETURNS INT;
RETURN round (x);
CORP;

rounding
PROC (x REAL) RETURNS INT;
RETURN round (x + 0.5);
CORP;


example PROC;
VAR p PTR TO PROC (float REAL) RETURNS INT

IF do_rounding
THEN p := proc_ptr (rounding)
ELSE p := proc_ptr (truncating)
FI;

estimate := p@ (accurate-measure)
CORP;
```

## 13.6 PARAMETRIC TYPE CONSTRUCTORS

A parametric type constructor defines a family of related data types by allowing the size attribute of certain data types to vary, i.e., be parameters.

Every use of such a type must supply actual values in place of the formal parameters. The parametric type constructor is like a template from which many types can be obtained by supplying actual parameters for the formals.

When such a type is associated with an object, the actual parameter describing that object parameter is substituted for the formal. The result is a fixed type and the semantics of that fixed type apply.

Because its size varies depending on the actual parameter given, a parametric type may not be used to construct an aggregate type or a data block. After a parametric type is declared, that type indicant may only be used as a procedure parameter or as the object of a pointer.

Only one formal parameter is allowed in the definition of the parametric type. (See 15.3).

## 13.7 EQUIVALENCE, COMPATIBILITY AND COERCIONS

A type indicant acts as an abbreviation for its definition. After all such abbreviations have been replaced by their definitions, two simple data types are 'equivalent' if their expanded definitions have the same range of values. Other data types are equivalent if their corresponding component types are equivalent and their structuring methods are the same. (Exception: separately described STRUCtures are not equivalent, even if the separate descriptions are identical in all respects).

A cast is an explicit conversion of data to another type. A coercion is an automatic, implicit conversion of data. Data types are 'compatible' if one can be converted to the other by cast or coercion.

The following table lists the types in SPRITE which are compatible. Thus, coercions and casts can be performed for those types. It should be noted that SPRITE only performs single-step conversions, say from BIT to BOOLEAN. Multi-step conversions such as numeric to DISPLAY to STRING are not attempted. For multi-step conversions, casts must be explicitly specified by the user.

## 13.7 EQUIVALENCE, COMPATIBILITY AND COERCIONS (Continued)

| DATA TYPE | DI-REC-TION | DATA TYPE | NOTES |
|---|---|---|---|
| STRING(1) OF TYPE | <==> | TYPE | |
| BOOLEAN | <==> | BIT | |
| STRING(n) | <==> | STRING(m) | truncation or fill (n¬=m) |
| subrange | ==> | parent subrange | |
| | ==> | overlapping subrange | range check |
| | <==> | BINARY | range check |
| PTR VAR | ==> | PTR CONST | |
| PTR fixed string | ==> | PTR parametric string | |
| PTR parametric string | ==> | PTR fixed string | range check |
| PTR parametric string1 | ==> | PTR parametric string2 | range check |
| CHAR | <==> | EBCDIC | |
| STRING(1) CHAR | <==> | EBCDIC | |
| STRING(1) EBCDIC | <==> | CHAR | |
| STRING(n) EBCDIC | <==> | STRING(n) CHAR | |
| DISPLAY subrange | <==> | subrange | range check |
| DISPLAY subrange | ==> | STRING(n) | left zero truncate; error if string too big |
| STRING(n) | ==> | DISPLAY subrange | left zero fill; error if string too big; verify all numeric; range check |
| STRING(n) CHAR or EBCDIC | ==> | STRING(n) HEX | CHAR or EBCDIC string must be a |

13.7 EQUIVALENCE, COMPATIBILITY AND COERCIONS (Continued)

| DATA TYPE | DI-REC-TION | DATA TYPE | NOTES |
|---|---|---|---|
| STRING(1) CHAR | ==> | HEX | CHAR or EBCDIC; string must be a constant |
| STRING(2n) HEX | ==> | STRING(n) EBCDIC | HEX string must be a constant |
| STRING(2) HEX | ==> | EBCDIC | HEX string must be a constant |
| STRING(256) EBCDIC or CHAR or STRING (512) HEX | ==> | TRANSLATE TABLE | String must be a constant |

## 13.8 DATA MAPPING

This section describes how SPRITE data types map onto the Burroughs 2000/3000/4000 machine data types. This section is divided into two parts: mapping of simple types and mapping of aggregate types. This is done because the mapping of aggregate types depends upon the mapping of their components types. Following this section is a table that summarizes this information.

The mapping of a SPRITE data type is described in terms of four quantities: unit size, machine type, digit size and modulo.

Unit Size refers to the size of the type in terms of some unit of the machine. These units are the bit (one fourth of a digit), 1 digit, byte (two digits) and word (four digits). The unit size is often used in the length fields of machine instructions which refer to data of a particular type.

Machine Type refers to the physical machine type to which the data is mapped. The possible values are listed below.

UN - unsigned numeric

SN - signed numeric

UA - unsigned alpha

Digit Size refers to the total size of the data type expressed in digits. A one-digit base four fraction is used to represent bit lengths. Thus, 0.1 represents a length of one bit, 0.2 two bits, 0.3 three bits and 1.0 four bits or one digit.

Modulo refers to the required memory address alignment for an object of a particular type. Again, a fractional part will be used, this time to express bit modulo, i.e., the modulo for the type BIT is 0.1.

### 13.8.1 Data Mapping of Simple Types

This section describes the data mapping of SPRITE simple types in terms of the four quantities described above.

#### 13.8.1.1 Data Mapping of Booleans

BOOLEAN objects have unit size, digit size and modulo of 1, and a machine type of UN. The BOOLEAN value false has a numeric value of 0 (low order bit off) and the value true

13.8.1.1 Data Mapping of Booleans (Continued)

has a numeric value of 1 (low order bit on).

13.8.1.2 Data Mapping of Bits

BIT objects have a unit size of 1, digit size and modulo of 0.1 (see descriptions of digit size and modulo) and a machine type of UN. The identifier false represents the bit off value.

13.8.1.3 Data Mapping of CHAR

CHAR type has a unit size of one, digit size and modulo of 2 and a machine type of UA. The values are the 95 graphic EBCDIC characters.

13.8.1.4 Data Mapping of EBCDIC

The values of unit size, digit size, modulo and machine type for type EBCDIC is the same as for CHAR, namely, 1, 2, 2 and UA. Values of type EBCCIC are arranged in the EBCDIC collating sequence.

13.8.1.5 Data Mapping of HEX

For HEX, the values of unit size, digit size, modulo and machine type are 1, 1, 1 and UN, respectively. The values of type HEX are the hex digits 0-F.

13.8.1.6 Data Mapping of Integer Types

The unit size of the integer subrange a..b is the maximum of the number of digits in a and in b (not counting leading zeros). This can be concisely written as log10 (max (|a|,|b|)) + 1, unless a = b = 0, in which case the unit size is 1. If a < 0, then the machine type is SN. If a >= 0, then the machine type is UN. If the machine type is SN, the digit size is one more than the unit size, otherwise, the digit size equals the unit size. The modulo of integer types is always 1.

13.8.1.7 Data Mapping of Display Integers Types

The unit size of the tupe DISPLAY a..b is the number of digits in b (not counting leading zeros). The digit size is two times the unit size, the modulo is 2 and the machine type is UA. Actually, the values are represented as character digits.

13.8.1.8 Data Mapping of Symbolic and Ordered Types

The unit size of SYMBOLIC and ORDERED types is the digit size of the number of symbolics defined. This can be represented as log10 (# of symbolics) +1. The digit size is the same as the the unit size, the modulo is one and the machine type is UN. SYMBOLIC and ORDERED names are assigned consecutive decimal values, starting at 1. The limit on the nummber of names that may be defined in a SYMBOLIC or ORDERED type is 9999.

13.8.1.9 Data Mapping of Packed Symbolic and Ordered Types

The unit size of PACKED SYMBOLIC and ORDERED types is the number of hexadecimal digits in the hexadecimal value associated with the final symbolic defined. The digit size is the same as the unit size, the modulo and machine type are 1 and UN. PACKED SYMBOLIC and ORDERED names are assigned consecutive hexadecimal values starting at zero, except when an override is supplied as described in section 13.2.1.7. The number of names that may be defined in a PACKED SYMBOLIC or ORDERED type is 9999 and the maximum value that may be associated with any name is "FFFF".

13.8.1.10 Data Mapping of Binary

The unit size of BINARY types is the number of bits given in the definition, and must be a multiple of 4. The digit size is the unit size divided by 4. The modulo is 1 and the machine type is UN.

13.8.2 <u>Data Mapping of Aggregate Type</u>

This section describes the mapping of SPRITE aggregate types to B2000/3000/4000 machine data types in terms of the four quantities unit size, digit size, modulo and machine type. By definition, aggreqrate types are composed of objects of other types, so detailed descriptions of how these components are arranged in relation to one another within the aggregate are included.

13.8.2.1 Data Mapping of Pointers

There are three different implementations of pointers - pointers to procedures (PROC PTR), pointers to parametic strings and all other pointers.

The third case is easiest to describe. The unit size, digit size, modulo and machine type of "regular" pointers are 8, 8, 2 and UN, respectively. These pointers are represented as extended machine addresses.

13.8.2.1 Data Mapping of Pointers (Continued)

If the object pointed to is a parametric string, then the unit size, digit size, modulo and machine type are 14, 14, 2 and UN, respectively. This type of pointer is stored as a six digit length field followed by a "regular" 8-digit extended address.

For PROC PTRs, the unit size, digit size, modulo and machine type are 9, 9, 2 and UN. These pointers are stored in memory as a 6 digit address followed by a 3 digit segment number.

13.8.2.2 Data Mapping of Sets

The unit size and digit size for sets are the number of elements in the base type of the set. The modulo and machine type are 1 and UN. Sets are just groups of booleans where the value of the nth BOOLEAN indicates the presence or absence of the nth element of the base type in the set. Users should be forwarned that sets of CHAR contain only 95 elements (one for each printable character) while sets of EBCDIC contain 256 elements. The maximum number of elements in a set is 256.

13.8.2.3 Data Mapping of Translate Tables

For type TRANSLATETABLE the unit size, digit size, modulo and machine type are 389, 778, 1000 and UA. Translate tables are initialized with STRING (256) OF CHAR or EBCDIC.

13.8.2.4 Data Mapping of Strings

For strings of HEX, the digit size and unit size are the number of elements specified in the string. The modulo and machine type are 1 and UN.

For strings of CHAR or EBCDIC, the unit size is the number of elements specified in the string and the digit size is twice the unit size. The modulo and machine type are 2 and UA. Parametric strings are mapped using two components. The first is a descriptor, which has a unit size, digit size, modulo and machine type of 14, 14, 2 and UN. This descriptor contains a six digit unit size field (not digit size) followed by an eight digit extended address. The address points to the other component, the string itself, which has the same attributes as a non-parametric string of the length indicated by the length field. The controller in the address field is the same as the machine type of the base type of the string.

### 13.8.2.5 Data Mapping of Arrays

The machine type of an ARRAY is UN. The modulo of the array is the same as the modulo of the component type. If necessary, the length of each component is rounded up to be a multiple of the component's modulo by including filler. The unit and digit sizes of the array are computed by multiplying the length of each element by the total number of elements in the array. Elements in an array are stored in row-major order.

Arrays of BIT are handled slightly differently, since the modulo and digit size of a BIT is less than one digit. The most basic row of an array of BIT (corresponding to the last index) has a modulo of 1 and its length is rounded up to the next whole digit. All rows containing this row as a component and the array itself will thus have a modulo of one and a length expressed in whole digits.

Examples:

```
    EXAMPLE1 = ARRAY [1..8,1..6] OF BIT,
    EXAMPLE2 = ARRAY [1..9] OF 0..999;
```

The array type EXAMPLE1 will have a modulo of 1 and will be 16 digits long. The array type EXAMPLE2 will also have a modulo of 1 and will be 27 digits long.

### 13.8.2.6 Data Mapping of STRUC

The type STRUC has a modulo of 4 and a machine type of UN. The digit size and unit size are equal to the number of digits from the beginning of the STRUC to the end of the last field, rounded up to a modulo 4 value. Within the STRUC, regular and tag fields are positioned consecutively one after the other at the first location that satisfies each field's modulo requirements. The first field in each variant is positioned at the first location following the tag field (or the last field preceding the CASE for omitted or disjoint tag fields) that satisfies the field's modulo requirements. Subsequent fields in a variant are positioned in the same manner as regular fields. The length of a CASE construct is from the first location allocated to a tag or variant field to the end of the longest variant and is not necessarily an integral digit value.

13.8.2.6 Data Mapping of STRUC (Continued)

Examples:

```
EX1 = STRUC                                    % Offset   Length
          field1      BIT,                      %  0.0       0.1
          CASE tag1   BIT                       %  0.1       0.1
          IS false:   field2  BIT               %  0.2       0.1
          OR true:    field3  BIT               %  0.2       0.1
          ESAC                                  % length = 4 digits
      CURTS,                                    % filler = 3.1 digits

EX2 = STRUC                                    % Offset   Length
          field1      BIT,                      %  0.0       0.1
          CASE 0..2                             %
          IS 0:     field2      EX1             %  4.0       4.0
          OR 1:     field3      CHAR            %  2.0       2.0
          OR 2:     field4      BOOLEAN         %  1.0       1.0
          ESAC,
          field5      BIT                       %  8.0       0.1
      CURTS                                     % length = 12.0 digits
                                                % filler = 3.3 digits
```

13.8.2.7 Data Mapping of PACKED STRUC

The machine type for PACKED STRUC is UN and the digit size and unit size are computed in the same manner as for regular STRUCs, except they are rounded up to modulo 1 values, not modulo 4 values. The modulo of a PACKED STRUC is 1 or that of the field with the highest modulo, whichever is higher. Fields in a PACKED STRUC are positioned in the same manner as fields in a regular STRUC.

13.8.2.7 Data Mapping of PACKED STRUC (Continued)

```
EX1 = PACKED STRUC
         field1        STRING (4) OF HEX
      CURTS                %mod 1, length 4

EX2 = PACKED STRUC                                 % Offset     Length
         field1        BIT,                        %   0.1         0.1
         field2        STRUC                       %   4.0
                           field3   CHAR           %   4.0         2.0
                       CURTS
      CURTS                                        % mod 4, length 8

EX3 = PACKED STRUC
         field1             BIT,                   %   0.0         0.1
         CASE tag1          BOOLEAN                %   0.1         0.1
         IS  true:
                  field2    BIT                    %   0.2         0.1
         OR  false:
                  field3    BIT                    %   0.2         0.1
ESAC
      CURTS                                        % mod 1, length 1
```

13.8.3 Data Mapping Summary

| TYPE | UNIT SIZE | DIGIT SIZE | MACHINE TYPE | MODULO |
|---|---|---|---|---|
| BOOLEAN | 1 | 1 | UN | 1 |
| BIT | 1 | 0.1 | UN | 0.1 |
| CHAR | 1 | 2 | UA | 2 |
| EBCDIC | 1 | 2 | UA | 2 |
| HEX | 1 | 1 | UN | 1 |
| a..b (a >= 0) | IF a = b = 0 THEN 1 | = unit size | UN | 1 |
| (a < 0) | ELSE 1 + log10( max(\|a\|,\|b\|)) | = unit size + 1 | SN | 1 |
| DISPLAY a..b (a >=0 ) | FI | = 2 * unit size | UA | 2 |
| SYMBOLIC, ORDERED | 1 + log10 (# of names) | = unit size | UN | 1 |

13.8.3 Data Mapping Summary (Continued)

| TYPE | UNIT SIZE | DIGIT SIZE | MACHINE TYPE | MODULO |
|---|---|---|---|---|
| PACKED SYMBOLIC, ORDERED | # of hex digits in hex expression associated in last name | = unit size | UN | 1 |
| BINARY | # bits / 4 | = unit size | UN | 1 |
| PTR | 8 | 8 | UN | 2 |
| PTR (to parametric string) | 14 | 14 | UN | 2 |
| PROC PTR | 9 | 9 | UN | 2 |
| SET | # of elements in base type | = unit size | UN | 1 |
| TRANSLATE TABLE | 389 | 778 | UA | 1000 |
| STRING (OF HEX) | string size | = unit size | UN | 1 |
| (OF CHAR/EBCDIC) | string size | = 2 * unit size | UA | 2 |
| ARRAY | (element size rounded up to element modulo times # of elements in array) | | UN | same as element type |
| OF BIT | (see 13.8.2.5) | | UN | 1 |
| STRUC | (see 13.8.2.6) | (mod 4) | UN | 4 |
| PACKED STRUC | (see 13.8.2.7) | (not mod 4) | UN | highest modulo of any field |

13.8.3 Data Mapping Summary (Continued)

| TYPE | UNIT SIZE | DIGIT SIZE | MACHINE TYPE | MODULO |
|---|---|---|---|---|
| PACKED SYMBOLIC, ORDERED | # of hex digits in hex expression associated in last name | = unit size | UN | 1 |
| BINARY | # bits / 4 | = unit size | UN | 1 |
| PTR | 8 | 8 | UN | 2 |
| PTR (to parametric string) | 14 | 14 | UN | 2 |
| PROC PTR | 9 | 9 | UN | 2 |
| SET | # of elements in base type | = unit size | UN | 1 |
| TRANSLATE TABLE | 389 | 778 | UA | 1000 |
| STRING (OF HEX) | string size | = unit size | UN | 1 |
| (OF CHAR/EBCDIC) | string size | = 2 * unit size | UA | 2 |
| ARRAY | (element size rounded up to element modulo times # of elements in array) | | UN | same as element type |
| OF BIT | (see 13.8.2.5) | | UN | 1 |
| STRUC | (see 13.8.2.6) | (mod 4) | UN | 4 |
| PACKED STRUC | (see 13.8.2.7) | (not mod 4) | UN | highest modulo of any field |

## 14 STATEMENTS

Essential to a computer program is action. That is, a program must do something with its data even if that action is the choice of doing nothing!

Statements describe these actions. In the SPRITE language system statements are either of the simple_type, selection type or of the iteration_type.

A statement list is a sequence of statements and comments separated by semicolons. The syntax is:

```
|
|    _________ ; _________
|   /                      \
\__\______statement______/________________________________
      \                 /                                  \
       \__comment____/                                      |
                                                            |
                                                            |
```

Examples:

```
     sum    := small + medium;
     mult   := medium * large;
     inc   +:= medium;              % Note how the semicolon (;)
                                    % separates these statements in
                                    % a list.
```

Statements in a list are executed sequentially.

### 14.1 SIMPLE STATEMENTS

The simple statements are assertion, assignment, situation exit, the null statement, procedure control statements (call, return, and fail (*)) and storage space generation. Comments may appear in a statement list.

#### 14.1.1 ASSERT Statement

The ASSERT statement requires a specified relationship among program variables to be true. The relationship should always be true when the ASSERT statement is executed. Verification and proofs of program correctness may utilize assertions. They are especially useful as inexpensive tests to provide confidence in the validity of input to a program. The syntax of the ASSERT statement is:

14.1.1 ASSERT Statement (Continued)

```
|
|
\____ASSERT______boolean:expression______________________
                                                         \
                                                         |
                                                         |
```

If evaluation of the expression yields false, a run-time error occurs, resulting in program termination. If the expression is true, control passes to the next statement.

Examples:

```
ASSERT x1/x2 < 2*y3 + 2*y4
ASSERT table [i] <= table [i+1]
```

14.1.2 Assignment Statement

Assignment statements change the values of program data. A primary determines the destination. (See 19.2.1). The syntax of the assignment statement is:

```
|
|   destination:
\_____primary______ :=  ________assignment value:expr______
                  \___ +:=  ___/                            \
                  |\__ -:=  __/|                            |
                  |\__ *:=  __/|                            |
                  |\__ /:=  __/|                            |
                  |\__ &:=  __/|                            |
                  |\__ |:=  __/|                            |
                  \___ #:=  ___/                            |
                                                            |
                                                            |
                                                            |
```

where the primary may be any type, including structured types and file attribute selections. Its access must be read/write. The evaluation order of the variable and the expression is undefined. If the destination and assignment values are not type equivalent, the assignment value is coerced to the type of the destination if possible.

The assignment operators +:=, -:=, *:=, /:=, |:=, &:=, and #:= are abbreviations. For example, "a +:= b" means "a := a + b". An assignment operator may only be used on values for which the operator is defined. They allow code

14.1.2 Assignment Statement (Continued)

generation optimization.

Examples:

```
    a := 3 * v[1]
    s := message [field_index :: field_length]
    new_message [indx :: lngth] := s
    file.MYUSE:= IO
```

### 14.1.2.1 Swap Statement

The swap operator :=: causes the exchange of values of the left and right hand sides. The syntax is:

```
|
|   left hand side:                    right hand side:
\_____primary___________ :=: ___________primary___________
                                                          \
                                                          |
                                                          |
```

The left and right hand sides must be type equivalent and both must have read/write access. Subscripting, dereferencing, field selection, and substringing are permitted on both.

Example:

```
    a [i] :=: a [j];
```

### 14.1.3 Situation Statement

A situation statement provides an exit from the do group of an UNTIL-CASE statement (see 14.3.3). It is allowed only in this context. The UNTIL-CASE statement declares the situation names. The syntax of the situation statement is:

```
|
|
\______ LOOP_EXIT ____situation name_____________________
                                                         \
                                                         |
                                                         |
```

Execution of a situation statement transfers control out of the loop (the UNTIL-CASE do group) to the CASE alternative labeled by that situation name. See the example in 14.3.3.

## 14.1.4 Procedure Call Statement

A procedure call invokes a routine which does not return a value. The routine invoked is the one that is associated with the procedure name by the procedure definition. The call passes a particular set of actual parameters.

* A FAILURE clause may optionally follow the call in a procedure call statement.

A primary of type PTR TO PROC may be dereferenced to invoke the procedure to which it points. (See 13.5).

The syntax for procedure calls is:

```
|
|               procedure:
|\_______________ident______________________________________________
|  module:      /          /  \           _____ , _____        /   \
|\_ident_ . _/             |  |        /actual param:\         |   |
|       procedure:         |  \_ ( __\__expression_/__ ) _/    |
\______ primary_________/             \_______________/            |
 ___________________________________________________________________/
/
\___________________________________________________________________
   \                                                           /    \
    \_IF___FAILURE(*)_____then part_______________FI_/          |
         \__EOF___________/          \_else part_/               |
           \_INVALID_KEY_/                                       |
                                                                 |
```

The number of actual parameters must be the same as the number of formal parameters in the procedure definition. If the parameter access is read/write (VAR), the type of the actual parameter must be equivalent to the type of the corresponding formal parameter. If the access is read-only (CONST), the actual is coerced to the formal tyoe if necessary and if possible.

When the access of the formal parameter is read-only, the actual parameter may be an expression, variable or constant. When the access is read/write, the actual parameter must also have reac/write access, i.e., be a variable. The default is read-only. The order in which actual parameters are evaluated is undefined.

If the call is qualified by a module name, an intermodule call is made.

## 14.1.5 RETURN Statement (Continued)

statement is:

```
|
|
\______RETURN_____________________________________________________
              \                                           /      \
               \_____return value:expression______/              |
                                                                  |
                                                                  |
```

Examples:

```
PROC (old_element OLD_TYPE) RETURNS NEW_TYPE
VAR  new_element  NEW_TYPE;
        .
        .
        .
RETURN  new_element;
CORP;
```

If the type of this new_element return value and the RETURN type NEW_TYPE for this procedure are not equivalent, the returned value is automatically coerced to the RETURN type NEW_TYPE.

A run-time error occurs if a RETURN statement is not executed before control passes to the end of the body of a function.

## 14.1.6 * FAIL Statement

The FAIL statement provides recovery from programmer defined errors. The syntax of the FAIL statement is:

```
|
|
\_____FAIL_________________________________________________________
                                                                   \
                                                                   |
                                                                   |
```

The FAIL statement sets the boolean program condition FAILURE and causes an immediate exit from the procedure in which it is contained. There is no RETURN value. The procedure is said to have failed. If the call of this procedure has a FAILURE clause, the FAILURE is handled there. Otherwise, active procedures continue to fail until the FAILURE is handled or the program entry point fails. The latter causes a run time error. Data block variables

14.1.5 RETURN Statement (Continued)

statement is:

```
|
|
\______RETURN_____________________________________________
               \                                    /      \
                \_____return value:expression______/       |
                                                           |
                                                           |
```

Examples:

```
    PROC (old_element OLD_TYPE) RETURNS NEW_TYPE
    VAR  new_element  NEW_TYPE;
              .
              .
              .
    RETURN  new_element;
    CORP;
```

If the type of this new_element return value and the RETURN type NEW_TYPE for this procedure are not equivalent, the returned value is automatically coerced to the RETURN type NEW_TYPE.

A run-time error occurs if a RETURN statement is not executed before control passes to the end of the body of a function.

## 14.1.6 * FAIL Statement

The FAIL statement provides recovery from programmer defined errors. The syntax of the FAIL statement is:

```
|
|
\_____FAIL_______________________________________________
                                                         \
                                                         |
                                                         |
```

The FAIL statement sets the boolean program condition FAILURE and causes an immediate exit from the procedure in which it is contained. There is no RETURN value. The procedure is said to have failed. If the call of this procedure has a FAILURE clause, the FAILURE is handled there. Otherwise, active procedures continue to fail until the FAILURE is handled or the program entry point fails. The latter causes a run time error. Data block variables

14.1.6 * FAIL Statement (Continued)

may be used to provide information to the caller about how to handle the FAILURE. This information should be set up before the FAIL statement is executed. (See 14.1.4).

## 14.1.7 GENERATE Statement

The GENERATE statement creates an object (variable) which may only be referenced through a previously declared pointer variable. The access of the pointer must be read/write (VAR) and the access of the object referenced by the pointer must be read/write (VAR). (CONST access would prevent the new object from being initialized). The syntax of the GENERATE statement is:

```
|
|                                   pointer:
\__GENERATE___EXTERNAL___primary______________________________
              \__LOCAL___/          \                 pos integer:  \
                                     \__LENGTH__simple expr__  |
                                                                \|
                                                                 |
```

Allocation of storage to be referenced can be in one of two places. If LOCAL is specified in the GENERATE statement, the storage area pointed to is generated in temporary storage. This 'local' data area will exist until the PROC is exited. If EXTERNAL is specified, the storage area pointed to is generated in storage that exists for the duration of the program.

External pointers are pointers declared in DATA blocks, as parameters or as STATIC variables (see 15.4) in PROCs. Local pointers are pointers which are declared as being local variables to a PROC. External pointers may only point to EXTERNAL storage areas while local pointers may point to either EXTERNAL or LOCAL storage areas.

The LENGTH clause is only applicable if the pointer references a parametric string. The length of the space generated for a parametric string will be the maximum size in its range, unless a LENGTH clause appears. In that case, the positive integer expression which follows the word LENGTH will be the length of the parametric string, provided this length is within its range.

14.1.8 Null Statement

The null statement consists of consecutive statement delimiters. Control passes to the next statement. Because the null statement is included in the language, it is possible to treat ";" as if it were a terminator rather than a separator. That is, a ";" is possible before a FI, ELSE, OD, CORP, etc.

## 14.2 SELECTION STATEMENTS

The selection statements are the IF statement and the CASE statement.

14.2.1 IF Statement

The IF statement allows the selection of one of two alternative groups of statements for execution. Selection depends on the truth or falsity of a boolean expression. ELIF is a syntactic contraction for ELSE IF. A single FI closes the entire IF statement, including ELIFs. The control flow diagram for the execution of the IF statement is:

```
         |            --------- ... ----------
         |           |                        |
 IF______V_____      |   ELIF___________      |
  |            |     |    |            |      V
  | <boolean   |---> ... ---> | <boolean   |---> ... ----
  |expression> |false     |expression> |false          |
  |____________|          |____________|               |
      true|                   true|                    |
          |                       |                    |
THEN_____V____            THEN_____V_____     ELSE______V_____
   |         |              |            |      |              |
   |<group 1>|              | <group n>  |      |<else group>  |
   |_________|              |____________|      |______________|
         |                       |                     |
         |                       V                     |
         |<--------- ... <-------------------- ... <---
         |
         |
     FI  V
```

(the ellipses indicate optional parts).

### 14.1.8 Null Statement

The null statement consists of consecutive statement delimiters. Control passes to the next statement. Because the null statement is included in the language, it is possible to treat ";" as if it were a terminator rather than a separator. That is, a ";" is possible before a FI, ELSE, OD, CORP, etc.

## 14.2 SELECTION STATEMENTS

The selection statements are the IF statement and the CASE statement.

### 14.2.1 IF Statement

The IF statement allows the selection of one of two alternative groups of statements for execution. Selection depends on the truth or falsity of a boolean expression. ELIF is a syntactic contraction for ELSE IF. A single FI closes the entire IF statement, including ELIFs. The control flow diagram for the execution of the IF statement is:

```
        |            --------- ... -----------
        |           |                         |
 IF_____V_____      |    ELIF___________      |
  |           |     |       |           |     V
  | <boolean  |---> ... --->| <boolean  |---> ... ----
  |expression>|false        |expression>|false        |
  |___________|             |___________|            |
    true|                     true|                  |
        |                         |                  |
THEN____V____            THEN_____V_____   ELSE______V_____
   |         |              |           |      |           |
   |<group 1>|              |  <group n> |      |<else group>|
   |_________|              |___________|      |____________|
        |                          |                  |
        |                          V                  |
        |<---------- ... <-------------------- ... <---
        |
        |
     FI V
```

(the ellipses indicate optional parts).

14.2.1 IF Statement (Continued)

The syntax of the IF statement is:

```
|           __________ELIF___________
|        /  boolean:                 \
\__IF__\__expression____then part__/_______________FI__
                                   \__else part__/     \
                                                        |
                                                        |
```

where then part is

```
|
|
\__THEN____statements_________________________________
                                                      \
                                                       |
                                                       |
```

and else part is

```
|
|
\__ELSE____statements___________________________________
                                                        \
                                                         |
                                                         |
```

Notice that the delimiter FI eliminates ambiguities of associating the ELSE part in nested IF statements.

14.2.1 IF Statement (Continued)

Examples:

```
IF a = 0
THEN a := b;
FI;

IF   something
THEN IF   another_something
     THEN action
     FI
ELSE another_action
FI;
```

The ELIF format below is logically equivalent to the right half.

```
IF   x < 0              % IF x < 0
THEN error;             % THEN error;
     RETURN;            %      RETURN;
ELIF x = 0              % ELSE IF x = 0
THEN zero;              %      THEN zero;
     save;              %           save;
ELIF x < 10             %      ELSE IF x < 10
THEN x := x - 1;        %           THEN x:=x-1;
     iterate;           %                iterate;
ELSE retry;             %           ELSE retry;
FI                      %           FI;
                        %      FI;
                        % FI;
```

14.2.2 CASE Statement

The CASE statement allows the selection of one of several alternative groups of statements for execution. The value of the selector determines which alternative is executed. If the selector expression delivers a value for which there is no case alternative, the statement group of the else part is executed. If there is no else part in this situation, a run-time error will occur.

14.2.2 CASE Statement (Continued)

The control flow diagram for the execution of the CASE statement is:

```
                                    |
                                    |
                         CASE______V______
                             |            |
                             |  <selector |_________________
                             | expression>| out of range    |
                             |____________|                 |
                               |  | ... |                   |
           ____________________|  |     |                   |
          |                 ______|     |___                |
          |                |                |               |
    IS____V_____     OR____V_____    OR____V_____   ELSE______V______
    |           |    |           |   |           |  |                |
    | <group 1> |    | <group 2> |...| <group n> |  | <else part>    |
    |           |    |           |   |           |  |  (optional)    |
    |___________|    |___________|   |___________|  |________________|
         |                |                 |                |
         V                V                 V                V
     ______________________________ ... <_________________________
                             |
                             |
                        ESAC V
```

(the ellipsis indicates optional parts).

The association between selector values and alternative groups is made by prefixing the alternative with one or more appropriate values followed by ":".

These values are called case labels. One or more labels may be associated with each alternative. A given label may not be associated with more than one alternative.

The type of the selector expression must be finite. That is, it must be BOOLEAN, symbolic, subrange, CHAR, BIT, HEX, BINARY, STRING, SET or mnemonic (e.g. file.MYUSE). The only exception is when an attribute inquiry is used as the selector expression. Example: CASE port.SECURITYTYPE. In this example, the type of selector is mnemonic.

The case labels are values of this type. For example, when the selector is of the type symbolic range, the alternatives are labeled by range values of that type. If necessary, case labels are coerced to the type of the

14.2.2 CASE Statement (Continued)

selector expression. The syntax of the CASE statement is:

```
|
|
\__CASE____selector:expression____IS____
                                        \
 _______________________________________/
/
|  ____________________OR_____________
|/   ________ , _______               \
|  / constant labels: \               |
 \_\_value collection_/_ : _statements_/_______________ESAC_
                                        \_else part_/       \
                                                            |
                                                            |
```

A case label is a constant expression that is a value of the selector's type.

* If the selector type is ordered, a range of values may be used for case labels.

Examples:

```
CASE op_type
  IS   plus:    result := a + b
  OR   minus:   result := a - b
  OR   mult:    result := a * b
  OR   div:     result := a / b
  ELSE op_error
ESAC;
```

## 14.3 ITERATION STATEMENTS

Each of the iteration statements provides a different means of terminating a loop. The WHILE statement terminates with a boolean pre-test; the DO UNTIL statement with a boolean post-test; the FOR statement after stepping through a range; the FIND statement, with the value sought or after searching the whole array or linked list (*); and the UNTIL-CASE statement with any one of several specified situations.

### 14.3.1 WHILE Statement

The syntax of the WHILE statement is:

### 14.3.1 WHILE Statement (Continued)

```
|
|
\__WHILE____boolean:expr____DO____statements____OD________
                                                          \
                                                          |
                                                          |
```

While the boolean expression is true, the DO group is repeatedly executed. If the expression is initially false, the DO group is not executed and control passes to the next statement in sequence.

Examples:

```
     WHILE another_transaction
     DO process_transaction
     OD;
```

### 14.3.2 DO UNTIL Statement

The syntax of the DO UNTIL statement is:

```
|
|
\__DO____statements____OD____UNTIL____boolean:expr________
                                                          \
                                                          |
                                                          |
```

The DO group is repeatedly executed until the boolean expression is true. Since termination is a post-test, the DO group is executed at least once.

14.3.2 DO UNTIL Statement (Continued)

Examples:

```
DO something_useful
OD
UNTIL end_of_input_file | fatal_error
```

14.3.3 UNTIL-CASE Statement

The UNTIL-CASE statement provides controlled multiple exits from a loop. The syntax of the UNTIL-CASE statement is:

```
|              ______ , _______
|            /                  \
\__UNTIL__\__situation:id__/__DO__statement list__OD____
                                                        \
   _____________________________________________________/
  /
  |                _____________ OR ________________
  |              /    ___ , ____                     \
  |             |   /situation:\                      |
  \_CASE__IS__\__\__ident___/__ : __statements__/__ESAC___
                                                          \
                                                           |
                                                           |
```

The situation identifiers are declared by and are local to the UNTIL-CASE statement.

The DO group is repeatedly executed. If a situation statement (see 14.1.3) is encountered, control passes to the CASE component labeled by that situation name. Control then leaves the UNTIL-CASE statement. Every situation listed after UNTIL must have an associated alternative in the CASE part. An alternative may be labeled by several situations. An alternative may be an empty statement list.

UNTIL-CASE statements may be nested, but the situation names must be unique. If a LOOP_EXIT to a situation of the outer UNTIL-CASE is encountered in an inner one, control passes directly to the corresponding alternative of the outer statement.

14.3.3 UNTIL-CASE Statement (Continued)

Examples:

```
     i := lwb (table) - 1;
     UNTIL done, notfound
     DO
          i +:= 1;
          token := table[i];
          IF    token = "this"
          THEN LOOP_EXIT done
          FI;
          IF    i = upb (table)
          THEN LOOP_EXIT not found
          FI
     OD
     CASE
     IS   done:      write ("i=", i);
     OR   notfound: write ("not found", upb(table));
     ESAC;
```

14.3.4 FOR Statement

The FOR statement controls the iteration by stepping a control variable through the specified range of values. The syntax of the FOR statement is:

```
|
|       control var:
\__FOR_____ident_____OVER___simple expr .. simple expr____
                          \______indicant . RANGE______/  \
     ______________________________________________________/
    /
    \____________________DO____statement list___OD______
       \_DESCENDING_/                                   \
                                                         |
                                                         |
```

The control variable is declared by and is local to the FOR statement. It has read-only access. Its type is that of the control range: scalar, ordered or unordered (i.e., BOOLEAN, symbolic, subrange or CHAR). The control range is a range of values or a RANGE inquiry. (See 19.2.4).

The control variable is stepped through the control range. If the lower bound is greater than the upper bound, the control range is empty and the DO group is not executed. If DESCENDING occurs, the control variable has an initial value of the range maximum and is decremented; otherwise, it is incremented. If the control range is unordered

## 14.3.4 FOR Statement (Continued)

(e.g., RANGE inquiry used), DESCENDING may not appear. The DO group is executed once for each value in the control range. The control variable is changed and tested before each execution of the DO group. If the control variable does not exceed its termination value the DO group is executed again.

Examples:

```
FOR i OVER 1..3
DO
    sing ("row")
OD;
sing ("your boat");

FOR i OVER 0..10 DESCENDING
DO
    countdown (i)
OD;
blast_off;

FOR i OVER 1..5
DO
    a[i] := i - 1
OD
```

## 14.3.5 FIND Statement

* The FIND statement provides the ability to search an array or a linked list for an element which satisfies a specified condition. (This feature is not implemented as of the ASR 6.6 release.) After the search is performed, one of the two alternate groups of statements is executed depending upon whether or not the search was successful. The syntax is:

14.3.5 FIND Statement (Continued)

```
|
|
\___FIND___find pointer spec______find control_____
                                                   \
    _______________________________________________/
   /
   \_WHERE_______find condition__________
                                         \
    _____________________________________/
   /
   \_THEN______statements___ELSE___statements___ONIF________
                          \__________________/              \
                                                            |
                                                            |
```

The find_pointer_spec clause specifies the statement's result pointer. It is either local or external. LOCAL means a statement-local variable (identifier) whose value and scope are available only in the THEN part of the statement. LOCAL is the default. EXTERNAL means an externally-declared (to the statement) variable (primary) which is a pointer to the type of the array's components (list elements) and which on a non-hit will receive the nil pointer. The syntax is:

```
find pointer spec
|
|                               pointer:
\_____________________________identifier______________________
       \   \___LOCAL___/    pointer:            /             \
        \___EXTERNAL________primary____________/              |
                                                              |
                                                              |
```

The find control clause specifies the type of the search to be performed (array or linked list). The syntax is:

14.3.5 FIND Statement (Continued)

```
find control
|
|                                                                      array:
|\____________________________________________________INTO____primary________
|                    \             index:           /                         \
|                     \_AND__identifier_/                                     |
|                                                                             |
|                base pointer:     limit pointer:            array:           |
|\___OVER____primary___ .. ___primary______INTO___primary___                  |
|                              \__ END __/                                    \|
|                                                                             |
|                               predecessor pointer:                          |
\_______________WITH_______identifier______                                   |
                   \_______________________/   \                              |
       _________________________________________/                             |
      /           list pointer:                                               |
      \__FROM_____primary______USING___link field_____________                |
                                                                             \|
                                                                              |
                                                                              |
```

The syntax for link field is:

```
|
|               ____________ . __________
|             /         field name:        \
\___________\_________identifier_________/_____________________
                                                               \
                                                                |
                                                                |
```

The first form of find control is used if an array is to be searched. An optional index variable may be declared. Its type is the same as the array's index type. Its value is the index of the element in the array satisfying the find condition. The array primary is an entire array or an array slice with variable or constant bounds. It is the array to be searched and must be one-dimensional.

The second form of the find control permits the use of pointers to an array's components as the delimiters of the FIND statement. The use of the new reserved word END provides access through and including the last array element. The array primary may not be an array slice.

The third form of the find control is used if a linked list is to be searched. The element pointer identifier defines a pointer variable whose type is PTR TO <list element type>. It points to an element in the list satisfying the

14.3.5 FIND Statement (Continued)

find condition. A predecessor pointer may optionally be defined. It is of type PTR TO PTR TO <list element type>. It points to the link field of the element which precedes the element satisfying the find condition. The predecessor pointer allows the programmer to delink the found element or perform other manipulations requiring access to the link of the found element. If the first element in the list satisfies the find condition, the predecessor points to the list head pointer.

The list pointer primary is a pointer to the first element in the list to be searched (the list is terminated by a nil link field). The link field clause specifies a list of field selections which are to be applied to the list element to get the field that points to the next element in the list (i.e. the link field).

The find condition specifies the condition which the element being searched for must meet. The syntax is:

```
          find condition
          |
          |
          |\____MIN_______find primary_______________________________
*         |   \__MAX__/                                              \
          |                                                           |
          |\__find primary__________________key:expression___________ |
          |                  \___  =  ___/
*         |                  \___ ¬=  ___/                           \|
          |                  |\__  <  __/|                            |
*         |                  |\__ <=  __/|                            |
*         |                  |\__  >  __/|                            |
*         |                  \___ >=  ___/                            |
          |                                                           |
          |                     set mask:                             |
          \___find primary__ * __expression____ =  __ {} ____________ |
                                                                      \
                                                                       |
                                                                       |
```

If the MIN or MAX form is used, the array/list is searched for the smallest or the largest element respectively. If a relational form of the fird condition is used, the array/list is searched for an element satisfying the relational condition. If the set intersection form of the find condition is used, the specified set mask expression is intersected with each find primary until an element is found which is either equal or not equal to the empty set after the intersection is performed.

## 14.3.5 FIND Statement (Continued)

Currently, only MIN, < and = are allowed.

Examples:

```
    % search an array, setting an index to the element
    FIND rw_ptr AND idx INTO reserved_words [1..num_res_words]
         WHERE rw_ptr@.name = id [1::leng]
    THEN
         next_sy := reserved_words [idx+1].value
    ELSE
         error ("Not a reserved word")
    DNIF;
        % search an array, setting a pointer to an element
    FIND rw_ptr INTO reserved_words [1..num_res_words]
         WHERE rw_ptr@.name = id [1::leng]
    THEN
         sy := rw_ptr@.value;
    ELSE
         error ("Not a reserved word")
    DNIF;


*   % search a linked list, setting a pointer
    %                          to an element
    FIND elem_ptr FROM list_head USING control.rlink
         WHERE elem_ptr@.data.string [1::4] = "wxyz"
    THEN
         conv_value := elem_ptr@.data.value;
    ELSE
         conv_value := no_value;
    DNIF;
```

## 15 DEFINITIONS AND DECLARATIONS

Definitions set forth meaning and character names used for constants and variables. Further, definitions associate names with blocks of data and blocks of code (procedures). Declarations, on the other hand, set forth the type of operations (read only, write only, read and write) that can be performed on variables by procedures. Declarations also grant to procedures access to blocks of data.

Data type (TYPE) names must be defined prior to their use in definitions and declarations. The only exceptions are the names associated with predefined data types and the referenced type of pointers.

### 15.1 CONSTANT DEFINITIONS

Constant definitions set forth a value to be associated with a name (identifier).

Using the constant name has the same effect as using the value in all contexts. The syntax is:

```
|
|               __________________ , __________________
|          /                                           \
\__CONST__\_identifier____________ = __const:expr__/__ ; ___
                     \__type__/                              \
                                                             |
                                                             |
```

This definition is introduced by the reserved word, CONST, followed by an identifier and the identifier's type in some cases, followed by an equal sign (=) and concluded with some constant value or constant expression followed by a semicolon (;). Where two or more definitions appear in a list, each definition is separated by a comma (,). The last line of the list terminates with a semicolon (;). Grouping or sets of constant definitions are permissible.

15.1 CONSTANT DEFINITIONS (Continued)

Example:

```
        TYPE
            VF_NUMB     = 0..99,
            DAY         = ORDERED (sun, mon, tues, wed,
                                   thurs, fri, sat),
            SET_OF_DAY  = SET OF mon..fri;

        CONST   % GROUP 1

            max_length = 20,
            max_size   = max_length + 2;

        CONST   % GROUP 2

            modl_header_file  VF_NUMB = 89,      % using indicant
            modl_entry_file   VF_NUMB = 90,      % using indicant
            modl_extn_file    VF_NUMB = 90 + 1;  % using indicant

        CONST   % GROUP 3

            d_day     DAY            = mon,           % using indicant
            d_day_2                  = DAY (tues),    % using cast
            day_pair SET_OF_DAY      = (thurs, fri);  % using subset

        CONST   % GROUP 4

            blank          = " ",
            comma          = ",",
            semicolon      = ";",
            upper_case     = "AEIOU",
            lower_case     = "aeiou",
            numeric        = "01235",
            alpha          = upper_case + lower_case,
            alphanumeric   = alpha + numeric,
                  .
                  .
                  .;
```

The indicant type name need not be used when the data type of the constant identifier is the same as the constant expression (Group 1 and Group 4).

In cases where the constant expression's type could be ambiguous, the indicant type name is required to interpret the expression (Group 2 and 3). When the indicant type name is present, the result value of the expression gets coerced (an implicit conversion) to the type specified, if

## 15.1 CONSTANT DEFINITIONS (Continued)

possible. The effect of having the indicant type name present can alternately be achieved by using a cast (Group 3) in the expression.

Constant identifiers may be defined in terms of other previously defined constant identifiers (Group 4).

Constant definitions may be coded in MID descriptions or in the program modules themselves. When these definitions are coded in the MID, the scope of the constant identifier is the MID and all program modules in the KNOWS list. If these definitions are coded in a program module, but outside of the procedures in the program module, the constant identifier's scope is the entire program module.

When the definition is coded inside a particular procedure of a program module, the scope is limited to that procedure.

## 15.2 TYPE DEFINITIONS

The type definition facility allows the programmer to create data types beyond those already predefined in the SPRITE language system. A type definition associates a name (indicant) with a type description. Wherever a type indicant name appears the effect is the same as though the type definition were used, except for separately described structures which are not compatible even if their descriptions match exactly. (See 13.3.3). The syntax is:

```
|              ________________ , _________________
|            /                                     \
\__TYPE__\_____indicant name___ = ___type______/__ ; ___
             \                              /          \
              \_parametric type definitions_/           |
                                                        |
```

This definition statement is introduced with the reserved word, TYPE, followed by any valid type indicant name, followed by an equal sign and a type descriptor. TYPE definitions appearing in a list are separated by commas (,).

The last definition in the list is terminated with a semicolon (;). Grouping or sets of type definitions are permissible.

## 15.2 TYPE DEFINITIONS (Continued)

Example:

```
          TYPE
              VF_NUMB    = 0..99,
              VF_REC_NUM = 0..99999,
              VF_PAGE    = VF_NUMB;
```

This example defines new data types called VF_NUMB, VF_REC_NUM and VF_PAGE. The first definition defines VF_NUM to be the set of all positive integers in the range 0 to 99. The second defines VF_REC_NUM to be all positive integers in the range 0 to 99999. The third makes use of a convenience in the SPRITE language of defining a new type in terms of a type previously defined.

TYPE definitions may be coded in MID descriptions or in the program modules. The scope of knowledge about the type indicant names follow the same rules as that of constant identifier names.

## 15.3 PARAMETRIC TYPE DEFINITIONS

A parametric type definition specifies a family of related data types. Each type in the family has the same base type and method of organization, but the 'size' attribute of each type may differ by allowing the size to be 'parameterized' in the parametric type definition.

Parametric type definitions can be used only with STRING or ARRAY (*) as the method of organization. The parametric STRING type constructor size parameter controls the length of strings whose type is derived from the parametric type. The parametric ARRAY type constructor (*) size parameter controls the upper bound of the arrays whose type is derived from the parametric type.

## 15.3 PARAMETRIC TYPE DEFINITIONS (Continued)

The syntax for parametric type part is:

```
  |
  |                                upper bound:
  \___indicant__ ( __ param:ident____subrange___ ) __ = __
                                                          \
    ____________________________________________________/
   /
  |\__STRING__ ( __param:ident__ ) __________________________
  |                                  \    \ element:string /   \
  |                                   \_OF_\__base type____/    |
  |                                                             |
  |              lower bound:      param:            element:   |
* \___ARRAY__ [ ___constant__ .. __ident__ ] ___OF___type___    |
                                              \___/          \ |
                                                                |
```

For the parametric STRING constructor, the upper bound subrange must be an integer range type. For the parametric ARRAY constructor, the lower bound constant must be type compatible with the upper bound subrange, which must be a finite scalar type.

Examples:

1. TYPE VECTOR (ubnd 10..100) = ARRAY [1..ubnd] OF 0..1000;

```
     sum_vector
     PROC (vector VECTOR) RETURNS 0..1000000000;
          VAR sum 0..1000000000 := 0;
          FOR i OVER 1..upb(vector)
          DO
               sum +:= vector [i];
          OD;
          RETURN sum;
     CORP;
```

2. TYPE LINE (n 1..132) = STRING (n) OF CHAR;

```
     fill_buffer
     PROC (text LINE);
          SHARES buffer-data;      %a data block containing buffer
                                   %and current length
          buffer [current:length (text)];
          current +:= length (text);
     CORP;
```

A parametric type indicant may be used only as the reference type of a pointer or the type of a formal

15.3 PARAMETRIC TYPE DEFINITIONS (Continued)

parameter.

## 15.4 VARIABLE DECLARATIONS

A data type in the SPRITE language system may be either referenced by a type definition or more directly described by a variable declaration.

Every variable coded inside a procedure must be declared in a variable declaration. This declaration, like other declarations and definitions, must textually precede any use of the variable.

A variable declaration associates an identifier (variable) with a data type. The syntax of a variable declaration is:

```
|
|      ------------------------ , ------------------------------
|     /  -----  ,  ---                                          \
|     | /            \                                          |
\_VAR_\_\_var:ident_/_type_____________________________________/__ ; __
                            \_STATIC_/ \___ := init value___/          \
                                                                       |
                                                                       |
                                                                       |
```

The reserved word VAR heads this declaration followed by a variable name (identifier) and the variable's TYPE. Any number of variables of the same type may be separated by commas and declared with a single type indicant name. This declaration is terminated with a semicolon (;). Grouping or sets of variable definitions are permissible.

The reserved word STATIC may optionally follow the TYPE indicant name where the programmer desires a one time allocation and initialization of the variables. Non-static variables (default) are allocated and initialized upon each activation of their home procedure.

The lifetime of STATIC variables is the execution time of the program. The lifetime of non-static (default) variables begins and ends with each execution of their home procedure.

Where the programmer desires to initialize variables in the declaration, a constant expression may be used. This expression follows the TYPE indicant name and is preceded by the assignment operator (:=). Variables may assume any values in their associated TYPE.

15.4 VARIABLE DECLARATIONS (Continued)

Example:

```
convert_data_block
PROC;
SHARES state_info;
VAR new_token, next_token, extra_token ICM_TOKEN;
    .
    .
```

The EOF condition signals end-of-file detected. It is only valid for the following: . .; CORP

```
convert_block_table
PROC;
SHARES state_info;
VAR new_ref     ADDRESS_REF,
    old_token   OLD_TOKEN,
    repl_token  REPL_TOKEN;
      .
      .
      .;
CORP

get_word
PROC;
VAR l_num 0..72;
    .
    .;
CORP

convert_code
PROC;
VAR new_type CODE_TOKEN STATIC := inline_data;
    .
    .;
CORP
```

15.5 SHARES DECLARATION

The facility for sharing data within a DATA block and the "files" within FILE, PORT and NSP blocks among procedures is called a SHARES declaration. This declaration allows a procedure to access variables within any block definition. (See 18).

The SHARES declaration is coded within a procedure definition. (See 19.1). In addition to granting a procedure access to a block or blocks of data, the SHARES declaration can restrict (by using CONST) a procedure's

15.5 SHARES DECLARATION (Continued)

operations to only be able to read the variables in a DATA block. By using a VAR restriction, a procedure is allowed to read and modify the values of variables in a DATA block.

The syntax of the SHARES declaration is:

```
|
|               _______________ , _________________
|              /                                   \
\__SHARES__\_____________block name:identifier_____/__ ; __
                \_CONST_/                                  \
                 \_VAR_/                                    |
                                                            |
                                                            |
                                                            |
```

This declaration is introduced with the reserved word SHARES followed by the block name and terminated with a semicolon(;). The block name may be optionally preceded with the reserved word CONST to prevent modification of the data block's variables. The reserved word VAR may optionally precede the data block name where unrestricted (READ/WRITE) access to the variables is desired. VAR is the default access afforded to procedures.

Where a procedure is to SHARE more than one DATA block, each block specified in the declaration is separated by a comma (,). The last block specified is followed by a semicolon (;).

When a DATA block is SHAREd amongst several procedures, the scope of the variables in the DATA block encompasses every procedure that SHARES the DATA block. These variables have a scope more global than those variables declared local to an individual procedure.

15.5 SHARES DECLARATION (Continued)

Examples:

```
    icm_utility
    MOD

       token_info
       DATA
           word         NAME,
           delim        BOOLEAN;

       scan_info
       DATA
           cbuf         STRING(80) := "%",
           cptr         1..73      := 1,
           eof          BOOLEAN    := false;

       names
       DATA
           old_icm,
           old_pack,
           new_icm,
           new_pack     FILE_NAME,
           lib_name     NAME;

       get_word
       PROC;
       SHARES token_info, CONST scan_info;
       :
       :
       :
       CORP;

       process_commands
       PROC;
       SHARES token_info,names;
       :
       :
       :
       CORP;

       skip_blanks
       PROC;
       SHARES scan_info;
       :
       :
       :
       CORP;

    DOM
```

## 15.5 SHARES DECLARATION (Continued)

Note that the procedure 'get_word' has restricted access to the variables in DATA block 'scan_info'. This restricted access prohibits modification of the initial values of the variables in 'scan_info'.

## 16 PROCEDURES

Procedures specify blocks of code and their associated data which can be invoked from other points in the program. A procedure is composed of zero or more constant definitions, type definitions, shares declarations, variable declarations, and executable statements.

Procedures can occur only within the inherently larger scope of a program module. Procedures serve as entry points into program modules. Consequently, the start of every SPRITE system program is via entry into a procedure in a program module.

All procedures are invoked by a procedure call statement, except the program entry point procedure. This procedure is defined by an ENTRY designation in the MID component of the program. It is automatically actuated when the program is executed.

Recursive procedures are permissible, i.e., one whose code includes a direct or indirect call to itself. Direct calls are those made from within its line of code to itself. Indirect calls are when a procedure 'A' calls some procedure 'B' who in turn calls procedure 'A'.

Procedure calls may be made to procedures within other modules. This is known as an intermodule call. However, this type of call must be formally declared in the program's MID component.

Statements inside procedures are executed sequentially unless the path of execution is changed by some conditional test.

A procedure may be exited by the execution of its last statement, or by the execution of a RETURN or FAILURE(*) statement.

A procedure that returns a value is referred to as a 'function'. Procedures may also name parameters.

Procedure parameters provide a communication channel between the caller and a called procedure. Actual parameters provided by the caller are bound to formal parameters declared in the called procedure. The data types of actual and formal parameters must match. The called procedure access to its parameters may be restricted to read-only (CONST) access.

16 PROCEDURES (Continued)

All names defined in a procedure are known only in that procedure. These names include: constant names, variable names, and indicant names. These names have a scope that is referred to as procedure-local. However, these same names may be defined differently in other procedures.

The lifetime of most names defined inside a procedure is the execution time of the procedure. Variables declared STATIC in variable (VAR) declarations are an exception.

## 16.1 PROCEDURE DEFINITIONS

The function of a procedure definition is to associate an identifier with a block of code and its data. Procedure definitions are coded within program modules. It is the only **mandatory** component in a program module. The syntax of a procedure definition is:

```
|
|
\__proc name:ident__PROC__
                          \
 _________________________/
/
\_________________________________________ ; __
   \__parameters__/  \__RETURNS__type__/       \
                                               |
 ______________________________________________/
/  _________________________    ______ ; ______
| /                         \  /               \
\_\_________________________/__\_______________/_________CORP_
   \___const def______ ; _/    \__statement__/  \_ ; _/        \
   |\__type def_____/                                          |
   |\__var dec_____/|                                          |
   |\__shares dec__/|                                          |
   \___comment______/                                          |
                                                               |
                                                               |
```

16 PROCEDURES (Continued)

All names defined in a procedure are known only in that procedure. These names include: constant names, variable names, and indicant names. These names have a scope that is referred to as procedure-local. However, these same names may be defined differently in other procedures.

The lifetime of most names defined inside a procedure is the execution time of the procedure. Variables declared STATIC in variable (VAR) declarations are an exception.

## 16.1 PROCEDURE DEFINITIONS

The function of a procedure definition is to associate an identifier with a block of code and its data. Procedure definitions are coded within program modules. It is the only **mandatory** component in a program module. The syntax of a procedure definition is:

```
|
|
\__proc name:ident__PROC__
                          \
 _________________________/
/
\__________________________________________ ; __
   \__parameters__/  \__RETURNS__type__/        \
                                                |
 _______________________________________________/
/  _________________________   ______  ; ______
| /                         \  /               \
\_\_________________________/__\_______________/_________CORP_
   \___const def______ ; _/   . \__statement__/  \_ ; _/      \
   |\__type def_____/                                         |
   |\__var dec_____/|                                         |
   |\__shares dec__/|                                         |
   \___comment______/                                         |
                                                              |
                                                              |
```

14.1.6 * FAIL Statement (Continued)

may be used to provide information to the caller about how tc handle the FAILURE. This information should be set up before the FAIL statement is executed. (See 14.1.4).

14.1.7 GENERATE Statement

The GENERATE statement creates an object (variable) which may only be referenced through a previously declared pointer variable. The access of the pointer must be read/write (VAR) and the access of the object referenced by the pointer must be read/write (VAR). (CONST access would prevent the new object from being initialized). The syntax of the GENERATE statement is:

```
|
|                                  pointer:
\__GENERATE___EXTERNAL___primary__________________________
              \__LOCAL___/          \             pos integer:  \
                                     \__LENGTH__simple expr__  |
                                                               \|
                                                                |
```

Allocation of storage tc be referenced can be in one of two places. If LOCAL is specified in the GENERATE statement, the storage area pointed to is generated in temporary storage. This 'local' data area will exist until the PROC is exited. If EXTERNAL is specified, the storage area pcinted to is generated in storage that exists for the duration of the program.

External pointers are pointers declared in DATA blocks, as parameters or as STATIC variables (see 15.4) in PROCs. Local pointers are pointers which are declared as being local variables to a PROC. External pointers may only pcint to EXTERNAL storage areas while local pointers may pcint to either EXTERNAL or LOCAL storage areas.

The LENGTH clause is only applicable if the pointer references a parametric string. The length of the space generated for a parametric string will be the maximum size in its range, unless a LENGTH clause appears. In that case, the positive integer expression which follows the word LENGTH will be the length of the parametric string, provided this length is within its range.

16.1 PROCEDURE DEFINITIONS (Continued)

where the syntax of the parameters is:

```
|
|       ___ , ____________________________________
|      /  param: \                                 \
\_ ( __\__ident__/______________________________type__/__ ) __
                  \___CONST___/ \__UNIV__/                     \
                   \___VAR___/                                 |
                    \_VALUE_/                                  |
                                                               |
```

This definition is introduced with a procedure name. This name is an identifier. The reserved word PROC follows. The programmer may optionally include a formal parameter list. This formal parameter list contains the identifier name of the argument variable, its access mode, and an optional UNIV designation and the argument variable's TYPE.

Although procedures do not necessarily have to have a parameter list, the reserved word PROC may be followed by a RETURNS type designation. The RETURNS type represents the TYPE of a solution value computed in the procedure to be returned to the procedure calling statement. The type of the solution value may be any established indicant type.

16.2 PARAMETERS

The procedure definition includes a description of the procedure's formal parameters, if any. At the procedure call these formal parameters are bound to actual parameters provided by the calling procedure. The formal-to-actual binding allows the actual parameter to be accessed through the name of the formal. The access may be "read-only" or "read/write".

16.2.1 Access

The keyword CONST preceding the parameter type specifies read-only access which is the default access. When CONST is used, the types of the actual and formal parameters must be compatible (see 13.7). When there is read-only access through a formal to an actual parameter, there may also be read/write access to the same variable (i.e., via data block variable names or read-write formal parameter names). If this is the case and the read/write access path is used to change the value then the value accessed through the read-only formal parameter may or may not be changed (i.e., it is undefined in the language). In general, there may be unexpected results whenever there is more than one access

16.1 PROCEDURE DEFINITIONS (Continued)

where the syntax of the parameters is:

```
|
|        ___ , ______________________________________
|       /  param: \                                   \
\_ (  __\__ident__/___________________________type__/__ )  __
                   \___CONST___/ \__UNIV__/                    \
                    \___VAR___/                                 |
                     \_VALUE_/                                  |
                                                                |
```

This definition is introduced with a procedure name. This name is an identifier. The reserved word PROC follows. The programmer may optionally include a formal parameter list. This formal parameter list contains the identifier name of the argument variable, its access mode, and an optional UNIV designation and the argument variable's TYPE.

Although procedures do not necessarily have to have a parameter list, the reserved word PROC may be followed by a RETURNS type designation. The RETURNS type represents the TYPE of a solution value computed in the procedure to be returned to the procedure calling statement. The type of the solution value may be any established indicant type.

16.2 PARAMETERS

The procedure definition includes a description of the procedure's formal parameters, if any. At the procedure call these formal parameters are bound to actual parameters provided by the calling procedure. The formal-to-actual binding allows the actual parameter to be accessed through the name of the formal. The access may be "read-only" or "read/write".

16.2.1 Access

The keyword CONST preceding the parameter type specifies read-only access which is the default access. When CONST is used, the types of the actual and formal parameters must be compatible (see 13.7). When there is read-only access through a formal to an actual parameter, there may also be read/write access to the same variable (i.e., via data block variable names or read-write formal parameter names). If this is the case and the read/write access path is used to change the value then the value accessed through the read-only formal parameter may or may not be changed (i.e., it is undefined in the language). In general, there may be unexpected results whenever there is more than one access

16.2.1 Access (Continued)

path to a given variable.

The keyword VAR specifies read/write access to the actual parameter. When VAR is used, the types of the actual and formal parameters must be equivalent (see 13.7) (if the formal parameter's type is not parametric) and the actual parameter must be a variable.

Parameters may be passed by value, as well as reference. To specify a value parameters, use "VALUE" in place of "CONST" or "VAR" in the parameter declaration. Use of the keyword VALUE specifies read/write access to the formal parameter but read-only access to the actual parameter. That is, it lets you change the formal parameter like a local variable without affecting the actual parameter. When VALUE is used, the types of the actual and formal parameters must be compatible. The type of the parameter must be any non-parametric type with a total size of 100 digits (or 100 characters for non-hex strings).

## 16.2.2 Universal Parameters

Universal parameters are used to construct general purpose procedures for a wide class of data by relaxing, though not eliminating, type checking between actual and formal parameters. When type checking is relaxed, the actual parameter may be of any type so long as the length is less than or equal to that of the formal parameter and its modulo requirement is greater than or equal to that of the formal parameter.

A universal parameter is specified by preceding the parameter type by the keyword UNIV in the procedure definition. If the procedure is a module entry point, the keyword UNIV must also appear in the MID specification of the procedure.

Universal parameters may only be used with formal parameters of a parametric string type (see 15.3).

16.2.1 Access (Continued)

path to a given variable.

The keyword VAR specifies read/write access to the actual parameter. When VAR is used, the types of the actual and formal parameters must be equivalent (see 13.7) (if the formal parameter's type is not parametric) and the actual parameter must be a variable.

Parameters may be passed by value, as well as reference. To specify a value parameters, use "VALUE" in place of "CONST" or "VAR" in the parameter declaration. Use of the keyword VALUE specifies read/write access to the formal parameter but read-only access to the actual parameter. That is, it lets you change the formal parameter like a local variable without affecting the actual parameter. When VALUE is used, the types of the actual and formal parameters must be compatible. The type of the parameter must be any non-parametric type with a total size of 100 digits (or 100 characters for non-hex strings).

## 16.2.2 Universal Parameters

Universal parameters are used to construct general purpose procedures for a wide class of data by relaxing, though not eliminating, type checking between actual and formal parameters. When type checking is relaxed, the actual parameter may be of any type so long as the length is less than or equal to that of the formal parameter and its modulo requirement is greater than or equal to that of the formal parameter.

A universal parameter is specified by preceding the parameter type by the keyword UNIV in the procedure definition. If the procedure is a module entry point, the keyword UNIV must also appear in the MID specification of the procedure.

Universal parameters may only be used with formal parameters of a parametric string type (see 15.3).

## 16.3 FORWARD DEFINITIONS

A forward definition permits a procedure call to appear in a procedure before the definition of the called procedure has appeared. Its syntax is:

```
|
|
\__proc name:ident PROC__________________________________________
                    \__parameters__/  \__RETURNS type__/  \
                                   __________________________________/
                                  /
                                  \__FORWARD__ ; ______________
                                                               \
                                                                |
                                                                |
```

## 17 IMPLEMENTATION OF MACRO FACILITY

1. A macro is a safe and useful implementation of "inline PROCs". No NTR/EXT code is generated.

2. The MACRO definition resembles that of a PROC:

```
<macro_ident>
        MACRO
                <optional parameters>
                <optional declarations>
                <statements>
        URCAM;
```

3. Parameters are allowed with the following restriction: VALUE, RETURNS, UNIV and parametric strings are disallowed.

4. TYPE, CONST, SHARES and VAR are allowed with the following restriction: STATIC is disallowed.

5. Scope rules are identical to those which would be applied to a PROC occurring at the same declaration point in the compilation.

6. A MACRO definition must precede its use. There are no such constructs as FORWARD, PTRs TO, or MID-defined MACROs.

7. Although there can be no intermodule references to MACROs, one may use INCLUDE to copy a MACRO definition from one module into another.

8. The MACRO call, or invocation, looks like a PROC call with the usual parameter checking enforced.

9. Invocations may be nested; i.e. one MACRO may invoke another. Recursive invocations are illegal.

10. RETURN statement is disallowed in MACRO.

17 IMPLEMENTATION OF MACRO FACILITY (Continued)

11. Example:

```
                  data
                  DATA
                        datum 0..999999 := 0;

                  macro
                  MACRO (p CONST 0 .. 10);
                        SHARES data;
                        IF    p ¬= 0
                        THEN
                              datum +:= p;
                        FI;
                  ORCAM;

                  proc
                  PROC;
                        VAR j 1 .. 10;

                        call_random (j);
                        macro (j);
             % becomes  IF    j ¬= 0
             %          THEN
             %               datum +:= j;
             %          FI;

                        macro (5);
             % becomes  IF    5 ¬= 0
             %          THEN
             %               datum +:= 5;
             %          FI;
                  CORP;
```

## 18 BLOCK DEFINITIONS

### 18.1 DATA DEFINITIONS

The function of the DATA definition is to associate an identifier with a group or block of variables. This definition is coded in the MIC component of a program or in a program module. In a program module, this definition is coded outside any of the module's procedures.

The syntax of the DATA definition is:

```
|
|
\___block name:identifier____DATA_____
                                      \
   ___________________________________/
  /   _______ , ______________________________________
  |  /                 \                              \
  \__\__var:identifier__/__type________________________/__
                                \__ := __const:expr__/     \
                                                           |
                                                           |
```

This definition is coded with the name of the block followed by the reserved word DATA. Following this is the variable name, its data type and an optional constant expression. The constant expression is used to initialize the variable.

The word "filler" is an identifier denoting an unused field. Its use is discussed under Structures, 13.3.3.

DATA block variables are STATIC; they are created and initialized once. Their lifetime is the execution time of the program no matter where the DATA block is coded.

The DATA block (the block name and its variables) has no implicit scope. Scope must be explicitly established for the DATA block.

A SHARES declaration must be coded in each module procedure that wishes to access the DATA block. In this case, the scope of the DATA block is every module that KNOWS and module procedure that SHARES the DATA block.

If the DATA block is coded in a module, a SHARES declaration must be coded in each procedure that is to access the DATA block. Here, the scope of the DATA block is every procedure that SHAREs the DATA block.

## 18.1 DATA DEFINITIONS (Continued)

Data blocks are not coded in procedures. The variable declaration VAR is used for this purpose (see 15.4).

Examples:

```
    icm_mid
    PROG                        % This example illustrates creation of
                                % DATA blocks in a MID component
          {icm_utility} KNOWS
          token_info
          DATA
               word     NAME,
               delim    BOOLEAN;

          {icm_utility} KNOWS
          scan_info
               cbuf     STRING(80) := "%",
               cptr     1..73      := 1,
               eof      BOOLEAN    := false;

          {icm_utility} KNOWS
          names
          DATA
               old_icm,
               old_pack,
               new_icm,
               new_pack   FILE_NAME,
               lib_name   NAME;
```

The examples in Section 15.5 (SHARES DECLARATIONS) illustrate these same DATA blocks as they would appear inside the module portion of a program.

## 18.2 FILE DEFINITIONS

A file definition associates an identifier with a group of files. This definition may appear in the MID of the program or in an individual module outside all procedures.

The syntax is:

```
file definition
|                                    __________ , _________
|  file block name:                 /                       \
\_____identifier______FILE___\___file description___/_________
                                                              \
                                                              |
                                                              |
```

18.2 FILE DEFINITIONS (Continued)

```
file description
|
|     filename:
\_____identifier_____file attr spec____OF______type___________
                                   \____/                     \
                                                               |
                                                               |


file attr spec
|
|                 ___________________ , _________________
|                /                                       \
\_____ [  ____\__attribute name__ = __attribute value__/__ ]  __
                                \___________________/          \
                                                                |
                                                                |
```

The file attribute specification serves to set up information about the logical and physical files. The attribute names and their values are described in Appendix C - File Attributes.

File descriptions are static; they are created once and exist throughout the lifetime of the program. A given file description might be changed to describe several different files during program execution.

The scope of the file block is any module that appears in its KNOWS list (see 12.3), if declared in the MID, or the procedures of the module, if declared in the module. The scope of the files in the file block includes every procedure that SHARES the block.

Examples:

```
    read_and_print_files
    FILE
         reader  [MYUSE = IN, KIND = READER]
                 OF CARD,
         printer [MYUSE = OUT, KIND = PRINTER]
                 OF PRINT_LINE;

    work_file
    FILE
         work [MYUSE= OI, KIND = DISK,
               BLOCKSIZE = 100 * WORK_RECORD.SIZE,
               BUFFERS = 2, ACCESSMODE =
               RANDOM, PROTECTION = TEMPORARY]
              OF WORK_RECORD;
```

18.2 FILE DEFINITIONS (Continued)

FILE ATTRIBUTES AND THEIR VALUES

Below is a list of the currently implemented file attributes and their range of values. Constant expressions can be used to set the file attributes where applicable. For additional information concerning the meaning and usage of file attributes, refer to CSG File Handling Standard 1955-2926.

| Attribute | Values | Default |
|---|---|---|
| ACCESSMODE | SEQUENTIAL or RANDOM | SEQUENTIAL |
| AREAS | 0..99 (0 = 100) | 20 |
| AREASIZE | 1..99999999 (number of records) | 1000 |
| AUTOPRINT | TRUE or FALSE | FALSE |
| BACKUPKIND | DISK, DISKPACK, DONTCARE or TAPE | <system default> |
| BACKUPPERMITTED | DONTBACKUP, MUSTBACKUP or DONTCARE | DONTCARE |
| BLOCK | 0..99999999 | (Read Only) |
| + BLOCKSIZE | 1..999999 (in digits) | Depends on record size |
| + BUFFERS | 0..9 (actual buffers-1) | 0 ( => 1 buffer) |
| CREATIONDATE | DISPLAY 0..99999 | (Read Only) |
| CURRENTBLOCK | 1..999999 | (Read Only) |
| CURRENTRECORD | 1..39996 | (Read Only) |
| CYCLE | DISPLAY 0..99 | 01 |
| DIRECTION | FORWARD or REVERSE | FORWARD |
| EXTMODE | EBCDIC (only for output punch files) | <Normal> |
| FAMILYNAME | STRING (6) | <none> |
| FILENAME | STRING (6) | <truncated filename> |
| FILESTATUS | OPEN or CLOSED | (Read Only) |
| FORCEIO | TRUE or FALSE | FALSE |
| FORMS | TRUE or FALSE | FALSE |
| INTNAME | STRING (6) | <truncated filename in uppercase> |
| KIND | DISK, DISKPACK, PRINTER, PUNCH, READER, REMOTE or TAPE | If OUT - PRINTER ELSE READER |
| LABEL | EBCDICLABEL or OMITTED | EBCDICLABEL |
| LASTRECORD | 0..99999999 | (Read Only) |
| + MAXRECSIZE | 1..39996 | <record size> |
| MYUSE | IN, OUT, IO or OI | IO |
| NEXTRECORD | 0..99999999 | (Read Only) |

18.2 FILE DEFINITIONS (Continued)

+ Increasing the values of these attributes beyond their compile-time values will destroy portions of the program at run-time.

18.2 FILE DEFINITIONS (Continued)

| Attribute | Values | Default |
|---|---|---|
| OPTIONAL | TRUE or FALSE | FALSE |
| PARITY | EVEN or ODD (only for 7 track tape) | EVEN |
| PROTECTION | TEMPORARY or ABNORMALSAVE | TEMPORARY |
| SAVEFACTOR | 0 through 999 | 999 |
| SECURITYFAMILY | STRING (6) | "DISK" |
| SECURITYGUARD | STRING (6) | <none> |
| SECURITYTYPE | PUBLIC, PRIVATE, GUARDED, NONE, or DEFAULT | DEFAULT |
| SECURITYUSE | IN, OUT, IO, or SECURED | IO |
| SENSITIVEDATA | TRUE or FALSE | FALSE |
| SERIALNO | DISPLAY 0..99999 | (Read Only) |
| SINGLEUNIT | TRUE or FALSE | FALSE |
| UNIQUENAME | NULL, PROCESSOR or WORK | NULL |
| VOLUMEINDEX | 1 through 999 | 1 |

+ Increasing the values of these attributes beyond their compile-time values will destroy portions of the program at run-time.

18.3 NETWORK SYSTEMS PROCESSOR (NSP) FILE DEFINITIONS

The syntax for a nsp_file_block is:

```
nsp_file_block
|
|                                     _________ , ___________
|                                    /                       \
\__nsp_file_block_name __NSP__\__nsp_file_description__/__ ; __
                                                               \
                                                               |
                                                               |
```

The syntax for a nsp_file_description:

```
nsp_file_description
|
|
\___nsp_file_name____nsp_file_attribute_specification_____
                                                          \
                                                          |
                                                          |
```

## 18.3 NETWORK SYSTEMS PROCESSOR (NSP) FILE DEFINITIONS (Continued)

The syntax for a nsp_file_attribute_spec is:

```
nsp_file_attribute_spec
|
|              _________________ , _________________
\__ [ __\__attr_name______________________________/__ ] __
                  \                          /
                   \___ = _attr_value____/                  \
                                                            |
                                                            |
```

The predefined module port_io is available for nsp file I/O with the following predefined procedures and expected parameters.

| | |
|---|---|
| open_wait | (nsp_file_name) |
| close_retain_wait | (nsp_file_name) |
| close_retain_dont_wait | (nsp_file_name) |
| close_release_wait | (nsp_file_name) |
| close_release_dont_wait | (nsp_file_name) |
| get_message | (nsp_file_name, pointer_buffer) |
| dump | (nsp_file_name, pointer_buffer) |
| read_state | (nsp_file_name, pointer_buffer) |
| discontinue | (nsp_file_name, pointer_buffer) |
| test_id | (nsp_file name, pointer_buffer) |
| send_message | (nsp_file_name, pointer_buffer) |
| load_first | (nsp_file_name, pointer_buffer) |
| load_intermediate | (nsp_file_name, pointer_buffer) |
| load_last | (nsp_file_name, pointer_buffer) |
| soft_clear | (nsp_file_name, pointer_buffer) |
| dlp_communicate | (nsp_file_name, pointer_buffer) |
| ack_message | (nsp_file_name, pointer_buffer) |
| reject_message | (nsp_file_name, pointer_buffer) |

NOTE: 1. The use of attributes as expressions is similar to that for file attributes (e.g., ASSERT nsp_file_name.FILESTATE = OPENED).

2. NSP attributes may not be passed as VAR parameters.

Pointer_buffer is a user_declared variable. It is a structure whose first field is an 8-digit UN type and whose second field is a 6-digit UN type.

### 18.3.1 NSP File Attributes and Their Values

Below is a list of the currently implemented NSP file attributes and their range of values. Constant expressions can be used to set the NSP file attributes where applicable. For additional information concerning the meaning and usage of NSP file attributes, refer to UIO/DATACOMM documents.

Availability

1 = declarable
2 = gettable
3 = settable

| Attribute | Values | Available | | |
|---|---|---|---|---|
| ATTRERR | TRUE, FALSE | | 2 | 3 |
| BLOCKSIZE | 2..999998 | 1 | 2 | 3 |
| BLOCKSTRUCTURE | FIXED | 1 | 2 | 3 |
| BUFFERS | 1..99 | 1 | 2 | 3 |
| FAMILYINDEX | 0..9999 | 1 | 2 | 3 |
| FAMILYNAME | STRING (17) | 1 | 2 | 3 |
| FILENAME | STRING (100) | 1 | 2 | 3 |
| FILESTATE | CLOSED, AWAITINGHOST, OFFERED, OPENED, SHUTTINGDOWN, BLOCKED, CLOSEPENDING, DEACTIVATIONPENDING, DEACTIVATED, DENIED, POSTPONED, DENIEDILLEGALUSE | | 2 | 3 |
| INTNAME | STRING(17) | 1 | 2 | 3 |
| IOCANCEL | TRUE, FALSE | | 2 | 3 |
| IOCOMPLETE | TRUE, FALSE | | 2 | |
| IOEOF | TRUE, FALSE | | 2 | |
| IOERRORTYPE | 0..9999 | | 2 | |
| IOLENGTH | 0..999999 | | 2 | |
| IOMASK | STRING(16) OF HEX | | 2 | 3 |
| IOPENDING | TRUE, FALSE | | 2 | |
| IORECORDNUM | 0..999999999999 | | 2 | |
| IORESULT | STRING(16) OF HEX | | 2 | |
| MAXRECSIZE | 2..999998 | 1 | 2 | 3 |
| MYUSE | IN, OUT, IO | 1 | 2 | 3 |
| OPEN | TRUE, FALSE | 1 | 2 | 3 |
| OTHERUSE | IN, OUT, IO, SECURED | 1 | 2 | 3 |

## 18.4 PORT FILE DEFINITIONS

The syntax for a port_block is:

```
port_block
|
|                                   _________ , __________
|                                  /                      \
\___port_block_name__PORT___\___port_description___/___ ; ___
                                                               \
                                                               |
                                                               |
```

The syntax for a port_description is:

```
port_description
|
|
\__________port_name__________port_attribute_spec____
                                                     \
                                                     |
                                                     |
```

The syntax for a port_attribute_spec is:

```
port_attribute_spec
|
|         ________________ , _________________________________
|        /                                                    \
\_ [ _\_attr_name____________________________________________/_ ] __
              \                       /\                 /          \
               \_ [ _const_expr_ ] _/  \_ = _attr_value_/           |
                                                                     |
                                                                     |
```

The predefined module port_io is available for port file I/O with the following predefined procedures and expected parameters.

| | |
|---|---|
| open_wait | ( port_name [expression] ) |
| open_offer | ( port_name [expression] ) |
| open_available | ( port_name [expression] ) |
| close_retain_wait | ( port_name [expression] ) |
| close_retain_dont_wait | ( port_name [expression] ) |
| close_release_wait | ( port_name [expression] ) |
| close_release_dont_wait | ( port_name [expression] ) |
| read_wait | ( port_name [expression], record ) |
| read_dont_wait | ( port_name [expression], record ) |
| write_wait | ( port_name [expression], record ) |
| write_dont_wait | ( port_name [expression], record ) |

Miscellaneous notes:

18.4 PORT FILE DEFINITIONS (Continued)

a. Attr_names and attr_values are defined in BNA (Burroughs Network Architecture) documents.

b. Attribute names appearing with an index are treated as subfile attribute names. Attribute names appearing with no index are treated as port attribute names.

c. If MAXRECSIZE is not declared, the default value is 19998 bytes.

d. Within the declaration, specification of port attributes must precede specification of subfile attributes.

All varieties of port I/O read and write may take the IF EOF option.

The use of attributes as expressions is similar to that for file attributes.

(e.g. ASSERT port_name [1].SUBFILEERROR = NOERROR).

The range of both const_expr and expression is 0 ..9999.

Port attributes may not be passed as VAR parameters.

Example

```
prog_a_mod
MOD

port_block
PORT
   port1 [MAXRECSIZE = 200, INTNAME = "PORT_A1",
         TITLE = "TEST_PORT", MYNAME = "PROG_A1",
         MAXSUBFILES = 2, SECURITYUSE = IO,
         YOURNAME [1] = "PROG_A2", YOURNAME [2] = PROG_B1" ],

   port2 [MAXRECSIZE = 150, INTNAME = "PORT_A2",
         TITLE = "TEST_PORT", MYNAME = "PROG_A2",
         MAXSUBFILES = 20, SECURITY = IO,
         YOURNAME [10] = "PROG_B2", YOURNAME [20] = "PROG_A1"];

prog_a_entry
PROC;

   SHARES
         port_block;
```

18.4 PORT FILE DEFINITIONS (Continued)

```
          VAR
             give          1..10,
             read_rec_a,
             write_rec_a  STRING (40);

          write_rec_a := "Hello, are you there?";
          port_io.open_wait (port1[1]);
          port_io.write_dont_wait (port1[1], write_rec_a);

          port_io.open_wait (port2[20]);
          port_io.read_wait (port2[20], read_rec_a);

          port_io.close_release_wait (port1[1]);
          port_io.close_release_wait (port2[20]);

       CORP;

       DOM;
```

## 18.4.1 Port Attributes and Their Values

Below is a list of the currently implemented port attributes and their range of values. Constant expressions can be used to set the port and subport attributes where applicable. For additional information concerning the meaning and usage of port attributes, refer to B2000/3000/4000 Burroughs Network Architecture documents.

Availability

1 = declarable
2 = gettable
3 = settable

| Attribute | Values | Available For Ports | Available For Subports |
|---|---|---|---|
| ACCEPTABLECHARSET | DONTCARE | 1 2 3 | |
| ACTUALCHARSET | ASCII,EBCDIC | | 2 |
| ATTERR | STRING(2) OF HEX | 2 | |
| BLOCKSTRUCTURE | FIXED,EXTERNAL | 1 2 3 | |
| CENSUS | 0..999999 | 2 | 2 |
| CHANGEEVENT | TRUE,FALSE | 2 | 2 |
| CHANGEDSUBFILE | 0..9999 | 2 | |
| COMPRESSION | TRUE,FALSE | | 1 2 3 |
| COMPRESSIONPOSSIBLE | TRUE,FALSE | | 2 |
| CURRENTRECORD | 2..19998 | | 2 |

18.4.1 <u>Port Attributes and Their Values</u> (Continued)

| Attribute | Values | Available For Ports | Available For Subports |
|---|---|---|---|
| DIALOGPROTOCOLLEVEL | 1..255 | | 2 |
| FILESTATE | AWAITINGHOST, OFFERED, OPENED, SHUTTINGDOWN, BLOCKED, CLOSEPENDING, DEACTIVATIONPENDING, DEACTIVATED, CLOSED | | 2 |
| + HISCODEFILEFAMILY | STRING(6) | | 2 |
| + HISCODEFILENAME | STRING(6) | | 2 |
| + HISCOMPRESSIONFLAG | STRING(1) OF HEX | | 2 3 |
| + HISFLOWSTATUS | BOOLEAN | | 3 |
| + HISMYNAME | STRING(100) | | 2 |
| + HISNULLFLAGS | STRING(1) OF HEX | | 2 |
| + HISOPENTYPE | 0..99 | | 2 |
| + HISPORTADDRESS | STRING(4) OF HEX | | 2 |
| + HISSUBFILEERROR | NOERROR, DISCONNECTED, DATALOST, NOBUFFER, NOFILEFOUND, UNREACHABLEHOST | | 3 |
| + HISSUBPORTADDRESS | STRING(4) OF HEX | | 2 |
| + HISUSERCODE | STRING(17) | | 2 |
| + HISYOURNAME | STRING(100) | | 2 |
| HOSTNAME | STRING(17) | | 1 2 3 |
| INPUTEVENT | TRUE,FALSE | 2 | 2 |
| INTNAME | STRING(17) | 1 2 3 | |
| MAXCENSUS | 0..9999 | | 1 2 3 |
| MAXRECSIZE | 2..19998 | 1 2 3 | 2 |
| MAXSUBFILES | 1..9999 | 1 2 3 | |
| MYHOSTNAME | STRING(17) | 2 | |
| MYNAME | STRING(100) | 1 2 3 | |
| + MYPORTADDRESS | STRING(4) OF HEX | | 2 |
| MYUSERCODE | STRING(17) | 1 | 2 3 |
| OUTPUTEVENT | TRUE,FALSE | | 2 |
| + PLMCHARACTERSETS | STRING(1) OF HEX | | 2 3 |
| + PLMMATCHRESP | BOOLEAN | | 3 |
| + PLMMAXMSGTEXTSIZE | 2..19998 | | 3 |
| + PLMMYCODEFILEFAMILY | STRING(6) | | 2 3 |
| + PLMMYCODEFILENAME | STRING(6) | | 2 3 |
| + PLMMYHOSTNAME | STRING(17) | | 2 3 |

+ These attributes are only valid for the BNA software.

18.4.1 Port Attributes and Their Values (Continued)

| | Attribute | Values | For Ports | | | For Subport | | |
|---|---|---|---|---|---|---|---|---|
| + | PLMMYNAME | STRING(100) | | | | | 2 | 3 |
| + | PLMSECURITYGUARD | STRING(6) | | | | | 2 | 3 |
| + | PLMSECURITYTYPE | GUARDED, PRIVATE, PUBLIC | | | | | 2 | 3 |
| + | PLMSECURITYUSE | IO | | | | | 2 | 3 |
| + | PLMTITLE | STRING(17) | | | | | 2 | 3 |
| | PORTRESULTS | STRING(100) OF HEX | | 2 | | | 2 | |
| | READYEVENT | TRUE,FALSE | | 2 | | | | |
| | READYSUBFILE | 0..9999 | | 2 | | | | |
| | SECURITYGUARD | STRING(100) | 1 | 2 | 3 | | | |
| | SECURITYTYPE | PUBLIC,PRIVATE,GUARDED | 1 | 2 | 3 | | | |
| | SECURITYUSE | IO | 1 | 2 | 3 | | | |
| | SUBFILEERROR | NOERROR, DISCONNECTED,DATALOST, NOBUFFER,NOFILEFOUND, UNREACHABLEHOST | | | | | 2 | |
| | TITLE | STRING(17) | 1 | 2 | 3 | | | |
| | YOURNAME | STRING(100) | | | | 1 | 2 | 3 |
| | YOURUSERCODE | STRING(17) | | | | 1 | 2 | 3 |

* These attributes are only valid for the BNA software.

## 19 EXPRESSIONS

Expressions are comprised of operators and operands. Operator precedence determines the order of evaluation of operands and sub-expressions. Except for parenthesizing, operators are evaluated in order of their precedence. A parenthesized expression is evaluated before any operator is applied to it.

Evaluation of expressions results in a single value. The type of the expression is determined from the operands and operators.

In most places in the language, expressions can appear wherever constants or variables can appear. The exceptions involve assignment, e.g., the left hand side of an assignment statement, and actual parameters where the formal parameters are modified by VAR. Some contexts, such as type descriptions, require that all expression components be constants.

## 19.1 OPERATORS

### 19.1.1 Monadic Operators

Monadic operators are operators which apply to one operand. These operators have the highest precedence.

| Operator Symbol | Operation | Operand Type | Result Type |
|---|---|---|---|
| ¬ | complement | boolean | boolean |
| ¬ | complement | HEX, string(n) of HEX | HEX, string(n) of HEX |
| + | (no effect) | arithmetic | arithmetic |
| - | negation | arithmetic | arithmetic |
| - | complement | set | set |

### 19.1.2 Dyadic Operators

Dyadic operators are operators which apply to two operands. The following table lists the operators in groups from highest to lowest precedence. Within a group the operators have the same precedence.

## 19 EXPRESSIONS

Expressions are comprised of operators and operands. Operator precedence determines the order of evaluation of operands and sub-expressions. Except for parenthesizing, operators are evaluated in order of their precedence. A parenthesized expression is evaluated before any operator is applied to it.

Evaluation of expressions results in a single value. The type of the expression is determined from the operands and operators.

In most places in the language, expressions can appear wherever constants or variables can appear. The exceptions involve assignment, e.g., the left hand side of an assignment statement, and actual parameters where the formal parameters are modified by VAR. Some contexts, such as type descriptions, require that all expression components be constants.

## 19.1 OPERATORS

### 19.1.1 Monadic Operators

Monadic operators are operators which apply to one operand. These operators have the highest precedence.

| Operator Symbol | Operation | Operand Type | Result Type |
|---|---|---|---|
| ¬ | complement | boolean | boolean |
| ¬ | complement | HEX, string(n) of HEX | HEX, string(n) of HEX |
| + | (no effect) | arithmetic | arithmetic |
| - | negation | arithmetic | arithmetic |
| - | complement | set | set |

### 19.1.2 Dyadic Operators

Dyadic operators are operators which apply to two operands. The following table lists the operators in groups from highest to lowest precedence. Within a group the operators have the same precedence.

19.1.2 Dyadic Operators (Continued)

Some symbols represent several distinct operations. The operation to be applied, in this event, is determined by the operand types.

19.1.2 Dyadic Operators (Continued)

| | Operator Symbol | Operation | Operand Types (left, right) | Result type |
|---|---|---|---|---|
| | REM | remainder | integer | integer |
| | * | multiply | arithmetic | arithmetic |
| | * | intersection | sets | sets |
| | / | divide | arithmetic | arithmetic |
| | + | addition | arithmetic | arithmetic |
| | + | union | sets | sets |
| | + | concatenation | strings | strings |
| | - | subtraction | arithmetic | arithmetic |
| | - | difference | sets | sets |
| | < | less than | ordered scalars | boolean |
| | < | alphabetic < | ordered strings | boolean |
| * | < | proper subset | sets | boolean |
| | > | greater than | ordered scalars | boolean |
| | > | alphabetic > | ordered strings | boolean |
| * | > | proper superset | sets | boolean |
| | <= | less than or equal | ordered scalars | boolean |
| | <= | alphabetic <= | ordered strings | boolean |
| * | <= | subset | sets | boolean |
| | >= | greater than or equal | ordered scalars | boolean |
| | >= | alphabetic >= | ordered strings | boolean |
| * | >= | superset | sets | boolean |
| | = | equal to | all types | boolean |
| | ¬= | not equal to | all types | boolean |
| | IN | subrange inclusion | scalar, subrange | boolean |
| | IN | set membership | scalar, set | boolean |
| | ¬IN | subrange exclusion | scalar, subrange | boolean |
| | ¬IN | set exclusion | scalar, set | boolean |
| | & | logical and | boolean | boolean |
| | & | logical and | HEX, STRING(n) of HEX | HEX, STRING(n) of HEX |
| | && | conditional & | boolean | boolean |
| | \| | logical or | boolean | boolean |
| | \| | logical or | HEX, STRING(n) of HEX | HEX, STRING(n) of HEX |
| | \|\| | conditional \| | boolean | boolean |
| | # | exclusive or | boolean | boolean |
| | # | exclusive or | HEX, STRING(n) | HEX, STRING(N) |

## 19.1.2 Dyadic Operators (Continued)

|  |  |  |  |
|---|---|---|---|
| # | symmetric difference | of HEX sets | of HEX sets |

In the table above, arithmetic refers to integer and real. When both operands are integer, the result is integer.

* When both operands are real or the operands are mixed, the result is real.

The order of evaluating the left and right operands is not guaranteed and may change, subject to compiler optimization, except for the conditional logical operators ( && and || ). The left operand is evaluated first, then the right operand is evaluated only if necessary; that is, if the left operand of && is true or the left operand of || is false. These operators allow more readable code in certain situations and still avoid certain run-time errors.

Example:

```
    IF pointer ¬= nil
    THEN
        IF pointer@ = 0
        THEN
            do_something;
        FI;
    FI;
```

can be written as

```
    IF pointer ¬= nil && pointer@ = 0
    THEN
        do_something
    FI;
```

## 19.2 OPERANDS

Operands may be primaries, denotations, parenthesized expressions, a range of values or inquiries.

### 19.2.1 Primaries

A primary is a variable, constant, cast, function call, or a file attribute inquiry followed by appropriate selections.

## Selections

A selection modifies a name to achieve one of five distinct purposes, namely, to invoke a function, to select a field component of a structure, to dereference a pointer, to subscript or slice an array, or to select a substring of a string. Since selections can be applied to the result of a function call, the parameter list is included in the selection syntax diagram shown below.

```
|
|      ___________________________________________________
|   /                                                      \
\__\_______ . ____field:identifier___________________/____
     \                   _________ , _________          /      \
     |              /    subscript/slice:     \         |      |
     |\____ [ ___\_______indexing_______/___ ] ____/|      |
     |                                                  |      |
     |\____ @ __________________________________________/|      |
     |      \             ________ , ________       /   |      |
     |       \          / actual parameter: \      /    |      |
     |        \__ ( __ \___ expression ____/__ ) _/     |      |
     |                                                  |      |
     \________ . __attribute name:indicant__________/      |
                                                                |
                                                                |
```

```
indexing
|
|
|\___expr______________________________________________________
|          \                    /                                \
|           \___ .. _____simple expr___/                          |
|            \__ :: __/                                           |
|                                                                 |
\____indicant__ . __RANGE_____________________________________    |
                                                                \ |
                                                                  |
                                                                  |
```

Examples:

```
    a[i] . p @
    strng [1::3]
    arr [2..5]
```

The first example is a one-dimensional array 'a' subscripted to yield a structure. A field 'p' of type pointer is selected and then dereferenced. The second example is a substring of 'strng'. The third example is an array slice of a one-dimensional array.

Access Attributes

The access of a primary (before selection) may be read/write (VAR) or read-only (CONST). The access of a primary after selection remains the same as its access before selection, with the exception of dereferenced pointers. A dereferenced pointer takes on the access of the referenced value instead of retaining the pointer's own access (see 13.4).

The type of a primary is determined from the declared type of the base name and the selections applied to it.

19.2.1.1 Variables

Use of a variable may involve all or part of a data structure.

```
|
|
\_________variable:identifier________selections___________
                                 \_________________/       \
                                                           |
                                                           |
```

Full generality is permitted in selections.

Examples:

```
     symb_tabl [i]. declr@. idtype
     table_ptr@ [i-1, m*size]. cp
```

19.2.1.2 Constants

Certain contexts require that only constants be used as operands of an expression. Some standard functions (abs, fill_array, fill_string, length, lwb, upb) may be used in constant expressions, if their value can be determined at compile time.

Named constants are defined by means of constant definitions.

If the constant has an aggregate type, selections may be applied to designate a component.

The access is read-only (CONST). The type is determined from the declared type and the selections applied.

19.2.1.3 Casts

In certain contexts, e.g., assignments, parameter passing, and expressions, it may be necessary to convert a value of one type to a value of a related type (see 13.7). Context may determine an implicit conversion (i.e., a 'coercion'). If the context is ambiguous, the desired type must be supplied explicitly. This construct is called a 'cast'.

Casts provide an unambiguous context for conversion where it would not otherwise exist. The syntax of a cast is:

```
|
|
\_____cast type:indicant____ ( ____expression____ ) ______
                                                          \
                                                          |
                                                          |
```

Appropriate selections may be applied to a cast. The access of a cast is read-only. The type of a cast is that of the cast indicant.

The value delivered by the expression must be convertible to the type represented by the indicant. This may require a series of casts. All coercions of the language are available as casts.

Examples:

```
TYPE SIGNED = -99..99,
     UNSIGNED = 0..99,
     DISP_UNSGN = DISPLAY UNSIGNED,
     STR2 = STRING (2) OF CHAR,
     STR3 = STRING (3) OF CHAR;

VAR si SIGNED := 25,
    ch3 STR3;

ch3 := STR2(DISP_UNSGN (UNSIGNED(si)));
     % Display can only be applied to unsigned integers.
     % The STR2 cast creates a string before assignment
     % to left justify the value. ch3 contains "25 ".
```

19.2.1.4 Function Calls

A function call invokes a routine which returns a value. The routine invoked is the one that is associated with the function name by the procedure definition. The call passes a particular set of actual parameters. The operators

19.2.1.4 Function Calls (Continued)

allowed on this operand (function call) are those appropriate for the return type of the function.

The type of the call is the type of the RETURN value specified in the definition of the function. A function call cannot be used where read/write access is required.

```
|
|           function:
|\___________ident_________________________
| module:   /     / \    ____ , _____     /  \
|\_ident__ . __/  | |     /act. param: \  |  |
|                 | \__ ( __\____expr____/__ ) __/  |
\__function:primary____/      \_____________/      |
                                                    |
                                                    |
                                                    |
```

A function primary, involving the dereferencing of a variable of type PTR TO PROC, may be used.

If the call is qualified by a module name, an intermodule call is made.

The number of actual parameters must be the same as the number of formal parameters in the procedure definition. If a parameter access is read/write (VAR), the type of the actual parameter must be equivalent to the type of the corresponding formal parameter. If the access is read-only (CONST), the actual is coerced to the formal type if necessary and if possible.

When the access of the formal parameter is read-only, the actual parameter may be an expression, variable or constant. When the access of the formal is read/write, the actual must also have read/write access, i.e., be a variable. The order in which the actual parameters are evaluated is undefined.

Use of the modifier VAR with a formal parameter means that changes to the value of the formal parameter are also changes to the value of the actual parameter. CONST, the default, means the actual parameter may not be changed via the formal parameter.

An empty parameter list may be used for documentation or clarity.

Selections may be applied to a function call.

19.2.1.4 Function Calls (Continued)

Examples:

```
abs (x)
length (string_name)
current_index +:= length (input_string);
token := get_token;
lwb (array_name, subscript)
round (y)
upb (array_name,1)
```

19.2.1.5 File Attribute Inquiries

File attribute inquiries select various attribute values from a file and may provide a destination where new attribute values can be assigned to a file. The syntax is:

```
|
|       filename:                      attribute name:
\_______identifier________ . ___________indicant_____________
                                                              \
                                                              |
```

The type of the primary is determined by the attribute chosen. The attribute chosen also determines the access to the value. Some file attributes are read only, while others may be correctly assigned values only when the file is in a particular state (open, assigned, etc.). Some file attributes may return legitimate values only when the file is open or assigned or in some other specific state. See Appendix C - File Attributes for details.

Examples:

```
card.BLOCKSIZE
print.BACKUPKIND
work.UNIQUENAME
```

19.2.2 Denotations

Denotations are representations of values. Denotations are strings (one or more characters), numbers (integer and real), symbolic identifiers, and the standard constant identifiers (nil, true, false). Each denotation has an 'a priori' type:

19.2.2 Denotations (Continued)

| | <u>denotation</u> | <u>type</u> |
|---|---|---|
| | integer n | subrange n..n |
| * | real number | REAL |
| | symbolic id | as defined |
| | string | STRING (actual length) OF EBCDIC |
| | nil | PTR (any access, level, type) |
| | true, false | BOOLEAN |

19.2.2.1 Aggregate Denotations

Aggregate denotations have the form

```
|
|            _______ , ________
|           /                  \
\___ [ ___\_____expression___/__ ] _______________________
                                                           \
                                                           |
                                                           |
```

They represent structures and arrays (see 13.3.1 and 13.3.3). The type is determined by context. The expressions are coerced to the component types if necessary.

The expressions may be constant or variable, although expressions associated with tag fields and data initialization must be constant.

Examples:

```
[1, 3, 5]
["a", 1, true, [true, false], nil]
```

19.2.3 Parenthesized Expressions

An expression enclosed in parentheses serves as a single operand. The syntax is:

```
|
|
\___ ( ______expression______ ) __________________________
                                                           \
                                                           |
                                                           |
```

19.2.4 Inquiries

Inquiries interrogate type attributes. The syntax is:

```
|
|
\_____type:indicant_____ . _____inquiry__________________
                                                         \
                                                         |
                                                         |
```

The type of an inquiry is the same as the type whose indicant is interrogated. The inquiry RANGE applies to scalar types. The result is an ordered or unordered collection or range of values, depending upon whether the scalar type is ordered or unordered. The inquiries are:

| | Type | Inquiry | Value |
|---|---|---|---|
| | ordered scalar | MIN | least (for numeric) or first (for symbolic) value of the type |
| | ordered scalar | MAX | greatest (for numeric) or last (for symbolic) value of the type |
| | scalar | RANGE | the collection of all values of the type |
| * | REAL | DELTA | smallest positive number for the type |
| * | REAL | EPSILON | smallest positive number such that 1.0 + EPSILON > 1.0 |
| | any type (except BIT or parametric) | SIZE | number of digits occupied by a data object of this type |

Examples:

```
OP_RANGE.MIN
MONTH.MAX
CHAR.RANGE
```

19.3 RANGE OF VALUES

A range of values is expressed by:

19.3 RANGE OF VALUES (Continued)

```
|
|
|\_____simple expr____ .. ____simple expr_______________
|                                           /           \
\______type:indicant___ . ____RANGE_______/             |
                                                        |
                                                        |
```

The expressions must be compatible ordered scalars. A range of values may be used as variant labels (*), CASE labels (*), a FOR range, an IN operand, set elements (*) and a slicing or substringing (*) selection. When used as labels, it must specify a range of constants.

## 20 DEBUGGING A SPRITE PROGRAM

Thanks to strong type checking and other compiler features, the SPRITE compiler catches as syntax errors many kinds of program logic problems which, in other languages, might be detected only as run-time errors - if at all! Despite this, you will find that debugging your program still takes time and effort, especially if you don't take advantage of all of the debugging tools which are available to help you.

We recommend that you take the following steps as you build your program, since they will allow you to make best use of your debugging time:

1. Use good, well-structured coding methods. You should try (within reason) to make many small procedures, rather than few large procedures, since most of your debugging aids involve procedures.

2. Avoid tricky coding methods whenever possible, especially STRUCtures with omitted tag fields. When you build your own pointers, or make use of SPRITE's failure to check the value of a tag field when making field selections, or do other sneaky (although sometimes necessary) things, you are bypassing any help that the SPRITE compiler can give you in detecting your errors.

3. Avoid resetting the SPRITE control card options DEBUGCALLS and ERRORCALLS (see Appendix I.5). These options reduce the debugging capabilities of your program at run-time.

4. Do not use the OPTION statement in BINDER to remove optional code during testing. You will need all of the help this optional code can give you. You might consider leaving this optional code in your program even for released products. Program errors cause much less trouble when they are caught immediately, and we can think of few products which have been released without uncaught errors in them.

5. Bind the SPRITE debug package into your program while you are testing and debugging it (see section 20.2.2). This package provides you with a lot of assistance in your debugging job. You must use the DEBUG statement in BINDER to set up the calls from your procedures to the debug module.

20 DEBUGGING A SPRITE PROGRAM (Continued)

6. Make use of the 'dbwrite' module to print the value of variables at critical places in your code (see section 20.2.4). The current debug package cannot print symbolic (or any other) representations of data in your program - a big weakness! You can save a lot of debugging time if you plan data printouts well.

   Note that you now have the problem of maintaining debug and non-debug versions of your ICMs. You will also need to comment out the debug code in your modules when you produce non-debug ICMs. This task can be eased by using the f_dbwrite module within ASSERT statements.

7. Write specialized code in your program to control the debugging actions taken by the debug module (see section 20.2.6). You may find it worth the effort in some cases to create your own interface to the debugging package to give yourself more flexibility in debugging.

8. Write specialized procedures in your program to print tables, structures, symbolic values, or anything else you think is worth the effort.

Many of the errors you will encounter while debugging your program are simple run-time detected errors. Other errors express themselves as strange program behavior. They efficiently avoid detection, and you may require powerful debugging aids to find them.

## 20.1 RUN-TIME PROGRAM ERRORS

You will find that many of your programming errors cause your program to encounter a run-time error situation and abort. You will also find that these common run-time errors are usually easy to fix.

Run-time errors are found by:

1. code generated by the SPRITE compiler,

2. the processor (hardware), or

3. the Operating System (MCP).

We discuss each of these below.

### 20.1.1 Run-Time SPRITE Errors

When code generated by the SPRITE compiler detects a run-time error, it calls the err module to print an error message. Run-time errors include:

CASE SELECTOR OUT OF RANGE

ASSERTION FAILED

NO PROC RESULT RETURNED

SUBSCRIPT OUT OF RANGE

STRING TRUNCATION FAILED

VALUE OUT OF RANGE

CONVERSION FAILED

STRING IS NON-NUMERIC

STRING OFFSET OUT OF RANGE

STRING LENGTH OUT OF RANGE

HEAP AND STACK COLLISION

HEAP OVERFLOW

HEX STRING LENGTHS NOT EQUAL

The err module also prints the name of the module in which the error happened and the statement number and line number of the source program statement in which the error happened. You should have no trouble fixing these kinds of run-time errors.

If you use the BINDER's OPTION statement to remove various levels of optional debug code, your program probably will not call the err module. The subsequent execution of your program becomes unpredictable, and you may have great trouble finding out exactly what went wrong.

### 20.1.2 Run-Time Hardware Errors

The hardware processor may detect one of the following kinds of errors during the execution of your program:

INSTR TIME OUT

20.1.2 Run-Time Hardware Errors (Continued)

ADDR ERR

MEM PAR

INV INSTR

INV ARITHMETIC DATA

Unless your program is armed, the MCP will display an error message on the SPO telling the type of error which happened, the address of the instruction in which the error happened, and the overlay (or segment) which contains the instruction. You will find a BIND listing useful to determine which of your procedures had the error.

If you are using the debug package, or if your program is armed and calls arm.get_error_message, you will get a message from your program which is similar to that from the MCP. However, the message from your program may include additional error information as described below.

You need not worry about a memory parity error, as this is caused by a hardware failure - not a bug in your program.

Address errors have several common causes. If your program's next stack address (at location 40, size 6) is beyond your program's limit register, your program has had a stack overflow, which you may usually correct by re-executing your program with more core (you may also wish to change the STACKSPACE statement in your BIND deck). If any of your program's index registers contain EEEEEE, your program was probably attempting to dereference a nil pointer. If any of your program's index registers contain FFFFFF, your program was probably attempting to dereference an uninitialized pointer.

If you use the ERRORCALLS control card option in your SPRITE compile, your program will get an invalid instruction when one of SPRITE's run-time errors (described in section 20.1.1) occurs. Your program prints one of the error messages described in section 20.1.1 (but not including the module name and statement numbers) if you are using the debug package or if your program uses the arm module to handle program errors and to print the resulting error message. Otherwise you will need a memory dump to determine what happened. The invalid instruction has an op-code of EC (error code), and the two digit field which follows is an index into the list of errors in section 20.1.1.

20.1.2 Run-Time Hardware Errors (Continued)

An invalid arithmetic data error says that your program is doing arithmetic on a variable which has A, B, C, D, E, or F digits in its value. You may usually assume that you are dealing with an uninitialized variable.

When you are using SPRITE's debug package, you need not worry so much about this, as debug will give you an error message describing the run-time error in great detail.

### 20.1.3 Run-Time MCP Errors

When the MCP detects an error in your program, it usually prints an error message giving the type of error, instruction address, and the segment number where the error happened. If you are using the debug package, or your program is armed, the MCP allows your program to execute at its error-handling routine rather than print the message. In either case, the MCP puts a result descriptor into your program (address 80, size 4) which describes the type of error it found.

The MCP typically detects errors in the way that your program uses MCP functions (files, stoque, date, time, display, etc.). The message

```
<prog-name>=<mix#> INV OPEN <filename> - INV RECD SIZE
```

is an example of messages which result from this type of error. The result descriptor which the MCP passes to your program has the format '9xx0', where 'xx' describes the particular error which the MCP found (see MCPVI PRODUCT INFORMATION UPDATE for a list of these errors).

## 20.2 SPRITE DEBUG PACKAGE

The debug package provided by SPRITE helps you in debugging the more subtle kinds of errors that you may find in your program. The package consists of the debug module and the subordinate modules which it needs to accomplish its task.

Note that you need not make any changes to your MID to use this debug package, unless you make explicit calls (such as to dbwrite) in your modules.

The following sections describe how you put the debug package into your program and how you use the debug package when you execute your program.

20.2.1 Debug in SPRITE Modules

Unless you reset the DEBUGCALLS control card option (see Appendix I), SPRITE will generate optional debug calls at procedure entry points and procedure exit points.

Unless you reset the DEBUGCALLS control card option or reset the DBSTMTCALLS control card option (see Appendix I), SPRITE will generate optional debug calls at statement marker points.

20.2.2 BINDing with Debug

When you bind your program with BINDER, you must use the DEBUG statement to direct BINDER to use the debug calls generated by SPRITE (they are otherwise deleted). You must also tell BINDER which files it should use for the following ICMs:

| | |
|---|---|
| arm | |
| dbwrite | (only if you use it) |
| debug | |
| err | (the debug version!) |
| print | |
| put | |
| readcd | |
| trace | |

We suggest that you use the PRINT LAYOUT statement in your bind deck to get the physical memory layout of your program, since some kinds of errors give you only the segment number and instruction address of the error.

As the debug package is quite large, you may wish to take some of the following steps to help keep your program smaller or to keep code under 300 kd.

1. Make debug.initialize a pass entry point. This will save you 12 to 15 kd if your program has multiple passes. You could also make it a segment if you use only one pass but have several segments.

20.2.2 BINDing with Debug (Continued)

2. Put the db_debug_proc_table data block into a HIGH DATA statement. This will help you avoid code over 300 kd. If you are using the statistics version of debug, you may also put the st_statistics_info data block into high memory.

## 20.2.3 Executing With Debug

We now explain - in gruesome detail - the ways that you might use the debug package to help you find your program errors.

Debug allows you to perform one or more of the following types of debugging when your program executes.

1. Monitor the flow of control in your program, showing the entry into and exit from procedures.

2. Monitor the flow of control within procedures in your program, showing your program pass from statement to statement.

3. Print the output from the dbwrite module, showing values of data from statements which you have coded into your modules.

4. Terminate your program at a specified point to limit execution or control runaway loops.

5. Trace your program.

6. Obtain memory dumps of your program at a specified point.

7.* Control the execution of your program interactively from your terminal, rather than from the normal batch processing mode.

In the following sections we show how you may use debug to accomplish the actions described above.

### 20.2.3.1 Primary Debug Input

You control the debugging actions that debug will take when your program runs by means of the **second** '/' execute parameter which you give to your program when you execute it.

20.2.2 BINDing with Debug (Continued)

2. Put the db_debug_proc_table data block into a HIGH DATA statement. This will help you avoid code over 300 kd. If you are using the statistics version of debug, you may also put the st_statistics_info data block into high memory.

## 20.2.3 Executing With Debug

We now explain - in gruesome detail - the ways that you might use the debug package to help you find your program errors.

Debug allows you to perform one or more of the following types of debugging when your program executes.

1. Monitor the flow of control in your program, showing the entry into and exit from procedures.

2. Monitor the flow of control within procedures in your program, showing your program pass from statement to statement.

3. Print the output from the dbwrite module, showing values of data from statements which you have coded into your modules.

4. Terminate your program at a specified point to limit execution or control runaway loops.

5. Trace your program.

6. Obtain memory dumps of your program at a specified point.

7.* Control the execution of your program interactively from your terminal, rather than from the normal batch processing mode.

In the following sections we show how you may use debug to accomplish the actions described above.

### 20.2.3.1 Primary Debug Input

You control the debugging actions that debug will take when your program runs by means of the <u>second</u> '/' execute parameter which you give to your program when you execute it.

20.2.3.1 Primary Debug Input (Continued)

You may use any combination of the following debug command letters in the second '/' parameter to your program:

/A monitors All procedures and statements.

/C extended input is in Capital letters.

/D memory Dump taken on error termination.

/H change number of monitor History entries printed on error termination.

/I* Interacts with debugger.

/M Monitors all procedures.

/N output on Narrow (SHORT) paper.

/O monitors Overlay calls.

/P Prints trace listings on Printer.

/S prints procedure Statistics when your program terminates.

/T Traces all procedures - not recommended!

/U output printed in Uppercase letters.

/W all dbWrite lines printed.

/X reads eXtended debug commands.

Debug uses the second '/' parameter to avoid conflicts with programs you may have which use the first '/' parameter.

Note: To use the second '/' parameter, you must provide something as the first '/' parameter when you execute your program, even if your program doesn't use it.

Example: EXECUTE TSPROG / DUMMY / X.

** /C Debug Command

This command allows you to create extended debug command decks ("/X") on a keypunch that does not have lower-case letters. In this deck, you must precede any 'real' upper-case words by an underscore (for example, '_MONITOR'). When processing any other words, debug will

** /C Debug Command (Continued)

convert all upper-case letters to lower-case letters before it uses them (these are usually procedure names).

** /H Debug Command

You may use the '/H' command plus a following number to specify the number of history monitor entries which debug prints if your program terminates abnormally. This history shows you the final 'n' procedure calls and/or returns that your program made. Debug defaults to ten history entries.

If you wished to see 39 monitor lines from the history, you would execute your program with '/H39' as one of the debug commands.

** /I Debug Command

This command allows you to control your program interactively from your terminal. More details are TBS.

** /O Debug Command

You may use this command to get information about the overlay calls which your program makes. When debug finds that an overlay call has been made (or is about to be made), it prints two lines giving you the names of the procedures that your program is coming from and going to. With this information, you may revise your BIND deck to improve your program's performance.

** /P Debug Command

This command allows you to break each part of a trace into a separate listing which goes directly to the printer. You must also use the '/T' command, '/X' and extended debug commands, or program action to cause debug to trace the desired procedures.

You might use this option to stop and start your program at will by making the printer not ready or ready.

You might also use this option to make your program execute faster, since the MCP does not trace debug. Without this option, the MCP does trace debug, even though you never see it on your trace output.

Since the trace listing goes directly to the printer, you may not use RCSPBO to examine the trace on your terminal, as you could without this command.

** /C Debug Command (Continued)

convert all upper-case letters to lower-case letters before it uses them (these are usually procedure names).

** /H Debug Command

You may use the '/H' command plus a following number to specify the number of history monitor entries which debug prints if your program terminates abnormally. This history shows you the final 'n' procedure calls and/or returns that your program made. Debug defaults to ten history entries.

If you wished to see 39 monitor lines from the history, you would execute your program with '/H39' as one of the debug commands.

** /I Debug Command

This command allows you to control your program interactively from your terminal. More details are TBS.

** /O Debug Command

You may use this command to get information about the overlay calls which your program makes. When debug finds that an overlay call has been made (or is about to be made), it prints two lines giving you the names of the procedures that your program is coming from and going to. With this information, you may revise your BIND deck to improve your program's performance.

** /P Debug Command

This command allows you to break each part of a trace into a separate listing which goes directly to the printer. You must also use the '/T' command, '/X' and extended debug commands, or program action to cause debug to trace the desired procedures.

You might use this option to stop and start your program at will by making the printer not ready or ready.

You might also use this option to make your program execute faster, since the MCP does not trace debug. Without this option, the MCP <u>does</u> trace debug, even though you never see it on your trace output.

Since the trace listing goes directly to the printer, you may not use RCSPBD to examine the trace on your terminal, as you could without this command.

** /S Debug Command

If you use the '/S' command, debug will print a list of all procedures which your program called, along with a count of the number of times each was called.

If you are using the statistics version of debug, debug will print a report giving the approximate time that each of your program's procedures used during execution (<u>NOTE</u>: for best results, do this with no other jobs running). This may give you some clues as to how you might improve your program's performance.

** /T Debug Command

While we hate to see trace as a debugging tool, we recognize its need in exceptional circumstances. However, you should think three or four times before using the '/T' option. In most cases, you can find your problem by tracing procedures selectively with the "/X" command and extended debug commands.

** /U Debug Command

You will find this command useful if your debug output must go to a printer which cannot print lower-case letters.

20.2.3.2 Extended Debug Input (/X)

If you have used the '/X' option, you must put extended debug commands into a card file (or editor file, if you use SYS COMP) for debug to read and process.

The syntax of the extended debug commands is:

```
extended debug commands
|
|      ---------------------------
|    /                            \
\___\___ debug specification ___/___ END ___
                                             \
                                              |
                                              |
```

20.2.3.2 Extended Debug Input (/X) (Continued)

```
                  debug specification
                  |
                  |
                  |    ---------------------
                  |   /                     \       ___ procedure list ___
                  \___\___ debug action ___/___/                          \___
                  |                                \___ ALL ___ counts ___/    \
                  |                                                             |
                  |                                                             |
                  |                   ----------------------                   |
                  |                  /                      \                  |
                  \___ DONT_DEBUG ___\___ procedure name ___/_____________     |
                  |                                                        \|
                  |                                                          |
                  |           ___ limit procedure list ______                |
                  \___ LIMIT ___/                               \________    |
                                \___ ALL ___ <limit: number> ___/         \|
                                                                            |
                                                                            |
                                                                            |


                  debug action
                  |
                  |
                  \___ DBWRITE _______
                  |                    \
                  \___ MONITOR ______   |
                  |                   \|
                  \___ STATEMENTS ___  |
                  |                   \|
                  \___ TRACE ________  |
                                      \|
                                       |
                                       |


              procedure list
              |
              |    -------------------------------------------------------
              |   /                                                       \
              |   |                                                       |
              \__\__ procedure name _______________________________________/__
                                     \              /  \               /     \
                                      \__options__/    \__counts__/          |
                                                                              |
                                                                              |
                                                                              |
```

20.2.3.2 Extended Debug Input (/X) (Continued)

```
counts
|
|
\___ <start count: number> ______________
                              \           \
 ______________________________/           |
/                                          |
|           ___ <end count: number> ___    |
\___ - ___/                           \__  |
          \___ END ___________________/  \ |
                                           |
                                           |
                                           |


procedure name
|
|
\___ <module name: string>.<proc name: string> ___
                                                  \
                                                  |
                                                  |


limit procedure list
|
|    ____________________________________________
|   /                                            \
\___\___ procedure name ___ <limit: number> ___/___
                                                   \
                                                   |
                                                   |


options
|
|    _________________________________________________
|   /                                                 \
|   |                                                 |
|   |     ___ LEVEL ___ <levels to debug: number> ___  |
\__\___/                                           \___/__
       \___ RELATIVE ______________________________/      \
                                                          |
                                                          |
                                                          |
```

You may provide only one debug specification for a particular procedure. However, you may use any number of debug specifications for 'ALL'. For example:

20.2.3.2 Extenced Debuo Inout (/X) (Continued)

MONITOR ALL 309 - END TRACE ALL 786

** MONITOR Debuo Command

The MONITOR command allows you to see your prooram call procedures ('NTR') and return from procedures ('EXT'). Each MONITOR line which debuo prints contains the ALL count, 'NTR' or 'EXT', the name of the module and procedure, and the procedure relative count (which is the number of times that this procedure has been called).

'MONITOR ALL' is equivalent to '/M'.

See Debuo Action Controls, below, for additional information about this command.

** STATEMENTS Debuo Command

The STATEMENTS command allows you to see your prooram execute within a procedure on a statement-by-statement basis. You will see which way an IF statement branched, what CASE statement code was executed, etc. This should give you some idea about the values of control variables in your program.

Each STATEMENTS line which debuo prints contains the ALL count, 'STMT', the record number of the statement, the line number of the source line containing the statement, and the procedure relative statement number (which is the number of statements executed so far in the current invocation of this procedure).

The STATEMENTS command also sets MONITOR, so that you will have 'NTR' and 'EXT' MONITOR lines to show you what procedure is executino.

"STATEMENTS ALL" is equivalent to "/A".

See Debuo Action Controls, below, for additional information about this command.

** DBWRITE Debug Command

The DBWRITE command enables the printing of debugging information lines by the dbwrite module. This allows you to see the values of variables as your program executes.

'DBWRITE ALL' is equivalent to '/W'.

** DBWRITE Debug Command (Continued)

See Debug Action Controls, below, for additional information about this command.

** TRACE Debug Command

The TRACE command allows you to trace the execution of your program. We hate to see trace used as a debugging tool, but we recognize the need in some cases to see the values of variables. We hope one day to have a high-level monitoring capability for variables.

You have a choice in the way you trace. By default, all of your tracing will go on a single printer backup listing. While this requires more overhead during execution (the MCP traces debug, but does not print those lines), it gives you a single listing which you may examine from your terminal with RCSPBD.

If you use the /P Debug Command (see section 20.2.3.1), each piece of your trace will go directly to a printer. This will be faster, and it gives you a means of directly controlling your program (by stopping the printer), but it may not be as convenient.

At the beginning of each section of trace, you will see a trace line containing "COUNT =", the ALL count, "/", the procedure relative count, ':', 'NTR' or 'EXT' or 'STM', the name of your procedure (in upper-case letters), and a marker of "<*<*<*<*<*<*<".

'TRACE ALL' is equivalent to '/T'.

See Debug Action Controls, below, for additional information about this command.

** DONT_DEBUG Debug Command

The DONT_DEBUG command directs debug to ignore the specified procedures, even if they would normally be used in some way (as in MONITOR ALL or /W).

You may use this to suppress debug information on procedures which you feel are reliable or unimportant, thereby reducing the amount of output from debug.

** LIMIT Debug Command

The LIMIT command directs debug to terminate execution of your program when any of the specified limits are exceeded.

You may use this command with the '/D' command to get a memory dump at a specific place in your program.

** Debug Action Controls

You may limit the scope or range of action of the MONITOR, STATEMENTS, DBWRITE and TRACE debug commands. You may wish a debug action to take place

1. within a specified range of ALL counts, or

2. when a particular procedure is called within a specified range of ALL counts, or

3. on the 'nth' through 'mth' calls of a particular procedure.

The ALL count gives the number of times that debug has found an 'NTR', 'EXT', or 'STM' during the execution of your procedure.

When you specify 'ALL' and (optionally) a range, your actions apply to that range (which defaults to 1 - END) of ALL counts.

When you specify a procedure and (optionally) a range, your actions apply to that procedure, but only if it is called during that range of ALL counts.

When you specify a procedure followed by RELATIVE and (optionally) a range, your actions apply to the 'range-start' through 'range-end' calls to that procedure. For example,

```
MONITOR your_mod.get_name RELATIVE 37
```

monitors <u>only</u> the 37th call of that procedure.

LEVEL determines how your debugging action will be applied to procedures which your specified procedure calls. By default, your debugging action applies to all called procedures, to all procedures which they call, etc.

You may limit the number of descendants to which your debugging applies by using the LEVEL clause. The level number you specify limits your debugging action to that

** Debug Action Controls (Continued)

number of descendants. For example, given the following flow of execution,

```
a ==> b                       (proc 'a' calls proc 'b')
      b ==> c
      b <== c                 (proc 'c' returns)
      b ==> d
            d ==> e
            d <== e
      b <== d
a <== b
```

the debug command

```
MONITOR a
```

will show monitor lines for procedures a, b, c, d, and e. The debug command

```
MONITOR a LEVEL 2
```

will show monitor lines for a, b, c, and d,

```
MONITOR a LEVEL 1
```

will show monitor lines for a and b, and

```
MONITOR a LEVEL 0
```

will show monitor lines only for procedure a.

20.2.3.3 * Interactive Debug Input

If you have used the '/I' option, you will provide interactive debug commands to a program in the time-sharing area which is connected to your terminal for input and output. This program will pass information to the debug package in your program and get information back from your program, which may then be displayed on your terminal.

The syntax of the interactive debug commands is TBS.

20.2.4 <u>Dbwrite Module</u>

You may use the dbwrite module to see the values of variables during the execution of your program. You may put calls to these procedures in your modules where you wish. These allow you to print labeled numbers, strings, booleans, and internal (hex) representations of anything -

20.2.4 Dbwrite Module (Continued)

structures, symbolics, etc.

You have complete control during the execution of your program as to whether these values are printed or not. Your use of the '/W' debug command or the DBWRITE statement determines if dbwrite will print anything, and when.

You can see that the main disadvantage to this module is that you lack flexibility, and that you must recompile your module to make changes.

20.2.5 Arm Module

You may use the arm module to let your program handle its own errors without your program being DS-ed. When you arm your program, you specify by a parameter the procedure that you wish to be called upon encountering a processor error or program trap. During the process of handling the error, you may, if you wish, call a routine in the arm module to format and (optionally) display the exact cause of the error.

The arm module stores much information about the error in the 'arm_parameters' data block. Your program may access this information as needed.

You would normally use the arm module to provide a graceful termination for your program (close files, write error messages, etc.). However, you could in principle recover from the error condition and resume normal execution of your program, but you would encounter great difficulties.

The arm module processes only the first call on the arming procedure. Others are ignored. When you are using the debug package, the debug module makes the first call, so your program's call gets ignored.

However, after your program has an error, you may once again arm your program. Note also that you may change your error handler routine if you wish, since arm keeps it in a PTR TO PROC variable in the 'arm_parameters' data block.

WARNING: If you use 'prog.arm' in your modules, you may interfere with the correct operation of the arm and debug modules.

NOTE: you may save almost 10kd in your program by putting 'arm.get_error_message' and its associated data 'arm.error_message_tables' in an overlay.

## 20.2.6 Program Interaction With Debug

You may write your program in such a way that it controls debugging actions internally. You accomplish this by means of the 'dbstatus' data block. Your program may inspect this block to determine if debug is monitoring, dbwriting, statement monitoring, or tracing. Your program may set flags in this block which cause debug to begin (or stop) monitoring, dbwriting, statement monitoring, or tracing.

As you can see, we have given you a lot of power for implementing your own debugging methods within your program.

## Appendix A SPRITE STANDARD OPERATIONS

### FUNCTIONS

#### abs

FUNCTION: to return the absolute value of either a REAL or an INTEGER.

RETURN TYPE: REAL or INTEGER (depending on the absolute value desired).

PARAMETER:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | INTEGER or REAL | CONST | number whose absolute value is sought |

#### edit_number

FUNCTION: to return a string which is the numeric value of the 1st parameter as formatted by the picture specified by the second parameter. See Appendix B (PICTURES).

RETURN TYPE: STRING(n) OF CHAR, where n is uniquely determined from the picture parameter.

PARAMETER:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | integer | CONST | numeric value to be edited |
| 2 | string | CONST | string which describes a picture (See Appendix B) |

Appendix A SPRITE STANDARD OPERATIONS (Continued)

fill_array

FUNCTION: to set all elements of a one dimensional array to the specified value.

RETURN TYPE: ARRAY [<range>] OF <elem type> (determined from destination array context)

PARAMETER:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | Any | CONST | value to be put into destination array elements |

fill_string

FUNCTION: to set a string to the replication of a specified string value.

RETURN TYPE: STRING (n) OF <string type> (determined from destination string context)

PARAMETER:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | string | CONST | the string to be replicated throughout the destination string |

index

FUNCTION: returns the (position) index of the first occurrence of the 1st string parameter in the 2nd string parameter.
It returns 0 if there are no occurrences.

RETURN TYPE: 0 .. string_2_length

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | string | CONST | substring sought |
| 2 | string | CONST | string searched for 1st parameter |

Appendix A SPRITE STANDARD OPERATIONS (Continued)

index_any

FUNCTION: returns the (position) index of the first occurrence of any elements from the 1st string parameter that is in the 2nd string parameter. It returns 0 if there are no occurrences.

RETURN TYPE: 0 .. string_2_length

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | string | CONST | string whose elements are sought |
| 2 | string | CONST | string searched for elements of 1st parameter |

index_inc

FUNCTION: returns the (position) index of the first occurrence of the 1st string parameter in the 2nd string parameter where the occurrence begins on a multiple of the 3rd numeric parameter (1, n+1, 2n+1, etc.) It returns 0 if there are no occurrences.

RETURN TYPE: 0 .. string_2_length

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | string | CONST | substring sought |
| 2 | string | CONST | string to be searched |
| 3 | 1..100 | CONST | increment between comparisons |

Appendix A SPRITE STANDARD OPERATIONS (Continued)

index_none

FUNCTION: returns the (position) index of the first occurrence of an element that is in the 2nd string parameter and not in the 1st string parameter. It returns 0 if every element of the 2nd string parameter is also an element of the 1st string parameter.

RETURN TYPE: 0 .. string_2_length

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | string | CONST | string whose elements are sought |
| 2 | string | CONST | string searched for |

length

FUNCTION: returns the length of a string.

RETURN TYPE: INTEGER.

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | string | CONST | the string whose length is the result |

lwb

FUNCTION: returns the lower bound of an array index.

RETURN TYPE: a scalar of the same type as the array index.

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | ARRAY | CONST | the array whose subscript is to be examined |

Appendix A SPRITE STANDARD OPERATIONS (Continued)

| # | Type | Access | Description |
|---|---|---|---|
| 2 | INTEGER | CONST | the subscript which is to be examined |

proc_ptr

FUNCTION: returns a proc pointer to the procedure argument.

RETURN TYPE: PTR TO PROC type.

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | **Any proc | CONST | procedure name or module name. procedure name |

** Having same parameter list, etc.

ptr

FUNCTION: returns a pointer to the argument.

RETURN TYPE: PTR TO <argument access> <argument type>

The access and type is determined by the access and type of the argument.

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | Any | CONST | declared data not an expression |

round (*)

FUNCTION: returns the closest integer of a REAL value.

RETURN TYPE: INTEGER

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | REAL | CONST | the expression whose closest integer is the result |

Appendix A SPRITE STANDARD OPERATIONS (Continued)

translate

FUNCTION: returns a string which is the value of the first string parameter translated with the translate table specified by the second parameter.

RETURN TYPE: STRING (string_1_length) OF EBCDIC

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | string | CONST | string to be translated |
| 2 | TRANSLATE_TABLE | CONST | Translate table to be used |

upb

FUNCTION : returns the upper bound of an array index.

RETURN TYPE: a scalar of the same type as the array index.

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | ARRAY | CONST | the array whose subscript is to be examined |
| 2 | INTEGER | CONST | the subscript which is to be examined |

Appendix A SPRITE STANDARD OPERATIONS (Continued)

## zone_index_any

FUNCTION: returns the (position) index of the first occurrence of any element from the 1st string having a zone digit that also occurs in the 2nd string. It returns 0 if there are no occurrences.

RETURN TYPE: 0 .. string_2_length

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | string | CONST | EBCDIC string whose zone digits are sought |
| 2 | string | CONST | EBCDIC string searched |

## zone_index_none

FUNCTION: returns the (position) index of the first occurrence of any element from the 1st string with a zone digit unlike any in the 2nd string. It returns 0 if every zone digit of the 1st string is also a zone digit of the 2nd string.

RETURN TYPE: 0 .. string_2_length

PARAMETERS:

| # | Type | Access | Description |
|---|---|---|---|
| 1 | string | CONST | EBCDIC string whose zone digits are sought |
| 2 | string | CONST | EBCDIC string searched |

Appendix A SPRITE STANDARD OPERATIONS (Continued)

MODULES

The following module descriptions describe the procedures and parameters for the standard modules "io" and "prog". Their use in any module requires no MID description modification, as they are predefined.

Any indicants used as parameter types should be understood as representing generic types similar to their names. For example: FILE = any file, RECORD = any legitimate record type for the associated file.

The names of the procedures describe their function very closely.

```
prog
MOD

    day
    PROC RETURNS STRING (9);                % day of week

    date
    PROC RETURNS STRING (6) OF HEX;         % MMDDYY form

    julian_date
    PROC RETURNS STRING (5) OF HEX;         % YYDDD form

    time
    PROC RETURNS STRING (10) OF HEX;        % OOHHMMSSss form

    millisec
    PROC RETURNS 0.. 9999999999;            % 0099999999 form

    arm
    PROC;                                   % arm the processor

    dump
    PROC;                                   % Dump to disk

    start_trace
    PROC (trace_to_disk BOOLEAN)            % CONSTANT!

    stop_trace
    PROC;
```

```
Appendix A     SPRITE STANDARD OPERATIONS (Continued)

               stop_run
               PROC (with_spo_err_msg BOOLEAN);    % CONSTANT!

               wait
               PROC (seconds 1.. 86399);

               bct
               PROC (bct_number  0..9999,
                     bct_param  UNIV PARAM_STRING_2_TO_400_HEX);

               change_memory_size
               PROC (size_in_kd  -999..999);   % (+) gets more,
                                               % (-) returns same

               complex_wait
               PROC ([<event_list>], use_expr 1..99)
                     RETURNS 1..99;      %See complex_wait function below.

               move_words
               PROC;
                                          %See move_words function below.

           DOM;
```

Appendix A    SPRITE STANDARD OPERATIONS (Continued)

```
        io
        MOD

            accept
            PROC (message VAR PARAM_STRING_1_TO_60);

            display
            PROC (message     PARAM_STRING_1_TO_60);

            display_lines
            PROC (message     PARAM_STRING_1_TO_73000,    % SPECIAL!
                  number_of_lines  1..999);          % A Null Char (hex 00)
                                                     % must end each line
                                                     % which is shorter than
                                                     % 72 characters.

            %  open and close procedures

            open
            PROC (file   FILE);            % file.MYUSE gives open type

            close
            PROC (file   FILE);

            close_lock
            PROC (file   FILE);

            close_release
            PROC (file   FILE);

            close_purge
            PROC (file   FILE);

            close_remove
            PROC (file   FILE);

            close_crunch
            PROC (file   FILE);

            close_remove_crunch
            PROC (file   FILE);

            close_no_rewind
            PROC (file   FILE);


            close_release_no_rewind
            PROC (file   FILE);
```

Appendix A SPRITE STANDARD OPERATIONS (Continued)

```
% I/O procedures

read                                      % sequential
PROC (file        FILE,
      record VAR RECORD);

write                                     % sequential
PROC (file     FILE,
      record   RECORD);

read_datacomm
PROC (file        FILE,
      record VAR RECORD);

write_datacomm
PROC (file        FILE,
      record      RECORD);

read_random                               % random disk/pack only
PROC (file        FILE,
      record VAR RECORD,
      key         1..99999999);

write_random                              % random disk/pack only
PROC (file          FILE,
      record        RECORD
      key           1..99999999);

print                                     % Printer only
PROC (file          FILE,
      record        RECORD,
      skip_lines    0..2);                % overprint, single, double

punch                                     % Card punch only
PROC (file          FILE,
      record        RECORD
      stacker       0..2);

% file positioning procedures

skip                                      % non_printer files
PROC (file              FILE,
      records_to_skip   -9999..9999);     % <0 -> backwards

position                                  % printer only
PROC (file               FILE,
      skip_to_channel    0..11,           % 0 = ignored 1 = T.O.P.
      lines_to_skip      0..99);          % valid if skip = 0
```

Appendix A    SPRITE STANDARD OPERATIONS (Continued)

```
prepare_user_defined_buffer_io
PROC (file                FILE,
      buffer              UNIV PARAMETRIC_HEX_STRING);
                                % Must be called before any of the
                                % four following procedures may
                                % be used. If called, the four
                                % followimg procedures must be
                                % used in place of read, write,
                                % read_random and write_random

read_buffer
PROC (file                FILE,
      buffer              UNIV PARAMETRIC_HEX_STRING);
                                                   % Modulo and
                                                   % size must be
                                                   % mod 4. Buffer
                                                   % must be same
                                                   % as used in
                                                   % user_defined_
                                                   % buffer_io

read_random_buffer
PROC (file                FILE,
      buffer              UNIV PARAMETRIC_HEX_STRING,
      key                 1..99999999);            % Modulo and
                                                   % size must be
                                                   % mod 4. Buffer
                                                   % must be same
                                                   % as used in
                                                   % user_defined_
                                                   % buffer_io

write_buffer
PROC (file                FILE,
      buffer              UNIV PARAMETRIC_HEX_STRING);
                                                   % Modulo and
                                                   % size must be
                                                   % mod 4. Buffer
                                                   % must be same
                                                   % as used in
                                                   % user_defined_
                                                   % buffer_io
```

Appendix A    SPRITE STANDARD OPERATIONS (Continued)

```
write_random_buffer
PROC (file            FILE,
      buffer          UNIV PARAMETRIC_HEX_STRING,
      key             1..99999999);           % Modulo and
                                              % size must be
                                              % mod 4. Buffer
                                              % must be same
                                              % as used in
                                              % user_defined_
                                              % buffer_io

DOM;
```

port_io procedures are listed in section 18.4.

## A.1 Complex_Wait Function

The syntax for the standard function complex_wait is:

```
<give_variable> := prog.complex_wait
                                  ([<event_list>],
                                   <use_expr>);
```

A legal event within the event_list is:

a. numeric expression between 0 and 86400
b. ODTINPUTPRESENT
c. <file_type>.<event_type>

where file_type is one of the following:
file,
port,
subport

and where event_type is one of the following:
OUTPUTEVENT,
INPUTEVENT,
CHANGEEVENT,
READYEVENT

A.1 Complex_Wait Function (Continued)

d. primary.<event_type>

where primary is either a program name or a STOQUE que name of type STRING(6)

and where <event_type> is one of the following:
CRCROUTPUTEVENT,
CRCRINPUTEVENT,
STOQOUTPUTEVENT,
STOQINPUTEVENT

The following table shows the legal combinations:

| | OUTPUTEVENT | INPUTEVENT | CHANGEEVENT | READYEVENT |
|---|---|---|---|---|
| file | false | true | false | false |
| port | false | true | true | true |
| subport | true | true | true | true |

Examples:

Legal:

file.INPUTEVENT, ODTINPUTPRESENT, port.READYEVENT

Illegal:

port.OUTPUTEVENT, file.READYEVENT

The order in which the events are tested is determined by <use_expr>. If <use_expr> = 1, the events are tested in the order they are specified in the list; otherwise the first event tested is the one which occupies the nth position in the list, where n is the value of <use_expr>.

A.2 Move Words Procedure

This standard procedure, **which should be used with extreme caution,** allows you to force the compiler to generate MVW code in circumstances in which it would not normally do so.

This procedure takes two UNIV parameters: the source field and the destination field. **No compile-time or run-time checks are made to see if these two fields are on MOD 4 addresses, have MOD 4 sizes and have the same size.**

**It is YOUR responsibility to insure that the MVW will function correctly when your program runs!** The SPRITE group will react with displeasure if you report "bugs" which turn

A.2 Move Words Procedure (Continued)

out to be caused by misuse of this standard procedure.

For example,

```
        move_words (source_field, destination_field);
```

Most people thinking of using the move_words standard function will have no need of it. SPRITE optimizes to MVW whenever it can guarantee at compile time that it will work. As a guide to those who are interested, the exact conditions under which SPRITE makes this optimization are spelled out below.

Both operands must have the same size and controller. The size and address of both operands must be MOD 4. (This includes a MOD 4 offset from the beginning of a data block, for example.) Both operands must be fixed length. Unless an operand's type is MOD 4, it cannot use indexing (except IX3, which is always MOD 4) or indirection. Furthermore, if indirection is involved, the final controller must be UN.

## Appendix B PICTURES

A picture is a constant character string which specifies the editing operations to be performed by the edit_number function. SPRITE pictures are similar in construction and operation to COBOL pictures.

Pictures are constructed from the special characters + - $ Z B I . Parentheses and period also have special meanings in pictures. The order among the characters + - $ Z 9 is significant and determines the legality of a picture.

The following special characters cause the specified action when they appear in a picture.

| | |
|---|---|
| 9 | Move a digit from the numeric source to the string result, converting it to a numeric character. |
| Z | Like 9, except replace leading zeroes in the source with blanks in the result. |
| $ | Insert a dollar sign; may float. |
| + | Insert a plus sign if the value is non-negative, or a minus sign if it is negative; may float. |
| - | Like +, but insert a blank if non-negative; may float. |
| B | Insert a blank. |
| Ix | Insert the character following the I (used to insert special characters + - $ 9 Z I B .). |
| . | Period is reserved for future use and may not be used directly. (Use I. to insert a period). |
| | Other characters simply insert the specified character. |

As a shorthand notation, any character (or the Ix pair) may be followed by a count in parentheses which specifies how many times the character is to be repeated, up to a maximum of 99. For example: '9(5)/9(5)' is the same as '99999/99999'.

As noted in the table, the special characters + - $ may float. This means that if two or more of the same characters are contiguous, that character will appear immediately to the left of the first non-zero character.

Appendix B PICTURES (Continued)

All previous character positions specified by the floating character will be blank. Thus, a floating character combines the zero suppression feature of Z with a simple insert. For example: value = -00023, picture = "+++++9", result = "    -23".

The order requirement among the special character + - $ Z 9 is as follows:

a) + or - must precede any $, Z, 9

b) + and - may not be mixed in the same picture

c) $ must follow +, - and precede Z, 9

d) Z must follow +, -, $ and precede 9

e) 9 must follow +, -, $, Z

Insertion characters may appear anywhere in the picture with no restriction. However, if they appear in a floating or Z field (i.e., ++,+++,++9 or ZZ/ZZ/Z9) they <u>will not be inserted</u> until the first non-zero character has been transferred to the result. For example: value = 000999, picture = "ZZ/ZZ/Z9" gives "    9/99" rather than "  /  9/99".

Examples:

```
9999                ZZZZ              ZZZZ9

+++ZZ9              Z9/99/99          Z9:99

+Z,ZZZ,ZZZI.99                        +$$$9

ZZZ9I.99B%
```

Appendix B PICTURES (Continued)

All previous character positions specified by the floating character will be blank. Thus, a floating character combines the zero suppression feature of Z with a simple insert. For example: value = -00023, picture = "+++++9", result = "    -23".

The order requirement among the special character + - $ Z 9 is as follows:

a) + or - must precede any $, Z, 9

b) + and - may not be mixed in the same picture

c) $ must follow +, - and precede Z, 9

d) Z must follow +, -, $ and precede 9

e) 9 must follow +, -, $, Z

Insertion characters may appear anywhere in the picture with no restriction. However, if they appear in a floating or Z field (i.e., ++,+++,++9 or ZZ/ZZ/Z9) they <u>will not be inserted</u> until the first non-zero character has been transferred to the result. For example: value = 000999, picture = "ZZ/ZZ/Z9" gives "    9/99" rather than "  / 9/99".

Examples:

```
9999                ZZZZ            ZZZZ9

+++ZZ9              Z9/99/99        Z9:99

+Z,ZZZ,ZZZI.99                      +$$$9

ZZZ9I.99B%
```

## Appendix C FILE ATTRIBUTES

As File Attributes are defined by the CSG FILE HANDLING Standard (1955 2926), they are not an integral part of the SPRITE language; however, as they do appear in the source text of a SPRITE program, the File Attributes supported by SPRITE are included here for documentation and reference.

Supported File Attributes

The format of each entry in the table describing the file attributes is:

ATTRIBUTE NAME

Applicable peripherals; Read/Write Status
TYPE, including mnemonic names
Default, if any

Brief description

The read/write status indicates when the attribute can be read or written.

The description section also notes non-standard attributes and non-standard interpretations of otherwise standard attributes. See the File Handling Standard (FHS) for more detail where necessary.

A read/write status of assigned means that a physical file must be associated with the opened logical file.

The supported file attributes are as follows:

ACCESSMODE

Disk/Diskpack; Read: anytime, Write: closed
Mnemonic: SEQUENTIAL, RANDOM
Default: SEQUENTIAL

Specifies the disk access technique. Non-standard.

## Appendix C FILE ATTRIBUTES

As File Attributes are defined by the CSG FILE HANDLING Standard (1955 2926), they are not an integral part of the SPRITE language; however, as they do appear in the source text of a SPRITE program, the File Attributes supported by SPRITE are included here for documentation and reference.

### Supported File Attributes

The format of each entry in the table describing the file attributes is:

ATTRIBUTE NAME

Applicable peripherals; Read/Write Status
TYPE, including mnemonic names
Default, if any

Brief description

The read/write status indicates when the attribute can be read or written.

The description section also notes non-standard attributes and non-standard interpretations of otherwise standard attributes. See the File Handling Standard (FHS) for more detail where necessary.

A read/write status of assigned means that a physical file must be associated with the opened logical file.

The supported file attributes are as follows:

ACCESSMODE

Disk/Diskpack; Read: anytime, Write: closed
Mnemonic: SEQUENTIAL, RANDOM
Default: SEQUENTIAL

Specifies the disk access technique. Non-standard.

Appendix C FILE ATTRIBUTES (Continued)

AREAS

DISK/DISKPACK; Read: anytime, Write: closed
Integer: 0..99, 0 means 100
Default: 20

Specifies the maximum number of areas the file may occupy.

NOTE: Ignored when opening a permanent file.

AREASIZE

DISK/DISKPACK; Read: anytime, Write: closed
Integer: 1..9999 999
Default: 1000 (approximately)

Specifies the number of records in an area.

NOTE: Must be evenly divisible by the number of records in a block. At declaration time, the default or actual value is adjusted to meet this condition. Non-standard (see FHS - AREALENGTH).

AUTOPRINT

PRINTER; Read: anytime, Write: anytime
BOOLEAN
Default: FALSE

Causes files that have been spooled to an intermediate peripheral to be printed automatically at EOJ. Non-standard.

BACKUPKIND

PRINTER/PUNCH(*); Read: anytime(*), Write: closed
Mnemonic: DISK, DISKPACK, DONTCARE(*), TAPE
Default: DONTCARE

Specifies the intermediate peripheral to which the logical file will be spooled. DONTCARE creates the file on the system's default backup media.

NOTE: Should only be used when BACKUPPERMITTED = MUSTBACKUP. Non-standard as this

Appendix C FILE ATTRIBUTES (Continued)

AREAS

DISK/DISKPACK; Read: anytime, Write: closed
Integer: 0..99, 0 means 100
Default: 20

Specifies the maximum number of areas the file may occupy.

NOTE: Ignored when opening a permanent file.

AREASIZE

DISK/DISKPACK; Read: anytime, Write: closed
Integer: 1..9999,999
Default: 1000 (approximately)

Specifies the number of records in an area.

NOTE: Must be evenly divisible by the number of records in a block. At declaration time, the default or actual value is adjusted to meet this condition. Non-standard (see FHS - AREALENGTH).

AUTOPRINT

PRINTER; Read: anytime, Write: anytime
BOOLEAN
Default: FALSE

Causes files that have been spooled to an intermediate peripheral to be printed automatically at EOJ. Non-standard.

BACKUPKIND

PRINTER/PUNCH(*); Read: anytime(*), Write: closed
Mnemonic: DISK, DISKPACK, DONTCARE(*), TAPE
Default: DONTCARE

Specifies the intermediate peripheral to which the logical file will be spooled. DONTCARE creates the file on the system's default backup media.

NOTE: Should only be used when BACKUPPERMITTED = MUSTBACKUP. Non-standard as this

Appendix C FILE ATTRIBUTES (Continued)

attribute modifies KIND.

BACKUPPERMITTED

PRINTER/PUNCH(*); Read: anytime, Write: closed
Mnemonic: DONTBACKUP, DONTCARE, MUSTBACKUP
Default: DONTCARE

Specifies whether an intermediate peripheral may be associated with the logical file.

BLOCK

General; Read: anytime, Write: never
Integer: 0..99999999

Returns the number of the logical block referenced in the last I/O statement.

BLOCKSIZE

General; Read: anytime, Write: closed
Integer: 1..999999
Default: size of one record

Specifies the length of a block in digits. Should be a multiple of the size of a record (MAXRECSIZE).

BLOCKSTRUCTURE

General; Read: anytime, Write: closed
Mnemonic: FIXED, VARIABLE(*)
Default: FIXED

Specifies the format of the records in a block.

BUFFERS

General; Read: anytime, Write: closed
Integer: 0..9, 0 is special
Default: 1

Specifies the number of buffers assigned to the file. Non-standard interpretation of zero, indicating that the user will supply his own buffer.

Appendix C FILE ATTRIBUTES (Continued)

attribute modifies KIND.

BACKUPPERMITTED

PRINTER/PUNCH(*); Read: anytime, Write: closed
Mnemonic: DONTBACKUP, DONTCARE, MUSTBACKUP
Default: DONTCARE

Specifies whether an intermediate peripheral may be associated with the logical file.

BLOCK

General; Read: anytime, Write: never
Integer: 0..99999999

Returns the number of the logical block referenced in the last I/O statement.

BLOCKSIZE

General; Read: anytime, Write: closed
Integer: 1..999999
Default: size of one record

Specifies the length of a block in digits. Should be a multiple of the size of a record (MAXRECSIZE).

BLOCKSTRUCTURE

General; Read: anytime, Write: closed
Mnemonic: FIXED, VARIABLE(*)
Default: FIXED

Specifies the format of the records in a block.

BUFFERS

General; Read: anytime, Write: closed
Integer: 0..9, 0 is special
Default: 1

Specifies the number of buffers assigned to the file. Non-standard interpretation of zero, indicating that the user will supply his own buffer.

Appendix C FILE ATTRIBUTES (Continued)

CREATIONDATE

TAPE; Read Only: Assigned
Integer: DISPLAY 0..99366

Returns the julian date (YYDDD format) that the file was created.

CURRENTBLOCK

General; Read: anytime, Write: never
Integer: 1..999999

Returns the size of the block currently in use.

NOTE: This is always equal to BLOCKSIZE, except for short block tape files (*).

CURRENTRECORD

General; Read Only: anytime
Integer: 1..39996

Return the size of the last record referenced by an I/O statement.

NOTE: This value is always equal to MAXRECSIZE.

CYCLE

TAPE; Read: assigned, Write: closed
Integer: 1..99
Default: 1

Specifies different generations of a permanent file.

The value of CYCLE attribute may be set only when the file is closed.

DENSITY (*)

TAPE; Read: anytime(*), Write: closed
Mnemonic: BPI556, BPI800, BPI1600, BPI6250
Default: BPI1600

Specifies the recording density of a magnetic tape file.

Appendix C FILE ATTRIBUTES (Continued)

DIRECTION

TAPE; Read: anytime, Write: closed
Mnemonic: FORWARD, REVERSE
Default: FORWARD

Specifies the direction in which records of the file will be accessed.

EXTMODE

PUNCH; Read: anytime, Write: closed
Mnemonic: EBCDIC

Specifies the character type of the file.

FAMILYNAME

DISK/DISKPACK/PRINTER/PUNCH/TAPE;
Read: anytime, Write: closed
STRING (6)

Specifies the family on which the file resides. Non-standard interpretation for printer and punch, which specifies the form name to be used (see FHS - FORMID).

FILENAME

General; Read: anytime, Write: closed
STRING (6)
Default: Value of INTNAME attribute

Specifies the external file name to be associated with the logical file.

FILESTATUS

General; Read Only: anytime
Mnemonic: OPEN, CLOSED

Returns whether or not the file is currently open. Non-standard (see FHS - OPEN).

Appendix C FILE ATTRIBUTES (Continued)

FORCEIO

DISK/DISKPACK; Read: anytime; Write: anytime
BOOLEAN
Default: FALSE

Forces a physical I/O to retrieve the record, does not examine buffers for the block.

NOTE: Applicable only to RANDOM access files. Non-standard.

FORMS

PRINTER/PUNCH; Read: anytime, Write: closed
BOOLEAN
Default: FALSE

Specifies that the file requires special forms when printed/punched. Non-standard (see FHS - FORMID).

INTNAME

General; Read: anytime, Write: declaration only
STRING (6)
Default: first six characters of internal file name translated to upper-case.

Specifies the name by which the file may be label equated. Non-standard interpretation.

KIND

General; Read: anytime, Write: closed
Mnemonic: DISK, DISKPACK, PRINTER, PUNCH READER, REMOTE, TAPE, DCP, ISC
Default: READER if MYUSE = IN
PRINTER if MYUSE = OUT
DISK(*) if MYUSE = IO or OI

Specifies the peripheral associated with the logical file. REMOTE is the KIND required to do datacomm I/O.

Appendix C FILE ATTRIBUTES (Continued)

LABEL

General; Read: anytime(*), Write: closed
Mnemonic: EBCDICLABEL, OMITTED
Default: EBCDICLABEL

Specifies whether or not the file has label records associated with it.

LASTRECORD

DISK/DISKPACK; Read Only: assigned
Integer: 0..99999999

Returns the record number of the last record in the physical file.

NOTE: May not be correct during file expansion.

MAXRECSIZE

General; Read: anytime, Write: closed
Integer: 1..39996
Default: size of one record

Specifies the size in digits of a record, or the maximum size of a VARIABLE length record (*).

NOTE: At declaration time, any supplied value will be overriden with the record size of the file.

MYUSE

General; Read: anytime, Write: closed
Integer: IN, IO, OI, OUT, EXTEND (*)
Default: IO

Specifies how the file will be used. Non-stanard values are OI and EXTEND (see FHS - NEWFILE).

NEXTRECORD

General; Read Only: anytime
Integer: 0..99999999

Returns the current position of the file.

Appendix C FILE ATTRIBUTES (Continued)

OPTIONAL

General; Read: anytime, Write: closed
BOOLEAN
Default: FALSE

Specifies whether or not the assignment of a permanent file is optional.

PARITY

TAPE; Read: anytime, Write: closed
Mnemonic: EVEN, ODD
Default: ODD

Indicates the parity to be used when writing a tape file.

NOTE: EVEN parity may be used with 7-track tapes only.

PROTECTION

DISK/DISKPACK; Read: anytime, Write: closed
Mnemonic: ABNORMALSAVE, TEMPORARY
Default: TEMPORARY

Specifies whether or not a file open at abnormal EOJ will be saved (and entered in the directory).

SAVEFACTOR

TAPE; Read: anytime, Write: closed
Integer: 0..999
Default: 999

Specifies the expiration date of the file in terms of days past the creation date. Non-standard default value.

SECURITYFAMILY

DISK/DISKPACK; Read: anytime, Write: closed
STRING (6)
Default: DISK

Specifies the family name on which the guard file for the file resides. Non-standard.

Appendix C FILE ATTRIBUTES (Continued)

SECURITYGUARD

DISK/DISKPACK; Read: anytime, Write: closed
STRING (6)
Default: NONE, must be set if needed

Specifies the file name of the guard file for the physical file.

SECURITYTYPE

DISK/DISKPACK; Read: anytime, Write: closed
Mnemonic: DEFAULT, GUARDED, NONE, PRIVATE,
PUBLIC
Default: DEFAULT

Specifies who, other than the owner (creator) may access the permanent file. Non-standard values are DEFAULT and NONE. DEFAULT gives the user's USERHO default security. NONE produces an unsecured file.

SECURITYUSE

DISK/DISKPACK; Read: anytime, Write: closed
Mnemonic: IN, IO, OUT, SECURED
Default: IO

Specifies the manner in which a file protected by security may be accessed.

NOTE: The use of a guard file overrides this attribute function.

SENSITIVEDATA

DISK/DISKPACK; Read: anytime, Write: closed
BOOLEAN
Default: FALSE

Specifies whether or not the disk sectors to be returned when the file is purged are to be overwritten with an arbitrary pattern so that the original data is erased.

Appendix C FILE ATTRIBUTES (Continued)

SERIALNO

TAPE; Read Only: assigned
STRING (5)

Returns the serial number of the labeled tape containing the physical file.

SINGLEUNIT

DISKPACK: Read: anytime, Write: closed
BOOLEAN
Default: FALSE

Specifies whether or not all areas of the file are to be contained on a single member of a family.

UNIQUENAME

DISK/DISKPACK; Read: anytime, Write: closed
Mnemonic: NULL, PROCESSOR, WORK
Default: NULL

Specifies whether or not the file name is to be altered in a unique fashion for a particular execution by the processor number only or the processor number and the mix number (WORK). NULL indicates no uniqueness. Non-standard.

VOLUMEINDEX

TAPE; Read Only: assigned
Integer: 1..999

Specifies the reel number (not the same as SERIALNO) of the physical file.

## Appendix D SYNTAX DIAGRAMS

Syntax diagrams are used to describe the syntactic attributes of languages in a readable manner. Syntax diagrams show how to combine elements of the language being described. These combinations are shown by the one-way paths that connect language elements. The direction of flow is implied by the angles at which paths join or fork. Language elements are primitive symbols (e.g., ASSERT, :=) and phrase-names (e.g., statement, identifier), which represent collections of primitives. Each phrase name is defined by a syntax diagram that shows what collections of primitives it represents. To determine allowable combinations of language primitives, replace a phrase name by any of the collections of primitives it represents.

The following rules define the manner in which syntax diagrams are constructed and the meaning of the special characters used in the diagrams.

1. Path Characters and Directions

   Continuous underline characters (___) denote horizontal direction, either left to right or right to left.

   Vertical bars (|) denote upward or downward directions.

   Directional slashes (/ and \) are used to indicate a branch in the path or a change in the direction of a path. To follow the syntax diagram, start with the item being defined (upper left) and follow the path.

   Directional slashes are also used to fold a long line. Folding of the main path to a continuation line must be the full width of the diagram from the right margin to the left margin. For example:

```
main path fold
|
|
\____THIS_________IS_________AN_________EXAMPLE___OF________
                                                            \
 ___________________________________________________________/
/
\____THE_____FOLDING____OF____A___MAIN____PATH___________
                                                            \
                                                            |
                                                            |
```

Appendix D SYNTAX DIAGRAMS (Continued)

2. Repetitions

Repetitions are always shown by a counter-clockwise loop. The return path travels from right to left, above the repeated item. Constructs which have zero or more occurrences of a single item are shown as:

```
|
|
|      __separator__
|     /             \
\____\____item______/____________________________________________
   \_________________/                                            \
                                                                   |
                                                                   |
```

An integer, preceded by a slash (/) and followed by a reverse slash (\), denotes the maximum number of times a path may be traversed.

Example:
```
__/3\__     maximum of three traverses
```

An integer, preceded by a reverse slash (\) and followed by a slash (/) prior to the construct, denotes the minimum number of times the path must be traversed.

Example:
```
__    __    minimum of two traverses
  \2/
```

If a minimum and a maximum number of traverses are specified, and the minimum and maximum numbers are the same, the number then specifies the exact number of traverses.

Example:
```
__/3\    __    exactly three traverses
      \3/
```

Specification of path traverses may appear in the main line of the path or in a repetition loop:

```
   _________ , _________              ____/9\__ , ________
  /                     \            /                     \
__\__/10\___id__________/__        __\__________id________/__
            ==>                                 ==>
```

Appendix D SYNTAX DIAGRAMS (Continued)

Both of the examples above mean the same; the numbers are different because they occur in different parts of the loop.

3. Start of Diagram

A railroad syntax diagram is started by the phrase name starting at the left margin on a line of its own where the main path begins.

4. Alternate paths

Alternate paths must always be vertically separated from the main path, or other alternate paths, by one line of blanks between directional slashes. Two directional slashes are used denoting the start and return for a single alternate path. A combination of directional slashes and vertical bars is used to denote the start and return of multiple alternate paths. For example:

```
diagram
|
|
\_______mainpath__________________________________________
     \                       /                            \
      \__single__alternate__/                             |
                                                          |
                                                          |

diagram
|
|
\_______mainpath__________________________________________
     \                       /                            \
     |\__first__alternate___/|                            |
     |                       |                            |
     |\__second__alternate__/|                            |
     |                       |                            |
     \___nth__alternate______/                            |
                                                          |
                                                          |
```

5. Termination of Diagram

A railroad syntax diagram is terminated by a vertical line down the right hand edge of the diagram.

Appendix D    SYNTAX DIAGRAMS (Continued)

```
example
|
|
\_______a_______short_________diagram______________________
                                                           \
                                                           |
                                                           |
                                                         (end)
```

6.  Syntactic Constructs

Primitives that are special characters or lower-case reserved words are delimited by a blank before and after them to provide visual separation from the path.

Phrase names are lower-case words. These and upper-case primitives (key words or reserved words) are not delimited by blanks but are immediately adjacent to the path's underline characters. Phrase names may be immediately preceded by a comment word and a colon. The comment and colon may be on the line directly above the phrase name or on the same line as the phrase name.

```
diagram
|
|  |                        <---
|  |                  ______ , ______
|  V                 /               \
\____________________\_______a_______/_____________________
        --->            \   --->  /          --->          \
                        |\___b___/|                        |
                        |         |                     |  |
                        \____c____/                     |  |
                            --->                        V  |
                                                           |
```

Appendix D SYNTAX DIAGRAMS (Continued)

The SPRITE syntax diagrams below specify the syntax for the complete language design although not all constructs have been implemented.

```
interface description
|
|                         _____________ ; _________
|   program name:        /                         \
\__identifier__PROG__\_____________component__/________GORP________
                      \__knows__/               /  \_ ; _/    \_ ; _/ \
                       \_comment__________/                            |
                                                                       |
                                                                       |

component
|
|\_________module description__________________________________________
|                                                                       \
|\_________data definition____________________________________________ |
|                                                                      \|
|\_________file definition____________________________________________ |
|                                                                      \|
|\_________port definition____________________________________________ |
|                                                                      \|
|\_________nsp file definition________________________________________|
|                                                                      \|
|\_________type definition ___________________________________________ |
|                                                                      \|
|\_________constant definition________________________________________ |
|                                                                      \|
\__________declaration block__________________________________________ |
                                                                       \|
                                                                        |

comment
|
|                        __________________
|                       /                  \
\______COM_________\_____text_________/____MOC________________
                      \__comment__/                            \
                       \________/                               |
                                                                |
                                                                |
```

Appendix D SYNTAX DIAGRAMS (Continued)

```
knows
|
|
|
|            __________ , __________
|           /      module name:      \
\_____ { ___\______identifier_______/___ } ______KNOW__________________
          \___________ALL____________/        \__KNOWS__/              \
                                                                         |
                                                                         |


module description
|
|    module name:
\____identifier______MOD______________________________
                          \___ENV_DEPENDENT___/    \
 __________________________________________________/
/
|    _____________________;_______________
|   /              entry point:            \
\___\_____________procedure description__/___________DOM______________
      \__knows__/                        /   \_ ; _/                   \
       \___________comment______________/                               |
                                                                         |
                                                                         |


procedure description
|
|
|
|  proc name:
\__identifier____PROC______parameters______return value________________
                \       \______________/ \________________/            \
                 \__ENTRY_______________________________/               |
                                                                         |
                                                                         |


parameters
|
|
|         ___ , ____________________________________________
|        /  param: \                                        \
\__ ( __\__ident__/_________________________________type__/__ ) ___
                     \___CONST___/  \__UNIV__/                          \
                      \___VAR___/                                        |
                       \_VALUE_/                                         |
                                                                         |
```

Appendix D SYNTAX DIAGRAMS (Continued)

```
return value
|
|
|
\___RETURNS_____type_______________________________________________
                                                                    \
                                                                    |
                                                                    |


data definition
|
|                                        ___________ , ___________
|  data block name:                     /                         \
\_____identifier________DATA_____\___________variable_________/_____
                                                                    \
                                                                    |
                                                                    |


variable
|
|             _______ , ______
|       / variable name: \
\_____\___identifier___/____type___________________________________
                              \__init value__/                      \
                                                                    |
                                                                    |


init value
|
|
\_______ := ______constant: expr___________________________________
                                                                    \
                                                                    |
                                                                    |


file definition
|                                        ________ , ________
|      file block name:                 /                   \
\________identifier_____FILE_______\__file description__/__________
                                                                    \
                                                                    |
                                                                    |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
file description
|
|     file_name:
\_____identifier_____file attr spec_____OF______type________________
                                       \___/                        \
                                                                    |
                                                                    |


file attr spec
|
|         _________________ , _______________________
|        /                                           \
\____ [ _\__attribute name__ = _____attribute value__/___ ] _______
                           \_________________________/             \
                                                                   |
                                                                   |


port_block
|
|                             __________ , __________
|                           /                        \
\___port_block_name__PORT___\___port_description___/___ ; _______
                                                                  \
                                                                  |
                                                                  |


port_description
|
|
\___________port_name___________port_attribute_spec_______________
                                                                  \
                                                                  |
                                                                  |


port_attribute_spec
|
|
|        _________________ , _______________________________
|       /                                                   \
\__ [ __\_attr_name__________________________________________/_ ] __
                    \                  /\                 /          \
                     \_ [const_expr] _/  \_= attr_value_/            |
                                                                     |
                                                                     |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
nsp_file_block
|
|                                        _________ , ___________
|                                       /                        \
\__nsp_file_block_name __NSP__\__nsp_file_description__/__ ; _____
                                                                   \
                                                                   |
                                                                   |


nsp_file_description
|
|
\___nsp_file_name____nsp_file_attribute_specification____________
                                                                  \
                                                                  |
                                                                  |


nsp_file_attribute_spec
|
|                ________________ , _________________
|               /                                     \
\__ [ __\__attr_name______________________________/__ ] _________
                          \_____= attr_value____/                 \
                                                                  |
                                                                  |


declaration block
|
|                  _____________ ; ______________
|                 /                               \
\____DEC___\_____constant definition_______/_____________CED______
                  \                              /   \__ ; __/        \
                  |\___type definition__________/|                    |
                  |                              |                    |
                  |\___data definition__________/|                    |
                  |                              |                    |
                  |\___file definition__________/|                    |
                  |                              |                    |
                  |\___port definition__________/|                    |
                  |                              |                    |
                  |\___nsp file definition______/|                    |
                  |                              |                    |
                  \____comment___________________/                    |
                                                                      |
                                                                      |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
constant definition
|
|
|             _____________________ , _____________________
|           /   constant:                          constant:  \
\__CONST____\___identifier________________ = ____expr______/_______
                             \__type__/                            \
                                                                   |
                                                                   |


type definition
|
|             _____________________ , _____________________
|           /    type name:                                  \
\___TYPE____\_____indicant__________ = _____type_____________/______
              \__________parametric type part___________/           \
                                                                    |
                                                                    |


parametric type part

|
|
|
|                                      bounds:
\____indicant__ ( __ param:ident____subrange___ ) ___ = __
                                                          \
 _________________________________________________________/
/
\____STRING__ ( __param:ident__ ) __________________________________
|                                  \     \  element:string  /      \
|                                   \__OF__\___base type____/      |
|                                                                  |
|             lower bound:        param:           element:        |
\____ARRAY__ [ __constant___ .. ___ident__ ] ___OF____type________ |
                                              \____/              \|
                                                                   |
                                                                   |
```

Appendix D SYNTAX DIAGRAMS (Continued)

```
type
|
|
|
|\__simple_type_________________________________________________________
|                                                                      \
|\__indicant_________________________________________________________  |
|                                                                      |
|\__SET____OF____finite scalar type_______________________________    |
|        \____/                                                        |
|                                                                      |
|                 length:                                              |
|\__STRING__ ( __simple expr__ ) ____________________________________  |
|                                  \           string      /          \|
|                                    \__OF___base type___/              |
|                                      \__/                             |
|                  ______ , ____________                                |
|                 /     index:          \                               |
|\__ARRAY___ [ __\__finite scalar type /__ ] ___OF____type________     |
|                                              \____/                  \|
|                                                                      |
|\__PTR___TO_________________________________type_______________       |
|        \__/ \ \_CONST_/   \___EXTERNAL___/                          \|
|              | \_VAR_/     \___LOCAL_____/                            |
|      ________/                                                        |
|     /                                                                 |
|     \___PROC_________________________________________________________ |
|              \                  / \                 /                \|
|               \___parameters___/   \_return value_/                   |
|                                                                      |
|                        _______ , _______                              |
|                       /                 \                             |
\______________STRUC___\_____field_______/___CURTS_______________      |
  \_PACKED_/                                                          \|
                                                                       |
                                                                       |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
simple type
|
|
|
|\__finite scalar type__________________________________________
|                                                               \
|\__REAL_______________________________________________________ |
|                                                              \|
\___LONG_REAL__________________________________________________ |
                                                               \|
                                                                |
                                                                |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
finite scalar type
|
|
|\___finite scalar indicant__________________________________________
|                                                                     \
|                              ______ , ______                        |
|                             /     symbolic:   \                      |
|\___SYMBOLIC________  (  ___\_____ident_____/___  )  _______________ |
|  \__ORDERED__/                                                     \|
|                                                                     |
|      constant:            constant:                                 |
|\___simple expr__  ..  __simple expr________________________________ |
|                                                                    \|
|                   integer subrange:                                 |
|\___DISPLAY___finite scalar type____________________________________ |
|                                                                    \|
|                  number_of_bits:                                    |
|\___BINARY_  (  _integer expr_  )  _________________________________ |
|                                 \                          /      \|
|                        ________/                           |       |
|                       /             lwb:              upb:  |       |
|                       \_RANGE__integerexpr_  ..  __integerexpr_/    |
|                                                                     |
|\___BOOLEAN_________________________________________________________ |
|                                                                    \|
|\___CHAR____________________________________________________________ |
|                                                                    \|
|\___EBCDIC__________________________________________________________ |
|                                                                    \|
|\___HEX_____________________________________________________________ |
|                                                                    \|
|\___BIT_____________________________________________________________ |
|                                                                    \|
|                                                                     |
|                              ________ , ___________                 |
|                             /         packed       \                |
\____PACKED___SYMBOLIC___  (  __\__    __scalar value__/__  )  ______ |
              \_ORDERED_/        \ \1/                /             \|
                                  \__________________/               |
                                                                     |
                                                                     |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
string base type
|
|
|\___CHAR____________________________________________________________
|                                                                    \
|\___EBCDIC_________________________________________________________ |
|                                                                   \|
\____HEX____________________________________________________________ |
                                                                    \|
                                                                     |
                                                                     |


packed scalar value
|
|
|
\______scalar:identifier_____________________________________________
                                \__ = __constant:expr__/             \
                                                                      |
                                                                      |


field
|
|             _____ , ______
|          /  field name:  \
|\______\__identifier__/______type_______________________________
|                                                                 \
\_____CASE___tag field____IS__________variants___ESAC____________ |
                                                                 \|
                                                                  |
                                                                  |


tag field
|
|
|\___normal tag______________________________________________________
|                                                                    \
|\___omitted tag____________________________________________________ |
|                                                                   \|
\____disjoint tag___________________________________________________ |
                                                                    \|
                                                                     |
                                                                     |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
normal tag
|
|
|
|      tag id:                     tag type:
\______identifier_________finite scalar type_____________________
                                                                  \
                                                                  |
                                                                  |


omitted tag
|
|      tag type:
\______finite scalar type________________________________________
                                                                  \
                                                                  |
                                                                  |


disjoint tag
|
|      tag_id:
\______identifier_________________________________________________
                                                                  \
                                                                  |
                                                                  |


variants
|
|      ____________________________ OR ______________________
|   /      ________ , ___________                             \
|   |   /constant variant labels:\                            |
\___\___\___value collection_____/__ : _______________________/__
                                      \   ______ , _______  /     \
                                      | /   variant alt:  \ |     |
                                      \_\_____field_______/_/     |
                                                                  |
                                                                  |


module
|
|
|                              _______ ; _______
|    module name:             /                 \
\____identifier____MOD____\_module component_/__________DOM_________
                                              \_ ; _/     \_ ; _/ \
                                                                   |
                                                                   |
                                                                   |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
module component
|
|
|
|\_________comment______________________________________________
|                                                               \
|\_________type definition _____________________________________ |
|                                                              \|
|\_________constant definition__________________________________ |
|                                                              \|
|\_________data definition______________________________________ |
|                                                              \|
|\_________file definition______________________________________ |
|                                                              \|
|\_________port definition______________________________________ |
|                                                              \|
|\_________nsp file definition__________________________________ |
|                                                              \|
|\_________procedure definition_________________________________ |
|                                                              \|
\__________forward definition___________________________________ |
                                                               \|
                                                                |
                                                                |


forward definition
|
|
\___proc header____FORWARD____ ; _______________________________
                                                                \
                                                                |
                                                                |


procedure definition
|
|
|
|
\___proc header____procedure declarations____statements_____CORP___
                   \_______________________/                       \
                                                                   |
                                                                   |
                                                                   |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
proc header
|
|
|
| procedure name:
\___identifier____PROC_______parameters________return value___  ; __
                           \______________/   \______________/       \
                                                                      |
                                                                      |


proc declarations
|
|    ________________________________________
|   /                                        \
\__\_______comment____________________ ; ___/_______________________
       \                          /                                  \
       |\__shares declaration___/|                                   |
       |                          |                                   |
       |\__type definition _____/|                                   |
       |                          |                                   |
       |\__constant definition__/|                                   |
       |                          |                                   |
       \___var declaration______/                                    |
                                                                      |
                                                                      |


shares declaration
|
|              _____________ , _____________
|             /                      shared:  \
\___SHARES___\______________________ident___/_______________________
                  \__CONST__/                                        \
                   \__VAR__/                                          |
                                                                      |
                                                                      |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
var declaration
|
|              _____________________ , ____________________
|             /      ___ , ___                              \
|             |     /  var:    \                             |
\___VAR___\__\__ident__/___type_______________________________/______
                               \_STATIC_/   \_init value_/          \
                                                                     |
                                                                     |
                                                                     |
```

```
statements
|
|
|                       ____ ; ____
|                      /           \
\______________\_statement_/_______________________________________
                                                                   \
                                                                    |
                                                                    |
                                                                    |
```

Appendix D SYNTAX DIAGRAMS (Continued)

```
statement
|
|
|
|\_____________________________________________________________________
|                                                                     \
|\_________simple statement_________________________________________   |
|                                                                      |
|\_________if statement_____________________________________________   |
|                                                                     \|
|\_________case statement___________________________________________   |
|                                                                     \|
|\_________while statement__________________________________________   |
|                                                                     \|
|\_________do until statement_______________________________________   |
|                                                                     \|
|\_________for statement____________________________________________   |
|                                                                     \|
|\_________find statement___________________________________________   |
|                                                                     \|
|\_________until case statement_____________________________________   |
|                                                                     \|
|\_________generate statement_______________________________________   |
|                                                                     \|
 \_________comment__________________________________________________   |
                                                                     \|
                                                                       |
                                                                       |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
simple statement
|
|
|   destination:                assignment value:
|\____primary______ :=  ___________expr__________________________________
|               \___ +:= ___/                                         \
|               |\__ -:= __/|                                          |
|               |\__ *:= __/|                                          |
|               |\__ /:= __/|                                          |
|               |\__ &:= __/|                                          |
|               |\__ |:= __/|                                          |
|               \___ #:= ___/                                          |
|                                                                      |
|  left hand side:       right hand side:                              |
|\____primary_____ :=: _____primary_______________________________     |
|                                                                    \|
|    proc call:                                                        |
|\____primary_____________________________________________________     |
|                  \____failure____/                                 \|
|                                                                      |
|                  situation:                                          |
|\____LOOP_EXIT______ident________________________________________     |
|                                                                    \|
|               boolean:                                               |
|\____ASSERT_____expr_____________________________________________     |
|                                                                    \|
|             return value:                                            |
|\____RETURN_____expr_____________________________________________     |
|             \______/                                               \|
|                                                                      |
\_____FAIL________________________________________________________     |
                                                                    \|
                                                                     |
                                                                     |


failure
|
|
|
\__IF____FAILURE_____then part_____else part_____FI________________
                                 \___________/                      \
                                                                     |
                                                                     |
                                                                     |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
if statement
|
|                ________ELIF_____________
|          /   boolean:                   \
\__IF____\____expr ____then part____/_____else part____FI_________
                                          \___________/           \
                                                                   |
                                                                   |


then part
|
|
|
|
\____THEN____statements___________________________________________
                                                                   \
                                                                   |
                                                                   |
                                                                   |


else part
|
|
|
|
\____ELSE____statements___________________________________________
                                                                   \
                                                                   |
                                                                   |
                                                                   |


case statement
|
|
|
|            selector:
\____CASE____expr____IS____cases____else part____ESAC_____________
                                    \___________/                 \
                                                                   |
                                                                   |
```

Appendix D SYNTAX DIAGRAMS (Continued)

```
cases
|
|    ________________________OR______________________
|   /     _______ , __________                        \
|   |   /constant case labels: \                      |
\__\__\__value collection____/___ : ____statements___/___________
                                                                 \
                                                                 |
                                                                 |
                                                                 |


while statement
|
|
|
|              boolean:
\___WHILE_____expr_____do group___________________________________
                                                                 \
                                                                 |
                                                                 |


do until statement
|
|
|
|                                boolean:
\_____do group_____UNTIL_____expr_________________________________
                                                                 \
                                                                 |
                                                                 |
                                                                 |


for statement
|
|
|
|       cortrol var:            limits:
\__FOR____ident_______OVER____value range____
                                             \
   __________________________________________/
  /
  \___________________do group_______________________________
     \__DESCENDING__/                                        \
                                                             |
                                                             |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
value range
|
|
|\____simple expr___ .. ___simple expr_____________________________
|                                                                  \
\_____indicant___ . ___RANGE_____________________________________  |
                                                                  \|
                                                                   |
                                                                   |


until case statement
|
|
|
|            _____ , _____
|           /  situation: \
\__UNTIL____\____ident____/____do group____
                                           \
   ________________________________________/
  /
  |                 _________________OR_________________
  |                /    _____ , ____                    \
  | .              |   / situation: \                   |
  \__CASE___IS___\___\___ ident___/__ : __statements___/___ESAC____
                                                                   \
                                                                   |
                                                                   |


do group
|
|
|
|
\____DO____statements____OD________________________________________
                                                                   \
                                                                   |
                                                                   |
                                                                   |
Appendix D    SYNTAX DIAGRAMS (Continued)

find statement
|
|
\___FIND___find control___WHERE___find condition___
                                                   \
                  _________________________________/
                 /
                 \___THEN___statements___ELSE___statements___DNIF__
                                      \___________________/        \
                                                                   |
                                                                   |


find control
|
|
|     pointer:
|\___identifier________________________INTO____array:primary_______
|               \       index:      /                              \
|                \_AND__identifier_/                               |
|                                                                  |
|  element pointer:        predecessor pointer:                    |
\____identifier_____WITH________identifier______                   |
                  \__________________________/  \                  |
                 _______________________________/                  |
                /        list pointer:                             |
                \__FROM_____primary______USING___link fields______ |
                                                                  \|
                                                                   |
                                                                   |


link fields
|
|     ______ . _________
|    /    field name:   \
\____\____identifier____/__________________________________________
                                                                   \
                                                                   |
                                                                   |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
find condition
|
|
|\___MIN____find primary____________________________________________
|  \__MAX__/                                                          \
|                                                                      |
|\___find primary_______  =   _______key:expression________________   |
|                   \___  ¬=  ___/                                  \|
|                   |\__  <   __/|                                   |
|                   |\__  <=  __/|                                   |
|                   |\__  >   __/|                                   |
|                   \___  >=  ___/                                   |
|                                                                     |
|                          set mask:                                  |
\____find primary___  *  ______expr_______  =  ___  {}  ______________|
                                      \_  ¬=  _/                    \|
                                                                     |
                                                                     |


generate statement
|
|
|
|                          pointer:
\__GENERATE____EXTERNAL___primary____________________________________
             \__LOCAL____/          \               integer:       /   \
                                     \___LENGTH___simple expr___/      |
                                                                       |
                                                                       |


expr
|
|            __________  |  _____________
|           /        \___  ||  ___/         \
|           |         \__   #  __/          |
|           |                               |
\__________\______conjunctive expr______/___________________________
                                                                     \
                                                                     |
                                                                     |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
conjunctive expr
|
|            __________  &  ______________
|           /      \___ && ___/            \
|           |                              |
\___________\______relational expr_______/_________________________
                                                                     \
                                                                     |
                                                                     |


relational expr
|
\___simple expr_____________________________________________________
                  \                   set or subr:          /        \
                  |\_______IN_________inclusion____________/|        |
                  |\__ ¬ __IN__/                            |        |
                  \_________  =  __________simple expr______/        |
                      \_____  ¬=  _____/                              |
                      |\____  <=  ____/|                             |
                      |\____  >=  ____/|                             |
                      |\____   <  ____/|                             |
                      \_____   >  _____/                             |
                                                                     |


inclusion
|
|\___simple expr____________________________________________________
|                  \___ .. ___simple expr___/                        \
|                                                                    |
\____indicant___ . ___RANGE_______________________________________ |
                                                                   \|
                                                                    |


simple expr
|
|            __________  +  ______________
|           /      \___  -  ___/           \
|           |                              |
\___________\__________term______________/_________________________
                                                                     \
                                                                     |
                                                                     |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
term
|             __________  * ______________
|            /      \____ / ___/          \
|            |       \___REM__/           |
|            |                            |
\____________\_________factor_____________/___________________________
                                                                     \
                                                                     |
                                                                     |
                                                                     |


factor
|
|
|\__________ + ________factor________________________________________
|       \__ - __/                                                    \
|        \_ ¬ _/                                                     |
|\_______________ ( ______expr_________ ) __________________________ |
|                                                                   \|
|\_______ { ______________________________________ } _______________ |
|              \    __________ , _________     /                    \|
|              |   /    set elements:      \   |                     |
|              \___\___value collection___/___/                      |
|                                                                    |
|               ______ , _______                                     |
|              /aggregate elem:\                                     |
|\_______ [ ___\_____expr______/__ ] _______________________________ |
|                                                                   \|
|          type:                                                     |
|\_______indicant____ . _____________MIN____________________________ |
|                           \________MAX________/                   \|
|                           |\______DELTA______/|                    |
|                           |\_____EPSILON_____/|                    |
|                           \_______SIZE________/                    |
|                                                                    |
|                                                                    |
\________primary____________________________________________________ |
                                                                    \|
                                                                     |
                                                                     |
```

Appendix D    SYNTAX DIAGRAMS (Continued)

```
orimary
|
|
|   var or constant:
|\______ident________________________________________________________
|                                                                    \
|                proc or function:                                   |
|\____________________ident_______________________________________  |
|   module:           /                                             \|
|\___ident__ . __   /                                                |
|                                                                    |
|\___literal______________________________________________________  |
|                                                                   \|
|   cast type:         cast argument:                                |
|\___indicant____  (  ____expr_______  )  ________________________  |
|                                                                   \|
|\___primary______selection_______________________________________  |
|                       _______ , _______                           \|
|                      /actual parameter:\                           |
|\___primary_____  (  __\__expression_____/__  )  ________________  |
|                                                                   \|
|      file,                                                         |
|      port or nsp                                                   |
|      filename:               attribute name:                       |
\______identifier____  .  _____indicant___________________________  |
                                                                   \|
                                                                    |


selection
|
|
|\____  .  ____field:identifier______________________________________
|                                                                    \
|            _________ , __________                                  |
|           /   subscript/slice:   \                                 |
|\____  [  ___\______indexing_______/___  ]  _____________________  |
|                                                                   \|
\_____  @  _______________________________________________________  |
                                                                   \|
                                                                    |
                                                                    |
```

Appendix D SYNTAX DIAGRAMS (Continued)

```
indexing
|
|
|\___expr_____________________________________________________________
|           \                                    /                      \
|            \___ .. _____simple expr___/                                |
|             \__ :: __/                                                 |
|                                                                        |
\____indicant__ . __RANGE_____________________________________________   |
                                                                      \|
                                                                       |
                                                                       |


literal
|
|
|\______ " _______string_________ " ____________________________________
|                                                                        \
\______________integer_________________________________________________   |
                                                                      \|
                                                                       |
                                                                       |


value collection
|
|
|\_____expr_____________________________________________________________
|                                         /                              \
|\_____simple expr .. simple expr__/                                      |
|                                       /                                 |
\______indicant . RANGE__________/                                        |
                                                                          |
                                                                          |
```

## Appendix E SPRITE CODED EXAMPLES

```
% MID - binary tree traversal example

tree_traversal
PROG

trees
MOD
    traversal ENTRY;
DOM;

{trees} KNOWS
scanner
MOD
    get_ch     PROC (char VAR STRING (1) OF CHAR);
DOM;

{trees, put} KNOW
DEC
    CONST max_int = 99999999999999999999999;
    TYPE PUT_STR (s 1..100) = STRING (s),
         PUT_INTEGER        = - max_int..max_int;
CED;

{trees} KNOWS
put            % an intrinsic - See Appendix H.
MOD
    blanks     PROC (size 1..100);
    eol        PROC;
    new_page   PROC;
    number     PROC (value PUT_INTEGER, size 1..100);
    string     PROC (str PUT_STR);
DOM;

GORP

% binary tree traversal example - SPRITE module

trees
MOD

TYPE POINTER = PTR TO NODE,
     NODE = STRUC
                 info    STRING (1) OF CHAR,
                 llink,
                 rlink   POINTER
             CURTS;

preorder
PROC (p POINTER);
```

Appendix E SPRITE CODED EXAMPLES (Continued)

```
          IF    p ¬= nil
          THEN
                put.string (p@.info);
                preorder   (p@.llink);
                preorder   (p@.rlink);
          FI;
     CORP;     % preorder

     inorder
     PROC (p POINTER);

          IF    p ¬= nil
          THEN inorder    (p@.llink);
               put.string (p@.info);
               inorder    (p@.rlink);
          FI;
     CORP;     % inorder

     postorder
     PROC (p POINTER);

          IF    p ¬= nil
          THEN postorder  (p@.llink);
               postorder  (p@.rlink);
               put.string (p@.info);
          FI;
     CORP;     % postorder

     enter
     PROC (p VAR POINTER);
     VAR ch STRING (1) OF CHAR;

          scanner.get_ch (ch);
          put.string (ch);
          IF    ch ¬= "."
          THEN GENERATE EXTERNAL p;
               p@.info := ch;
               enter (p@.llink);
               enter (p@.rlink);
          ELSE p := nil;
          FI;
     CORP;     % enter

     traversal
     PROC;
     VAR root POINTER;

          put.string (" ");              % See Appendix H.
```

Appendix E SPRITE CODED EXAMPLES (Continued)

```
          IF    p ¬= nil
          THEN
                out.string (p@.info);
                preorder   (p@.llink);
                preorder   (p@.rlink);
          FI;
      CORP;     % preorder

      inorder
      PROC (p POINTER);

          IF    p ¬= nil
          THEN inorder    (p@.llink);
                put.string (p@.info);
                inorder    (p@.rlink);
          FI;
      CORP;     % inorder

      postorder
      PROC (p POINTER);

          IF    p ¬= nil
          THEN postorder  (p@.llink);
                postorder  (p@.rlink);
                out.string (p@.info);
          FI;
      CORP;     % postorder

      enter
      PROC (p VAR POINTER);
      VAR ch STRING (1) OF CHAR;

          scanner.get_ch (ch);
          out.string (ch);
          IF    ch ¬= ".";
          THEN GENERATE EXTERNAL p;
                p@.info := ch;
                enter (p@.llink);
                enter (p@.rlink);
          ELSE p := nil;
          FI;
      CORP;     % enter

      traversal
      PROC;
      VAR root POINTER;

          out.string (" ");             % See Appendix H.
```

Appendix E    SPRITE CODED EXAMPLES (Continued)

```
          enter      (root);
          put.eol;
          put.string (" ");
          preorder   (root);
          put.eol;
          put.string (" ");
          inorder    (root);
          put.eol;
          put.string (" ");
          postorder  (root);
          put.eol;
     CORP;

     DCM
```

Appendix E    SPRITE CODED EXAMPLES (Continued)

```
            enter       (root);
            put.eol;
            put.string (" ");
            preorder    (root);
            put.eol;
            put.string (" ");
            inorder     (root);
            put.eol;
            put.string (" ");
            postorder  (root);
            put.eol;
        CORP;

        DCM
```

## Appendix F RESERVED WORDS

Reserved words establish the structural and semantic context of the program. Reserved words and their reversed spellings are used to bracket parts of certain constructs (e.g., WHILE condition DO statement list OD;). Reserved words may not be used as data type names. The reserved words are:

| | | | |
|---|---|---|---|
| ALIAS | ELIF | LENGTH | PTR |
| ALL | ELSE | LOCAL | RANGE |
| AND | ENTRY | LOOP_EXIT | REM |
| ANY_ONE_BIT_IN | ENV_DEPENDENT | MACRO | REMAPS |
| ARRAY | EPSILON | MATCHES | RETURN |
| ASSERT | ESAC | MAX | RETURNS |
| BINARY | EXTERNAL | MIN | SET |
| BIT | FAIL | MOC | SHARES |
| BOOLEAN | FAILURE | MOD | SIZE |
| CASE | FI | NO_ONE_BIT_IN | STATIC |
| CED | FILE | NSP | STRING |
| COM | FIND | OD | STRUC |
| CONST | FOR | OF | SYMBOLIC |
| CORP | FORWARD | OR | THEN |
| CURTS | FROM | ORCAM | TO |
| DATA | GENERATE | ORDERED | TYPE |
| DEC | GORP | OVER | UNIV |
| DELTA | IF | OVERLAY | UNTIL |
| DESCENDING | IN | PACKED | USING |
| DISPLAY | INTO | PORT | VALUE |
| DNIF | IS | PROC | VAR |
| DO | KNOW | PROCESS_RUN | WITH |
| DOM | KNOWS | PROG | |

## Appendix G SUGGESTED SPRITE CODING CONVENTIONS

### G.1 INDENTATION

To make the logic of a program more readable the contents of that program's units should be appropriately indented to reflect their structure.

The bracketing pairs: PROG - GORP, MOD - DOM, DEC - CED, and PROC - CORP should always be lined up.

In the MID (outside DEC blocks) and in SPRITE code (outside a PROC), the words: TYPE, CONST, DATA and SHARED also must be lined up.

In a MID DEC block, the words: TYPE, CONST and DATA are indented 5 spaces. In a SPRITE PROC, the words: SHARES, VAR, TYPE and CONST are also indented 5 spaces.

Key words must be aligned on the same margin and things that they bracket must be appropriately indented. A constant indentation of five spaces has been agreed to for all constructs.

6.1 INDENTATION (Continued)

For example:

```
IF xxxx
THEN
     xxxxxxxx
ELSE
     xxxxxxxxx
FI
```

or (using indent of 5 spaces)

```
CASE xxxx
IS l1:
     xxxxxxxx
OR l2:
     xxxxxxxxxx
ELSE
     xxxxx
ESAC
```

To allow easy changes to their lists, the keywords: DATA, SHARED, VAR, SHARES, TYPE and CONST should be on lines by themselves. The keywords THEN, ELSE, DO, OD, DEC, CED, COM, MOC and the case labels following IS and OR should also be on lines by themselves.

## G.2 SPACING

To make individual lines easier to read, there should be a liberal use of blanks. This is especially important around special symbols, after commas, before parentheses, etc.

There should be only one statement per line.

Boolean conditions in IF, WHILE, etc., which involve anything more complex than simple boolean variables, should have the conditions divided up one per line.

As a general rule, when it becomes necessary to split an expression across more than one line, the operator or symbol at which the split occurs goes on to the second line with the rest of the expression.

Blank lines should be used to separate distinct groups of statements. For example, between the proc heading and any SHARES; between the SHARES and any VARs; between the VARs and the statements; and between the statements and the CORP.

## G.2 SPACING

To make individual lines easier to read, there should be a liberal use of blanks. This is especially important around special symbols, after commas, before parentheses, etc.

There should be only one statement per line.

Boolean conditions in IF, WHILE, etc., which involve anything more complex than simple boolean variables, should have the conditions divided up one per line.

As a general rule, when it becomes necessary to split an expression across more than one line, the operator or symbol at which the split occurs goes on to the second line with the rest of the expression.

Blank lines should be used to separate distinct groups of statements. For example, between the proc heading and any SHARES; between the SHARES and any VARs; between the VARs and the statements; and between the statements and the CORP.

## 6.3 PROCEDURES IN SPRITE

ALL procedures will be declared FORWARD. This will make it possible to write and read the module in a top down fashion.

Each actual procedure name will be set off in a box for ease of recognition. (A model in an editor file should be available.) The procedure name should also be repeated after the CORP as a comment.

Parameters (and RETURNS) should be written one to a line and commented if there is the slightest doubt what they are used for.

Following the procedure's header, there should be a full description of what process this procedure performs. This should include any railroad syntax diagrams as well as a written description.

Any SHARES statements should have their data block names one per line, note the usage of that data block and where it originated.

Variables should be declared one per line and have their usage documented if the names are not exactly descriptive.

Groups of statements within the procedure which perform some sub-function should be separated by blank lines and their method or function should be documented.

## G.3 PROCEDURES IN SPRITE

ALL procedures will be declared FORWARD. This will make it possible to write and read the module in a top down fashion.

Each actual procedure name will be set off in a box for ease of recognition. (A model in an editor file should be available.) The procedure name should also be repeated after the CORP as a comment.

Parameters (and RETURNS) should be written one to a line and commented if there is the slightest doubt what they are used for.

Following the procedure's header, there should be a full description of what process this procedure performs. This should include any railroad syntax diagrams as well as a written description.

Any SHARES statements should have their data block names one per line, note the usage of that data block and where it originated.

Variables should be declared one per line and have their usage documented if the names are not exactly descriptive.

Groups of statements within the procedure which perform some sub-function should be separated by blank lines and their method or function should be documented.

## G.4 MEANINGFUL NAMES

All names should explain the usage or function they represent. This should be as exact as possible without getting carried away in name length. It is especially necessary that module and procedure names reflect their function.

TYPE names should be fully descriptive. Pointers to types should be descriptive of both the type pointed to and the fact that the type is a pointer, i.e., CHAR_PTR.

SYMBOLIC identifiers should reference the parent type's name or some abbreviation of it. For example:

```
    TYPE
         SYMBOL = SYMBOLIC (
                             sym_addop,
                             sym_subop
                            );
```

NOTE: With field names, don't get too flowery as there tend to be several required to get to the field needed which makes for long qualification sequences anyway.

## 6.5 MODULES IN THE MID

The module descriptions should be arranged alphabetically by module name.

Each procedure should be on a separate line and documented as to its usage. Each parameter of a procedure should be on a separate line, and should be commented as necessary.

Example:

```
    read
    MOD
        read_card PROC (card_image STRING (80),
                            eof_flag    BOOLEAN);
                   COM
                       gets the next input record
                       and returns it in card_image.
                       When end of file occurs,
                       eof_flag is set true.
                   MOC;
    DOM;
```

## 6.6 DATA BLOCKS

These should be arranged alphabetical by block name. The variables should be given one per line and documented as necessary.

The names of the variables in the data block should reflect the name for the data block or some abbreviation of it. This is to assist in determining which data block a variable used in a procedure is shared from.

Example:

```
    error_counter
    DATA
         error_count 0 .. max_errors + 1 := 0;
                    % # errors detected
```

## 6.7 KNOWS LISTS AND DEC - CED BLOCKS

Due to the necessity of hiding as much information as possible, there will be extensive use of comprehensive KNOWS lists. DEC - CED blocks should be used to encapsulate inter-dependent constants, types and data blocks under one KNOWS list.

The KNOWS list itself should be indented as shown below and should have only one module name per line for easy changes.

Example:

```
            {
             module_one,
             module_two
            } KNOW
            % input file buffer information
      DEC
            CONST
                  input_size = 80;

            TYPE
                 INPUT_LINE = STRING (input_size);

            input_buffer
            DATA
                 input_line INPUT_LINE := "%";
                            % to force first read
      CED;
```

NOTE: Things in DEC-CED blocks are indented to set them off. Names should show the inter-dependencies (i.e., 'input' above). It would probably be useful to organize the DEC blocks alphabetically by this inter-dependent name.

## 6.8 TYPES AND CONSTS

Within a KNOWS list or a DEC block, multiple type or constant identifiers should be arranged alphabetically.

Within each level forced by KNOWS lists or the compiler's predefinition requirements, the type and constant names should be arranged alphabetically.

## Appendix H INTRINSICS

Certain standard modules are provided with the SPRITE language System as intrinsics. These include arm, binary_convert, compare, debug (various flavors), dbwrite, err (two flavors), move, print, put, readcd and trace. Calls to some of these intrinsics are generated implicitly by the SPRITE compiler.

When SPRITE generates an implicit call to an intrinsic, it uses the intrinsic's module name instead of an actual ICM name.

### H.1 INTRINSICS INTERFACE

If explicit calls to the SPRITE intrinsics are to be made, their MID descriptions must be embedded in the user's MID. In the following MID descriptions of the intrinsics, the comments describe the functions of the modules.

H.1 INTRINSICS INTERFACE (Continued)

```
------------------------------------------------------------
|                                                          |
|                    I N T R I N S I C S                   |
|                                                          |
------------------------------------------------------------
```

```
arm                         %  SPRITE module:  arms the program.
MOD  ENV_DEPENDENT
    processor_and_trap
        PROC (error_handler_proc_ptr VALUE PTR TO PROC);

             %  This routine does an arm BCT and enables the
             %  accumulator trap.  If any type of error happens,
             %  the specified procedure is called.  Information
             %  about the error is in 'arm_parameters' data block.

    get_error_message PROC (display_error_message BOOLEAN)
                          RETURNS STRING (100);

             %  This routine determines the specific type of
             %  error that happened and returns an error message
             %  describing the error.  If requested, it will
             %  display this error message.

DOM;


binary_convert
MOD

    to_decimal PROC (hex CONST UNIV HEXADECMAL,
                     dec VAR UNIV DECIMAL);
```

H.1 INTRINSICS INTERFACE (Continued)

```
        to_binary  PROC (dec CONST UNIV DECIMAL,
                         hex VAR UNIV HEXADECIMAL);

                    %  WARNING:  Parameters other than binary or
                    %            decimal may produce unpredictable
                    %            results!!!!

    DOM;


    compare
    MOD

        do_compare PROC ( first_field   COMPARE_STRING,
                          second_field  COMPARE_STRING)
                   RETURNS 1..4;

                    %  compares two variable length strings
                    %  and returns:
                       %  1 if first < second
                       %  2 if first = second
                       %  3 if first > second

    DOM;
```

H.1 INTRINSICS INTERFACE (Continued)

```
          debug                   %  SPRITE module to handle various types
          MOD  ENV_DEPENDENT      %  of program debugging actions such as
                                  %  monitoring, tracing, dumping, etc.

              enter PROC (debug_mod_and_proc_name  VALUE  STRING (49) );

                                  %  SPRITE generates calls to this
                                  %  routine at procedure entry
                                  %  points.

          statement PROC (stmt_location VALUE POSITION_INFO);

                                  %  SPRITE generates calls to this
                                  %  routine at statement marker
                                  %  points.

              exit PROC;

                                  %  SPRITE generates calls to this
                                  %  routine at procedure exit
                                  %  points.


              summary PROC;
                                  %  SPRITE generates one call to this
                                  %  procedure just before the stop
                                  %  run BCT. This procedure prints
                                  %  any summary debugging information
                                  %  which is required.


              error PROC;
                                  %  The debug version of err.error
                                  %  calls this routine whenever a
                                  %  program error occurs. This
                                  %  routine prints any information
                                  %  about the error.


              initialize PROC;

                                  %  SPRITE generates one call to
                                  %  this procedure at the beginning
                                  %  of the program. This routine
                                  %  does debug initialization, reads
                                  %  input specifications, etc.

          DOM;
```

H.1 INTRINSICS INTERFACE (Continued)

```
dbwrite                 %  SPRITE module to print information as
MOD                     %  a debugging aid.

     number PROC   ( label_for_number DB_STR       ,
                     number_to_print  VALUE
                                      - 9999999999999999999999 ..
                                      + 9999999999999999999999  );

               %  This routine adds your label, a
               %  blank, the number, and a semicolon
               %  to the current debug line.


     string PROC   ( label_for_string      DB_STR ,
                     string_to_print       DB_STR );

               %  This routine adds your label, a
               %  blank, the second string, and a
               %  semicolon to the debug line.


     boolean PROC   ( label_for_boolean    DB_STR ,
                      boolean_to_print     VALUE BOOLEAN );

               %  This routine adds your label, a
               %  blank, the cleartext value of
               %  the boolean, and a semicolon
               %  to the current debug line.


     hex PROC  ( label_for_hex         DB_STR ,
                 hex_to_print     UNIV DB_HEX_STR );

               %  This routine adds your label, a
               %  blank, the cleartext value of
               %  the hex string, and a semicolon
               %  to the current debug line.


     ebcdic PROC   ( label_for_ebcdic      DB_STR ,
                     ebcdic_to_print UNIV DB_STR );

               %  This routine adds your label, a
               %  blank, the ebcdic string, and a
               %  semicolon to the current line.
```

H.1 INTRINSICS INTERFACE (Continued)

```
        eol PROC
                  %  This routine prints all of the
                  %  information in the current debug
                  %  line.

      DOM;




      err                     %  SPRITE module which handles run-
      MOD                     %  time detection of certain types of
                              %  program errors as defined by the
                              %  compiler.
        error PROC  ( module_name    STRING (24),
                      locator        VALUE  POSITION_INFO,
                      type_of_error  VALUE  RUNTIME_ERRORS);

                  %  This routine prints the error
                  %  message consisting of where the
                  %  error was detected (module,
                  %  rec#, and line#), and what the
                  %  error was.

      DOM;
```

H.1 INTRINSICS INTERFACE (Continued)

```
    f_dbwrite                      %  SPRITE module to print information
    MOD                            %  as a debugging aid.

        number                     %  This routine adds your label, a
                                   %  blank, the number, and a semi-
                                   %  colon to the current debug line.
           PROC ( label_for_number       DB_STR       ,
                  number_to_print VALUE -9999999999999999999999 ..
                                        +9999999999999999999999   )
                  RETURNS  BOOLEAN;         %  always returns "true"
                                                        %  max_int

        string                         %  This routine adds your label, a
                                       %  blank, the second string, and a
                                       %  semicolon to the debug line.
             PROC  ( label_for_string       DB_STR ,
                       string_to_print      DB_STR )
                  RETURNS  BOOLEAN;         %  always returns "true"

        boolean                        %  This routine adds your label, a
                                       %  a blank, the cleartext value of
                                       %  the boolean, and a semicolon to
                                       %  the current debug line.
             PROC  ( label_for_boolean         DB_STR  ,
                       boolean_to_print  VALUE  BOOLEAN )
                  RETURNS  BOOLEAN;         %  always returns "true"

        hex                            %  This routine adds your label, a
                                       %  a blank, the cleartext value of
                                       %  the hex string, and a semicolon
                                       %  to the current debug line.
             PROC  ( label_for_hex           DB_STR     ,
                       hex_to_print       UNIV  DB_HEX_STR )
                  RETURNS  BOOLEAN;         %  always returns "true"
        ebcdic                         %  This routine adds your label, a
                                       %  blank, the ebcdic string, and a
                                       %  semicolon to the current line.
             PROC  ( label_for_ebcdic       DB_STR ,
                       ebcdic_to_print  UNIV  DB_STR )
                  RETURNS  BOOLEAN;         %  always returns "true"

        eol                          %  This routine prints all of the
                                     %  information in the current debug
                                     %  line.
             PROC
                  RETURNS  BOOLEAN;         %  always returns "true"

    DOM;
```

H.1 INTRINSICS INTERFACE (Continued)

```
move
MOD
     dc_move PROC ( module_name        STRING (24) ,
                    statement_number   0..99999    ,
                    sending_field      MOVE_STRING ,
                    receiving_field    MOVE_STRING );

               %  moves (left justified) a variable size
               %  string from the third argument to the
               %  fourth argument.  Arguments one and two
               %  specify the module name and node number of
               %  the statement performing the move.

DOM;




print                   %  BPL module to handle printer
MOD  ENV_DEPENDENT      %  output.

     line PROC  ( paper_motion_to_use  VALUE PR__PAPER_MOTION ,
                  line_to_print              PR__LINE         );

               %  This routine prints the line
               %  with the appropriate paper
               %  motion.

     print_line PROC  ( paper_motion_to_use  VALUE CHAR            ,
                        line_to_print              STRING (132) );

               %  This routine is the old
               %  interface to the PRINT
               %  module.  DO NOT USE!!
```

## H.1 INTRINSICS INTERFACE (Continued)

```
skip PROC  ( nr_of_lines_to_skip     VALUE  0..99 );

          %  This routine skips the
          %  number of lines specified,
          %  or to the top of the next
          %  page.


end_of_page PROC;
          %  This routine inhibits any
          %  other lines from being
          %  printed on the current
          %  page.


chg_page_nr PROC  ( page_nr_of_next_page  VALUE  1..9999 );

          %  This routine changes the
          %  page number of the next page
          %  to the specified value.  It
          %  also stops any more lines
          %  from being put on the current
          %  page.


chg_page_size PROC ( nr_of_lines_on_page  VALUE  1..99 );

          %  This routine changes the
          %  number of lines which are
          %  printed on a page.


chg_file_id PROC  ( new_file_id          VALUE  STRING (6) ,
            new_multi_file_id            VALUE  STRING (6) ,
            special_forms_required       VALUE  BOOLEAN    );

          %  This routine changes the file
          %  ID and special forms flag of
          %  the print file when closed.


get_file_id PROC  ( real_file_id            VAR  STRING (6) ,
                    real_multi_file_id      VAR  STRING (6) );

          %  This routine obtains the file
          %  ID of the print file.
```

## H.1 INTRINSICS INTERFACE (Continued)

```
hex PROC  ( hex_string_to_add  UNIV  PUT__HEX );

          %  This routine appends the clear
          %  text value of the hex string to
          %  the end of the current line.


blanks PROC  ( chars_to_skip_over        VALUE  0..132 );

          %  This routine skips over the
          %  specified number of characters
          %  in the current line.


eol PROC;

          %  This routine prints the current
          %  line.


new_page PROC;

          %  This routine marks the current
          %  page as being finished.


adv_to_col PROC  ( column_to_advance_to     VALUE  1..133 );

          %  This routine moves ahead (if
          %  not past it) to the specified
          %  column in the output line.


go_to_col PROC  ( column_to_go_to          VALUE  1..133 );

          %  This routine moves (in either
          %  forward or backward direction)
          %  to the specified column in the
          %  output line.


chg_line_size PROC  ( new_line_width       VALUE  1..132 );

          %  This routine changes the width
          %  of the output line.
```

H.1 INTRINSICS INTERFACE (Continued)

```
        close PROC;

                    %  This routine closes the out-
                    %  put printer file.


        get_list_date_time PROC (date_of_listing VAR  STRING (9)  ,
                                                      % (dd MMM yy)
                                 time_of_listing VAR  STRING (10) );
                                                      % (hh:mm a.m./p.m.)

                    %  This routine obtains the date
                    %  and time which appears on the
                    %  first page-header line.


        get_real_date_time PROC  (current_date    VAR  STRING (9)  ,
                                                      % (dd MMM yy)
                                  current_time    VAR  STRING (10) );
                                                      % (hh:mm a.m./p.m.)

                    %  This routine obtains the
                    %  current date and time from the
                    %  MCP.

    DOM;




    put                      %  SPRITE module to format lines.
    MOD  ENV_DEPENDENT
         string PROC  ( string_to_add           PUT__STR );

                    %  This routine appends the string
                    %  to the end of the current
                    %  line.


         number PROC  ( number_to_add  - 9999999999999999999999 ..
                                       + 9999999999999999999999,
                        container_size_of_nr  VALUE  1..132 );

                    %  This routine appends the number
                    %  to the end of the current line.
```

H.1 INTRINSICS INTERFACE (Continued)

```
          backup PROC;

                    % This routine deletes trailing
                    % blanks from the end of the
                    % output line.


          chg_indentation PROC (normal_line_indent     VALUE  0..131 ,
                                overflow_line_indent   VALUE  0..131 );

                    % This routine changes the values of the
                    % indentations for both normal lines and
                    % for overflow lines (printed without an
                    % explicit call on 'eol').

     DOM;




     readcd               % BPL module to get input card images
     MOD  ENV_DEPENDENT   % from either a card deck or an
                          % editor file.
          read_card PROC  ( next_card_image    VAR  STRING (80) ,
                            end_of_file_found  VAR  BOOLEAN     );

                    % This routine obtains the next
                    % card image from the input file.


          get_id PROC  ( name_of_input_file  VAR  STRING (6) ,
                         is_an_editor_file   VAR  BOOLEAN    );

                    % This routine returns the file
                    % id and whether the input comes
                    % from an editor file or not.


          close PROC;

                    % This routine closes the input
                    % file.

     DOM;
```

H.1 INTRINSICS INTERFACE (Continued)

```
trace                   %  BPL module to start/stop
MOD                     %  tracing.
     program PROC  ( turn_trace_on   VALUE   BOOLEAN );

                 %  This routine starts or stops
                 %  tracing, as specified.


    on_with_limits PROC (starting_address      VALUE  0..999999 ,
                         limiting_address      VALUE  0..999999 ,
                         starting_segment_nr   VALUE  0..999    ,
                         limiting_segment_nr   VALUE  0..999    );

                 %  This routine does a start trace BCT with the
                 %  specified parameters.  You may limit trace output
                 %  lines by setting appropriate values.

DOM;
```

## H.2 INTRINSICS CALLED IMPLICITLY

Some intrinsic calls are generated implicitly by the SPRITE compiler depending on certain conditions in the source program or certain options provided by the user in the source program or in the Binder Specifications. These conditions and options are described below:

| Condition or Option | Intrinsic Called | Other Intrinsics Called |
|---|---|---|
| run-time error | err | print, stop |
| debug specified | debug | arm, err, print, put readcd and trace |
| calls to dbwrite | dbwrite | put, print |
| variable length move of >100 digits or characters | move | none |
| variable length compare | compare | none |
| assignment statement or operation with mixed binary and decimal operands | convert.to_binary and/or convert.to_decimal | |

A run-time error call is always generated for a CASE statement without an ELSE clause, ASSERT statements whose expressions are false, and at the end of functions with no return value.

If the bounds-checking (BOUNDS) option (see Appendix I, Section 5.4) to the SPRITE compiler is set, a run-time error is sometimes generated for coercions and casts involving range checking, or subscripting with an expression that is not a constant.

Calls to the debug module are generated if the debug option is set within the Binder Specifications.

## H.3 INTRINSICS USED EXPLICITLY

Intrinsics may be called explicitly by the SPRITE program. A specification of the intrinsic module must then be included in the MID description of the user's program. An inter-module call (i.e. mod.proc) in the SPRITE source program may be used to explicitly call the intrinsic. The module name, procedure name, parameter types, and return value are checked according to the MID specification.

## H.4 SUMMARY OF HOW TO MINIMIZE GENERATED RUNTIME ERRORS

1. Include ELSE clauses in CASE statements.
2. Comment out ASSERT statements.
3. Make a RETURN statement be the last statement in a function.
4. Don't use variable length objects.
5. Don't use variable indices into arrays where bounds of index's type are either lower than or higher than the bounds of the array.
6. Don't use variable substring offset whose type includes values less than 1.
7. Don't use GENERATE EXTERNAL.
8. Don't use $ HIGHHEAP.
9. Don't move a string to a string of a smaller size.
10. Don't have possibly overlapping operands unless identical.
11. Don't use string to display conversions.
12. Don't move operand to operand of differing bounds.

## Appendix I SPRITE COMPILER

This appendix describes the SPRITE language compiler, based on the B2000/3000/4000 SPRITE Language, which produces machine language programs for use on B2000/3000/4000.

Features marked with an asterisk (*) are not available on the current release of this product on the field software release tape as of the issue of this specification revision level.

## REFERENCES

| CSG Std. | Title |
|---|---|
| 1955 2959 | Compiler Control Images |

| P.S. No. | Title |
|---|---|
| 1943 0198 | B2000/3000/4000 EDITOR Program |
| 1962 6951 | B2000/3000/4000 SPRITE Language |
| 1962 7009 | Type 3 ICM Product Specification |

## I.1 GENERAL DESCRIPTION

The SPRITE compiler produces Independently Compiled Modules (ICMs) to operate with the B2000/3000/4000 BINDER.

## I.2 PROGRAM DESCRIPTION

### I.2.1 LANGUAGE

The language acceptable as input to this compiler is the B2000/3000/4000 SPRITE language. (See B2000/3000/4000 SPRITE Language P.S. #1962 6951.)

### I.2.2 INPUT

The compiler uses one or more source input files to make a complete, updated SPRITE program. All input files are expressed in the EBCDIC character set.

Disk, diskpack, or magnetic tape can be specified as source language input media for a single input file or for a master file. Punched cards may also be used as a single input file.

If two input files are used, the compiler merges them on a sequence number basis.

Records of input files are 80 bytes in length. Input from disk, diskpack, or magnetic tape is blocked 9. The format of each 80-byte record is 72 characters of source information in columns 1 through 72 and an optional 8 character sequence number in columns 73 to 80.

Source input can also be an EDITOR file on disk or diskpack. EDITOR files may be used for the update and master file input. EDITOR files are unblocked. (See B2000/3000/4000 EDITOR Program Specification.)

Full upper and lower case character set input is required.

#### I.2.2.1 Library Files

SPRITE language source text which is common to several programs may be stored on disk or diskpack as source library files. One or more library files may be included in a program by use of INCLUDE CCI commands. A library file may be an EDITOR format file or a file with record length of 80 bytes and blocked 9.

#### I.2.2.2 Program Inserts

The SPRITE compiler uses the values of switches 2 and 6 to determine the hardware type of the primary input file. If the value of switch 2 is initially 0, then the value of switch 6 is moved to switch 2. The value in switch 2 is then interpreted as follows:

I.2.2.2 Program Inserts (Continued)

| | |
|---|---|
| 1 | = primary file is an editor file on disk |
| 3 | = primary file is an editor file on pack |
| others | = primary file is a card file |

If an editor file is used, a file equate must be supplied which equates the EDITOR internal file to the desired external file.

## I.2.3 OUTPUT

### I.2.3.1 New Source Language Files

An updated master symbolic output file, acceptable as input to the SPRITE compiler, may be created by use of the control card option NEW. This file may have a record length of 80 bytes and be blocked 9, or it may be an EDITOR format file. Source language output media may be disk, diskpack or magnetic tape.

### I.2.3.2 Output Listings

Listings provided by this compiler include:

Diagnostic messages

Input source language (may be inhibited)

Indication of inserted, replaced, or deleted source

Generated code, in ICM format (upon request)

Summary information

Cross reference listing (upon request)

Listings to be printed require an output device capable of full upper and lower case output.

### I.2.3.3 Generated ICMs

The ICM generated is written to disk.

ICMs can be optionally written to diskpack.

## I.2.3.4 Label Equate Information

In a multiprogramming environment, to identify source program files uniquely, it may be necessary to use label equation statements. The internal filenames and external file-identifiers for SPRITE files are:

| Internal File-name | External File-ID | Function |
|---|---|---|
| CARD | CARD | Update card input. Default device is card reader. |
| EDITOR | EDITOR | Update EDITOR input. Default device is disk. |
| SOURCE | SOURCE | Source program input. Default device is disk. |
| NEWSOU | NEWSOU | Updated source program output. Default device is disk. |
| LINE | LINE | Printed output listing. |
| ICM | ICM | ICM file. |

For example, to change the external file-identifier of an ICM file, the label equation card used is:

```
?FILE ICM = <file-id>
```

where <file-id> is a unique file-identifier of 6 characters or fewer.

## I.2.3.5 Program Standards

ICMs produced by this compiler conform to the Type 3 ICM Product Specification P.S. #1962 7009.

## I.2.3.6 Debugging and Diagnostic Facilities

The following compile time facilities are available:

A. Syntax error messages are printed following the line in error. A pointer indicates the location of the error.

B. Optional warning messages are printed following the line to which they apply. A pointer indicates location of the possible error.

I.2.3.6 Debugging and Diagnostic Facilities (Continued)

C. Various informational messages are printed.

D. No ICM is generated if syntax errors are present.

All error messages are specific and are sufficient to determine the cause of the error.

## I.3 PERFORMANCE AND EFFICIENCY

### I.3.1 COMPILE SPEED

The compiler compiles source language modules at approximately 300 to 750 card images per minute. Module Interface Descriptions are compiled at 500 to 1000 card images per minute. Compile speed is proportional to additional main memory and inversely proportional to the size of the source program.

### I.3.2 COMPILER SIZE

The minimum main memory requirement of 125 K bytes is sufficient for all compilations. Compile speed on large programs is improved if additional main memory is given. A maximum of 500 K bytes of main memory can be used.

### I.3.3 MEASURE OF OBJECT PROGRAM EFFICIENCY

The code produced by the compiler has an execution efficiency comparable to that produced by other B2000/3000/4000 compilers.

### I.3.4 CONDITIONAL COMPILATION

The code generator of the compiler detects the presence of unreachable code in certain situations and is able to suppress code generation. This suppression is transparent to the user. Example situations are when compilation takes place for a CONST expression tested in IF_THEN_ELSE statements, in the selector expression of CASE statements, WHILE loops and FOR statements.

## I.4 ENVIRONMENT

### I.4.1 HARDWARE ENVIRONMENT

The minimum hardware required for use with the compiler is:

| | |
|---|---|
| B2000/3000/4000 Processor | Must have extended address functions. |
| Main Memory | 125 K bytes for the compiler. Plus MCPVI requirements. |
| Disk or System Memory for workfiles | 360 K bytes for 1000 line program. The requirement increases in increments of 360 K bytes per 1000 lines. |
| Card Reader, or SPO | 1 for control card. |
| Input Device | 1 of the input devices specified for the source program (Section 2.2). |

Optional hardware which may be used with this compiler is:

| | |
|---|---|
| Printer | 1 for listing, 132 columns used, if user wishes to print any output. |
| Main Memory | Additional main memory provides faster compile speeds. |

Additional devices are required if requested in the control cards.

### I.4.2 SOFTWARE ENVIRONMENT

A master control program is assumed resident in main memory. Master Control Program MCPVI of a compatible release level (ASR) is acceptable.

## I.5 COMPILER CONTROL CARDS

Options are available during compilation and are activated or deactivated by control card images (CCI). These cards along with the $ character in column 1 may be interspersed at any point within the program and contain the following functions.

(*) indicates option is not implemented.

Below is a list of the CCIs and their parameters. For additional information concerning the meaning and usage of CCIs, refer to CSG Control Card Images 1955-2926.

In the following list a '+' preceding the CCI indicates that there is more information on that CCI in a later paragraph. Also, a '*' preceding 'Boolean' in the format column indicates that the CLEAR CCI does not affect that boolean valued CCI.

The phrase "settable once only" means that if the option is ever set, it may never be reset. Likewise, "resettable once only" means that once it is reset, it may never be set.

Single dollar CCIs ($) do not appear in the listing unless overridden by LIST$ or LISTDOLLAR. Double dollar CCIs ($$) always appear in the listing.

I.5 COMPILER CONTROL CARDS (Continued)

THIS OPTION AFFECTS OTHER BOOLEAN OPTIONS

| CCI | FORMAT | PURPOSE |
|---|---|---|
| CLEAR | Immediate | All Boolean options are reset (except where * preceded Boolean) |

THESE OPTIONS AFFECT THE LISTING FUNCTION

| CCI | FORMAT | DEFAULT | PURPOSE |
|---|---|---|---|
| CODE | Boolean | False | List generated code |
| + CONTENTS | Boolean | False | List string (up to 72 chars) in table of contents |
| DOUBLE | Boolean | False | Double space listing |
| + FORMAT_UNITS | Boolean | False | Special listing formatting |
| LIST | Boolean | True | List source input (and summary) |
| LIST$ or LISTDOLLAR | Boolean | False | List all single dollar CCIs |
| LISTDELETED | Boolean | False | List all DELETEd or VOIDed images |
| LISTINCL | Boolean | False | List all INCLUDEd images |
| LISTOMITTED | Boolean | Falst | List all OMITted images |
| LISTP | Boolean | False | List all primary (patch) images (if LIST is reset) |
| + MAP | Boolean | False | Show size, offset, block number of data blocks |
| PAGE | Immediate | (none) | Skip listing to top of page |
| + PAGESIZE | Value | 56 | Specify (usable) lines per page, range 6..104 |
| SUMMARY | Boolean | False | List summary information, (if LIST is reset) |

I.5 COMPILER CONTROL CARDS (Continued)

| | | | |
|---|---|---|---|
| + TITLE | Boolean | False | List string (up to 72 chars) as part of page heading |
| XREF | *Boolean | False | Gather and list cross ref- eremce; settable once cnly |

THESE AFFECT THE MERGE, NEW AND INCLUDE FUNCTION

| CCI | FORMAT | DEFAULT | PURPOSE |
|---|---|---|---|
| + INCLUDE | Immediate | (ncne) | Enable copying of file |
| + NEW | *Boolean | False | Create source output file, settable once only |
| + MERGE | *Boolean | False | Enable merging process, settable once only |
| DELETE | Boolean | False | Discard source (master) images until reset |
| INCLNEW | Boolean | False | INCLUDEd images written to NEW file |
| OMIT | Boolean | False | Ignore all source images until reset |
| + VOID | Immediate | (ncne) | Discard source images until line number is exceeded |

I.5 COMPILER CONTROL CARDS (Continued)

THESE OPTIONS AFFECT MISCELLANEOUS FUNCTIONS

| CCI | FORMAT | DEFAULT | PURPOSE |
|---|---|---|---|
| + BOUNDS | *Boolean | True | Specify bounds checking & level range 0..9, resettable once only |
| + COPYBEGIN | Special | (none) | Beginning INCLUDE symbolic range mark |
| + COPYEND | Special | (None) | Ending INCLUDE symbolic range mark |
| DEBUG | Special | (none) | Internal compiler use only |
| CORRECTOK | Boolean | False | Corrected errors ignored |
| DBSTMTCALLS | *Boolean | True | Debug statement calls generated; resettable once only |
| DEBUGCALLS | *Boolean | True | Debug calls generated, resettable once only |
| + ERRORCALLS | *Boolean | True | Error calls generated, resettable once only |
| ERRORLIMIT | Value | 100 | Specifies error limit abort, range 1..999 |
| HIGHHEAP | *Boolean | False | Specifies heap lies above stack, settable only |
| MCPVI | Boolean | False | Makes available certain new features. |
| + SEQ OR<br>+ SEQUENCE | Boolean | False | Specifies source image resequencing |
| SEQCHECK | Boolean | False | Give warning on seq. errors |
| SYNTAX | *Boolean | False | Syntax check only, no code |
| WARNFATAL | Boolean | False | Warnings are syntax errors |

I.5 COMPILER CONTROL CARDS (Continued)

In the following explanations, items in brackets are optional.

I.5.1 FORMAT_UNITS

Setting this option specifies that special formatting is desired for 'units' in MIDs and modules. A unit is a procedure, a macro, a module description, a data or file block, a DEC-CED block or a TYPE or CONST declaration occurring outside of a procedure.

Two numeric parameters are accepted. The first specifies the number of lines to skip between units and the second specifies the minimum number of lines remaining on a page before a new unit will be started on that page.

The syntax is FORMAT_UNITS [=] [skip_lines [, lines_rem]]. The defaults are skip_lines = 3 and lines_rem = 20. The parameters should not be present when resetting FORMAT_UNITS.

I.5.2 TITLE

Setting this option specifies that a legend in the heading is desired. The given string is centered on the third heading line and may be up to 72 characters long.

The syntax is TITLE [=] string. The string must be in quotes.

Resetting TITLE will clear the title line to blanks.

I.5.3 CONTENTS

The CONTENTS option has the same format as the TITLE option. However, the string you specify appears only in the table of contents at the end of the compile listing. You may use this option for easily finding things within your MIDs witout affecting your present page headings.

The syntax is CONTENTS [=] string. The string must be in quotes.

For example,

```
    $$ CONTENTS "3.7    Virtual File TYPES"
```

## I.5.4 SEQ OR SEQUENCE

Setting this option specifies that resequencing the source images (after merging) is desired. The new line numbers will appear on the listing and in the NEW file, if any.

Two numeric parameters are accepted. The first is the base line number and the second is the increment between line numbers. Both must be in the range 1..99999999.

The syntax is SEQ [base] [+ inc]. The defaults are base = 1000 and inc = 1000. The parameters should not be used when resetting this option.

## I.5.5 BOUNDS

Setting this option specifies that bounds checking code is desired. This option is initially set.

One numeric parameter in the range 0..9 is accepted and specifies the level of checking required.

The syntax is BOUNDS [=] level. The initial default for level is 9.

Resetting BOUNDS or setting the level to 0 disables all bounds checking code, INCLUDING ASSERTs!

Each level includes all checks at (numerically) lower levels. The currently specified levels and what they check are:

| | |
|---|---|
| 2 - | Check assertions |
| 4 - | Check indices, substring bounds, parametric lengths |
| 6 - | Check size conversions for parameters |
| 8 - | Check value conversions |

## I.5.6 PAGESIZE

Setting this option specifies the number of usable lines per page, which does not include the 5 header lines. A sixty line page (physical) has 55 usable lines on it.

The syntax is PAGESIZE [=] size. The default is size = 56. Resetting PAGESIZE restores the value to 56.

## I.5.7 MERGE AND NEW

MERGE enables the master and patch file merge process. NEW enables the creation of a new file from the compiled source images. Once invoked, MERGE and NEW remain set throughout

I.5.7 MERGE AND NEW (Continued)

the compilation.

Both MERGE and NEW accept two parameters. The first is the title of the file desired and the second is the physical type of the file.

The syntax is MERGE (or NEW) [title] [device]. The format of the title is a quoted string up to 13 characters long of the form family_name/file_name. The legal devices are DISK, PACK, DISKPACK, TAPE, EDITOR, EDITOR_DISK and EDITOR_PACK. EDITOR and EDITOR_DISK are synonyms, as are PACK and DISKPACK.

EDITOR, EDITOR_DISK and EDITOR_PACK specify that an editor format file is to be the master file. DISK, PACK, DISKPACK and TAPE specify that an 80 blocked 9 file is to be the master file.

The defaults are device = DISK, MERGE title = SOURCE or EDISOU for an editor format file, and NEW title = NEWSOU or NEWEDI for an editor format file for NEW. These defaults may be overridden by MCP label equation.

Examples:

```
MERGE "JUNKMY" DISK
NEW "XXXXuc" EDITOR
MERGE "MYPACK/FYLEuc" EDITOR_PACK
NEW "MYPACK/XXXX" DISKPACK
```

I.5.8 INCLUDE

This option allows inclusion of all or part of another source file into the compilation process. It functions independently of the merge process and, in fact, suspends merging while including.

The syntax is: INCLUDE title device [range]. Title and device are those mentioned in MERGE and NEW, except that INCLUDEs from TAPE are not permitted. If a range is not present, the entire named file is included. A range may be a section name (see below, COPYBEGIN and COPYEND) or it may be a sequence range which specifies the beginning line number and may optionally specify the ending line number.

Examples:

```
INCLUDE "JUNKMY" DISK
INCLUDE "XXXXuc" EDITOR 1000000
```

I.5.8 INCLUDE (Continued)

```
INCLUDE "MYPACK/YYYY" DISKPACK 1000000 3000000
INCLUDE "MYPACK/FYLEuc" EDITOR_PACK fib_declarations
```

An INCLUDEd file may contain INCLUDEs. Nesting of up to 9 levels is supported.

I.5.9 COPYBEGIN AND COPYEND

These CCIs are markers for the INCLUDE function and serve to delimit named sections in the file. They cause no action and have no other effect.

The syntax is COPYBEGIN (or COPYEND) name;. The name is an un-quoted string up to 24 characters long. The semicolon is required.

I.5.10 ERRORCALLS

This option is initially true and causes calls to the run-time intrinsic, err.error, to be generated. Resetting it replaces calls to 'err.error' with an invalid opcode, 'EC' in HEX, followed by the two digit error code.

I.5.11 VOID

This option specifies that all images from the source (master) file are to be discarded until the source line number exceeds the specified line number.

The syntax is VOID line_numb. Line_numb must be in the range 0..99999999.

Examples

```
VOID 9999
VOID 1000000
```

I.5.12 MAP

For the portion of the source code for which this option is set, the output listing lines of STRUCture definitions and DATA definitions are modified to show details of the structure of the structure or data block.

The card-image origin field of the affected output lines (normally "EDITOR", "INCLUDE", "PATCH", etc.) will contain 3 columns of information:

1. size (if BIT, then "." plus allocated bit)2.

I.5.12 MAP (Continued)

offs3. block number
(only for DATA definitions)

For example,

```
------------------------------------------------------------------
TYPE                                      01010000 EDITOR
STR1 = STRUC                        01011000 EDITOR
a BOOLEAN,          01012000      1      0
b,                  01013000     .8      1
c BIT,              01014000     .4      1
d HEX,              01015000      1      2
e CHAR              01016000      2      4
CURTS;                 01017000 EDITOR
TYPE                                      01018000 EDITOR
STR2 = STRUC                        01019000 EDITOR
f 0..999,           01020000      3      0
g STRING (99),      01021000    198      4
h STR1              01022000      8    204
CURTS;                 01023000 EDITOR
data                                      01024000 EDITOR
DATA                                      01025000 EDITOR
v1 STR1,                            01026000       8      0   54
v2 STR2,                            01027000     212      8   54
v3 CHAR,                            01028000       2    220   54
v4 HEX,                             01029000       1    222   54
v5 BOOLEAN,                         01030000       1    223   54
v6 BIT,                             01031000      .8    224   54
------------------------------------------------------------------
```

NOTE: For the most information to be supplied by this option, each DATA variable or STRUCture component must be on a separate source line.

I.5.13 SAMPLE SPRITE CONTROL CARDS

The following control cards are for a Module Interface Description compilation:

```
%?.REMOVE <icm name>.     % Optional -- closed with REMOVE
%?.REMOVE <mid name>.     % Optional -- closed with REMOVE
%? COMPILE <name> WITH SPRITE/P.   % "P" forces MID compile
%? FILE SYSTEM = <mid name>.    % File equate needed
```

## I.5.13 SAMPLE SPRITE CONTROL CARDS (Continued)

```
%? FILE ICM     = <icm name>.    % File equate needed
%? FILE LINE    = <listing name, etc>.
%? DATA CARD.
```

The SYSTEM file contains the information from the MID needed for a module compilation.

As a mnemonic, remember: "P for PROG" to compile a MID.

The following control cards are for a module compilation:

```
%?.REMOVE <icm name>.    % Optional -- closed with REMOVE
%? COMPILE <name> WITH SPRITE.    % NO "P" for modules!
%? FILE SYSTEM = <mid name>.    % File equate needed
%? FILE ICM    = <icm name>.    % File equate needed
%? FILE LINE   = <listing name, etc.>.
%? DATA CARD.
```

## Appendix J <u>CONVERTING BPL MODULES TO SPRITE MODULES</u>

Any BPL module should be carefully coded so it can easily be converted to the SPRITE language.

Small BPL modules are generally used for machine-dependent algorithms, and MCP calls. Extensive use of complicated BPL modules is hazardous, since the BPL processor is not integrated into the SPRITE MID specification/verification scheme, and doesn't understand SPRITE data types. The BPL module is described in the MID, for the sake of calls from SPRITE code. This description includes the entry points, shared data segments and parameters.

Care should be taken that data layouts agree. The BPL module should be compiled with a "CONTROL EXTENDED" source card so that parameters passed will look the same. Interfacing with SPRITE shared data blocks (or its data through parameters) may be done provided that the data mapping is known (see 13.8).

All parameters must be NAME parameters. If the parameter access (as specified in the MID) is CONST, this BPL module must not assign a value to that parameter. If the entry point specification is a function, the return value is coded as a final 'write-only' NAME parameter. If a UNIV parameter is passed then the BPL module must allow 6 digits for the length of the parameter followed by the parameter address.

The format of the reference passed is uniform for all modules of a program. It is 8-digit extended IA. The controller must be exactly as anticipated by the called module. IA and IX controllers on the actual parameter are forbidden. Index registers are preserved only by 'function' procedures.

If the module is to be overlayed, all file declarations (FIBs) and 'own' variables should be put into a single COMMON with a unique internal name. The COMMON area can be overlayed or not, as required.

The interface described here allows this BPL module to reference data that is shared with other modules in the program. A DATA segment COMMON area is declared. <DATA segment name> must be the same as the DATA name specified in the MID description. The variables in the COMMON must have types equivalent to the types of the corresponding variables specified in the MID DATA component. The COMMON variables must be declared in the same order that the variables in the MID DATA segment are declared.

## J.1 TYPE II ICMS FOR BINDER

A Type II ICM is created from BPL by placing <MODULE NAME>:BEGIN as the first source card. In addition, a leading dollar card must have the 'ICM2' option.

Immediately following the <MODULE NAME>:BEGIN card the following declarations must appear:

a. "PROG_ENTRY" is required if the module (ICM) contains the program entry point.

b. "ENTRY" is required if the module (ICM) contains any entry points that are referenced by other modules (ICMs).

c. "EXTERNAL" is required if the module (ICM) calls procedures in other modules.

No codefile is created for any ICM being compiled. A codefile is only created when one or more ICMs are bound together.

An ICM can be a single procedure or multiple procedures. All segmentation directives (i.e., SEGMENTED, UNSEGMENTED) are treated as noise words.

The following are language restrictions for creating Type II ICMs.

All global data will be allocated into a special named COMMON block.

Executable code within the outer block is disallowed.

A <module name> is required prior to the first BEGIN (i.e., <MODULE NAME>:BEGIN).

All segmentation directives are ignored.

All procedures within procedures will be part of their outer procedure's 'code block'.

Users of dynamic storage are required to utilize a BINDER-initialized pointer instead of direct addressing.

## J.2 TYPE I TO TYPE II SOURCE CONVERSION

By eliminating the @ICM declaration, deleting option $MCPB, and by including the $ option 'ICM2' plus the <MODULE NAME>:BEGIN statement the compiler will attempt to produce a Type II ICM. Any violations of the Type II requirements will be syntaxed. With a minimal number of changes the Source Type I ICM can be converted to a Source Type II ICM.

Currently the BPL processor outputs TYPE II ICMs. The ICM must be converted by a filter program into a Type III ICM. Since the format of the input into the filter program changes periodically, the user should read the latest documentation on the program.

## Appendix K GLOSSARY OF TERMS

This glossary presents a list of words and terms used in this document. For a more extensive discussion of the terms, see the (<chapter>.<section>) referred to with each term. For example, (11.5) refers to Chapter 11, Section 5.

_actual parameter_ (_16_)

An actual parameter is a variable or expression, supplied as part of a call to a procedure that replaces the formal parameter for the invocation of the procedure.

_access attribute_ (_18.2.1_)

An access attribute controls how a variable may be accessed through a particular reference to it. Every reference (access path) to a variable has associated with it an access attribute. Access may be either read-only (usually denoted by the appearance of CONST) or read/write (usually denoted by VAR).

_aggregate data types_ (_13.3_)

An aggregate (structured) data type is a data type composed of component data types. An aggregate data type is defined by describing its component types and by indicating a structuring method (e.g., array, strings, PACKED or unpacked structures, or sets.

_array_ (_13.3.1_)

An array is a data structure containing a fixed number of elements, all of the same type.

_bit_ (_13.9_)

A binary value of either 0 or 1.

_cast_ (_18.2.1.3_)

The explicit conversion from one data type to another data type.

codefile (1.0)

A codefile is a file containing executable code. It is created by binding together all independently compiled modules (ICMs) of a program.

coercion (9)

A coercion is an automatic, implicit conversion from one data type to another type.

constant definition (15.1)

A constant definition associates a name with a fixed, compile-time value. The name may be used in place of the value throughout the scope of the definition.

control variable (14.3)

A scalar variable which is given an initial value and then counted up or down in controlling a FOR statement.

data block (16.2)

A data block is a named block of storage containing one or more variables.

data block definition (16.2)

A data block definition consists of the name of a data block and declarations of the variables that it contains.

data type (15.2)

A data type specifies a collection of values that any object of that type may take on and a set of operations allowed on the object.

declaration block (12.1)

A declaration block is an unnamed block containing constant, type, data and file definitions. Declaration blocks permit a single KNOWS list to control the access to a group of related definitions in the MID.

denotation (18.2.2)

A denotation is the source text representation of a value of a particular type.

entry point (12.2)

An entry point is a procedure that may be called from other modules. Module entry points are declared in the MID.

file block (16.3)

A file block is a named block of storage containing one or more files.

file block definition (16.3)

A file block definition consists of the name of the file block and declarations of the files that it contains.

file declaration

A file declaration performs two functions. First, it causes the creation of a File Information Block (FIB) and associates with it 1) a collection of attributes which describe the characteristics of the physical file, and 2) a type which describes the records of the logical file. Secondly, it associates with the file a name by which it may be referenced in the text of a module.

formal parameter (16 and 18.2.1.4)

An identifier declared to be a parameter in a procedure or function declaration. When the procedure or function is called later, the formal parameter will be replaced by an actual parameter.

function (18.2.1.4)

A function is a procedure that returns a value of a fixed type.

identifiers (6)

Identifiers are names denoting constants, variables, various kinds of files, procedures and functions. They are composed using lower-case letters (a..z), digits (0..9) and the underscore identifiers (6) (Continued)

character.

indicants (6)

Indicants are names denoting user-defined data types, file attributes and their mnemonic values. These are composed using upper-case letters (A..Z), digits (0..9) and the underscore character.

intrinsics (Appendix H)

These are already-compiled routines which provide debugging, I/O, and other miscellaneous functions for a SPRITE program. A program that makes an explicit call to an intrinsic must declare its interface in the MID.

knows list

A knows list is a list of those modules that have access to a module, an entry point procedure, data block, constant definition, type definition or declaration block declared in a MID. The purpose of a knows list is to control the access to components and the scope of names.

MID

Associated with each SPRITE program is a Module Interface Description (MID). The MID describes the definitions held in common and the interface requirement for a SPRITE program.

module

A module is the basic unit of compilation. It consists of one or more procedures and zero or more constant definitions, type definitions, file definitions and data definitions.

module-local names (10.2)

These names are established and known within the scope of a single SPRITE module. They may be known to one or more procedures within the module. Examples of module-local names are the names of procedures, constants, data types, and data blocks declared within the individual module.

parametric data types (15.3)

A STRING or ARRAY(*) data type whose size attribute is allowed to vary as different objects of either type are created. Parametric data types are established using parametric type definitions.

pointer (13.4)

A pointer is a variable whose value is a reference to an object of a particular type. A pointer variable may have a value (nil) which references no object.

procedure (16)

A procedure specifies a named block of code and its associated data. Procedures are named or identified via procedure definitions. Procedures exist within the scope of SPRITE modules and may be invoked from other points in a program. Some procedures act as functions and return values.

procedure definition (16.1)

This definition associates an identifier with a block of code and its data.

procedure-local names (10.3 and 16.1)

These names are established and known only within a single SPRITE procedure. Local variables are examples of such names. The names involved may have different meanings in other procedures because names need not be unique if their scopes do not intersect. Procedures which do not share module-local or program- local names may redefine those names.

program

A SPRITE program is composed of one or more modules which are compiled independently. Each SPRITE program has associated with it a MID component which specifies the allowable interactions among SPRITE modules and defines constants, types, and data blocks which may be used throughout the program.

program entry point (16)

The procedure to be invoked when a SPRITE program begins executing. Exactly one entry point must be declared in the MID.

program-local names (10.3)

These names are established in the MID component of a SPRITE program. These names may be known to one or more modules of a SPRITE program. Examples of program-local names are the names of data blocks and the variables they contain, and the names of modules.

reserved word

A word within the SPRITE language system which has a predefined meaning and may not be redefined.

scalar

A simple, unstructured data type, Boolean, symbolic range, numeric subrange, and character may be scalar types.

scope (10)

Scope is a property associated with a name. The scope of a name is the extent of program text in which the name is known. A name must be unique within its scope.

set (13.3.4)

A set type defines the set of values that is the power set of its base type, i.e. the set of all subsets of values of the base type.

SHARES declaration (15.5)

This declaration enables procedures to access variables in a data block or files in a file block. The SHARES declarations appear within procedure definitions.

SPRITE

SPRITE is a procedural, statement-oriented, strongly typed systems implemention language.

standard functions (Appendix A)

Standard functions are predefined procedure-like features provided by the language to perform certain common tasks.

standard module (Appendix A)

The standard modules are predefined module-like features provided by the language to access common capabilities provided by the operating system.

statement-local names (10.4)

These names are known only within the context of a single statement. FOR and FIND statement control variables and the UNTIL- CASE statement situation names are statement-local names. These control variables and situation names may not be used in any other way within the containing procedure, but they may have a different meaning in other procedures.

string (13.3.2)

A string is a sequence of characters or hexadecimal digits of a particular positive length.

structure (13.3.3)

A structure is a datatype consisting of a collection of components, called "fields". A structure may have one or more variant parts.

subrange (13.2.2)

A subrange defines a contiguous subset of values of an ordered scalar base type. The operations on values of a subrange are inherited from the base type.

symbolic range (13.2.1.6)

A symbolic range is a collection of names enumerating all values of the type. The values may or may not be ordered.

<u>type definition</u> (<u>15</u>.<u>2</u>)

A type definition is used to define a SPRITE data type. A type definition associates an indicant with a type description.

<u>undefined</u>

The value of a variable is said to be undefined if no value has yet been assigned to it, or after conclusion of certain operations.

<u>variant</u>

A means whereby a portion of a structure declaration may have several different meanings. A structure may have one or more variant parts. Alternative field lists are grouped and identified by constants of a particular finite type. Accessibility to each variant field list is determined by a tag field. When the value of the tag field corresponds to a particular constant that identifies a variant, that variant's field list may be accessed. A run-time error will result if the field accessed is not in the variant specified by the tag field's value (*).

<u>variable declaration</u> (<u>15</u>.<u>4</u>)

A variable declaration performs two functions. First, it causes creation of a run-time object (a variable) and associates with it a collection of possible values (its type) that it may have. Secondly, it associates with the variable a name by which it may be referenced in the SPRITE source text.